Eurographics Symposium on Parallel Graphics and Visualization (2008)
J. Favre, K. - L. Ma, and D. Weiskopf (Editors)

Acceleration of Opacity Correction Mechanisms for Over-sampled Volume Ray Casting

Jong Kwan Lee and Timothy S. Newman

Department of Computer Science, University of Alabama in Huntsville, U.S.A.

Abstract

Techniques for accelerated opacity correction for over-sampled volume ray casting on commodity hardware are described. The techniques exploit processing capabilities of programmable GPUs and cluster computers. The GPU-based technique follows a fine-grained parallel approach that exposes to the GPU the inherent parallelism in the opacity correction process. The cluster computation techniques follow less finely-granular data parallel approaches that allow exploitation of computational resources with minimal inter-CPU communication. The performance improvements offered by the accelerated approaches over opacity correction on a single CPU are also exhibited for real volumetric datasets.

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

1. Introduction and Background

Ray-casting methods to visualize volumetric data are used in many application areas such as engineering, medicine, and geosciences. Volume ray casting (VRC) [Lev88, GM96, WMG98] methods can produce high quality renderings that aid in discovery of valuable information. For example, they can allow discovery of internal defects in manufactured parts and anatomical structure locations in humans. VRC creates a rendering by casting rays from the image plane into the volume of data and integrating lighting effects along the rays. Typically, VRC uses discrete sample composition, for example, front-to-back (FTB) composition, to approximate continuous integration. The FTB composition, which we use here, can be described for one ray as:

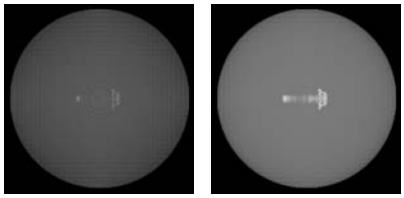
$$\mathbf{I}_F = \sum_{i=1}^{m} \left(I_i \times \prod_{j=1}^{i-1} (1-\alpha_j) \right), \qquad (1)$$

where $\mathbf{I}_F$ is the final composited intensity associated with the ray, m is the sample count along the ray, I_i is the i^{th} sample's intensity, and α_j and $(1-\alpha_j)$ are the j^{th} sample's opacity and transparency, respectively. The light intensity, I_i, at a sample is often taken as $I_i = C_i \times \alpha_i$, where C_i is the sample's color. In Eqn. 1, sample indices increase with distance from the viewer (e.g., I_1 is the intensity of the closest sample to the viewer along the ray).

An advantage of a high sampling rate (i.e., sampling at less than unit distance between samples) in VRC is that it can produce results more similar to continuous ray integration. In particular, certain artifacts can be avoided or reduced. For example, a high sampling rate can be used to avoid ringing (i.e., aliasing) artifacts [RGW*03].

1.1. Opacity Correction for Over-sampled VRC

However, compositing more than one sample per cell in VRC can produce another sort of rendering artifact—over-composition of opacities. Over-composition produces renderings that are overly-influenced by certain cells in the dataset. Correction is possible, however, using opacity correction mechanisms within VRC. Fig. 1 illustrates an over-composition problem for a dataset that contains a bar-shaped object inside a sphere. In Fig. 1 (a) and (b), 5 times over-sampled VRC renderings without and with opacity correction, respectively, are shown. Some parts of the bar-shaped object inside the sphere are not visible in the rendering without opacity correction (that object is more apparent in the rendering that uses opacity correction). The rendering without correction in Fig. 1 (a) suffers from the over-composition of opacities; along most rays, the transparency reaches zero before samples from the region containing the bar-shaped object are taken.



(a) (b)

Figure 1: *Rendering of same scene: (a) without opacity correction and (b) with opacity correction*

Two classes of opacity correction mechanisms are known. Both are independent of the interpolation-classification order. One follows from a data homogeneity assumption; it is provably correct if there is homogeneity. It was first described by Lacroute [Lac95] for shear-warp factorization. A variant (but equivalent) formulation for it was shown by Lichtenbelt et al. [LCN98]. The homogeneity the formulation assumes is on a cell basis; it assumes all samples within a cell have the same opacity (and color) values. The formulation is:

$$\alpha' = 1 - \sqrt[N]{1-\alpha}, \tag{2}$$

where N is the over-sampling rate, α is the original opacity, and α' is the corrected opacity. In this paper, the homogeneity-assuming correction mechanism will be denoted as *classic opacity correction*.

The second class of opacity correction does not rest on the assumption of data homogeneity within cells. A mechanism of this type is our Correct-to-First (CTF) opacity correction [LN07]. The CTF was aimed at a scenario where the first sample per cell is at the face of the cell. Its basis is unit-sampling that takes samples at cell faces.

The CTF opacity correction within a cell follows from Eqn. 3, which expresses that the corrected N-times over-sampling has the same opacity as unit-sampling.

$$\begin{aligned}
&(-1)^0 \left(\prod_{k=1}^{N} \alpha_k \right) p^N + \\
&(-1)^1 \left\{ \sum_{t_1=1}^{N} \left(\prod_{\substack{k=1 \\ k \neq t_1}}^{N} \alpha_k \right) \right\} p^{N-1} + \\
&\vdots \\
&(-1)^{N-1} \left\{ \sum_{t_1=1}^{1} \cdots \sum_{\substack{t_{N-1}= \\ t_{N-2}+1}}^{N} \left(\prod_{k \neq t_1, \cdots, t_{N-1}}^{N} \alpha_k \right) \right\} p^1 + \\
&(-1)^N \alpha_1 \, p^0 = 0,
\end{aligned} \tag{3}$$

where α_k is the k^{th} opacity within a cell, p is the factor that corrects opacity for the cell, and $N \geq 2$. CTF opacity correction first solves Eqn. 3 for the opacity correction factor, p, for a given over-sampling rate. It then uses p to re-scale all opacities within each cell (e.g., $\alpha'_k = p\,\alpha_k$). To avoid the difficulty of evaluating Eqn. 3 when the sampling rate is high (i.e., to avoid solving high-order polynomials), the approach approximates the high-order polynomials by fitting second degree polynomials. These can be straightforwardly fit (i.e., using the quadratic formula, $A^2 p + Bp + C = 0$). The CTF formulation can likely be extended to higher degree polynomial approximations (which might improve its accuracy).

For all opacity corrections, using either direct computation of the correction factor or a lookup table scheme (indexed by opacity) is possible. However, using a table may trade off accuracy for speed due to the discretization imposed by the table. In this paper, direct computation is used.

Use of either class of opacity correction has been reported to yield roughly comparable quality, although CTF is apparently better in situations where the first sample value is very small [LN07]. The classic correction has also been reported to run slower than CTF on a standard CPU.

In this paper, two classes of techniques for efficient realizations of the known opacity correction mechanisms in over-sampled volume ray casting using commodity hardware are introduced. The first class enables exploiting a programmable graphics processing unit (GPU). The second class enables exploiting cluster computing for environments lacking programmable GPUs. We are unaware of any prior reports of parallelized opacity correction.

The paper is organized as follows. In Section 2, related work is discussed. The acceleration techniques introduced here are described in Section 3. Experimental results are shown in Section 4. Section 5 concludes the paper.

2. Related Work

The classic correction mechanism has been used or discussed previously in some reports (e.g., [MHB*00, Pfi05]). A variation on the classic mechanism that corrects final color (i.e., instead of opacity) has also been developed by Schulze et al. [SKLE03] for use in under-sampled VRC.

Many techniques for achieving high performance in direct volume rendering have been reported, including techniques aimed at exploiting custom hardware [RPSC99] and desktop streaming media capabilities (e.g., [Kni00]). Multiprocessing techniques for volume rendering have also been investigated (e.g., [GM96]). Early work in parallel volume rendering has previously been surveyed by Wittenbrink [Wit98]. Some of the more recent works in parallel volume rendering include methods based on perspective shear-warp factorization on cluster computers [SL02] and based on combining ray-casting and object-order processing to allow very large datasets to be efficiently processed on supercomputers [CDM06].

Computation using programmable GPUs has been utilized in graphics as well as in an increasing number of more general-purpose applications. Some recent applications of GPU-based techniques for volume visualization have also been reported. For instance, some fast volume ray-casting techniques that can skip over empty spaces have been reported (e.g., [EKI06, KEI07, KSSE05, KW03]). Two of these ([EKI06] and [KEI07]) also feature ways to efficiently process regions of homogeneity. Another ([KSSE05]) includes over-sampling in high gradient regions to reduce aliasing. One ([KW03]) includes a mechanism for early ray termination. In addition, a technique for efficient tetrahedral grid data visualization using GPU-based barycentric coordinate system sampling has been presented [GW06]. Some works that utilize clusters of GPUs for very fast volume rendering have also been presented (e.g., [CMCL06, MSE06, SMW*04]). Castanié et al. [CMCL06] have described a system that exploits distributed shared memory using a cluster of GPUs for interactive viewing of volume renderings of very large datasets. Weiskopf [Wei06] has well-summarized recent work in GPU-based visualization.

To our knowledge, high performance opacity-corrected over-sampled VRC has not been reported previously.

3. Accelerated Opacity Correction

The parallel opacity correction techniques introduced here enable exploitation of parallelism (1) on programmable GPUs and (2) on cluster computer systems lacking programmable GPUs. Since VRC does not require cast rays to share intermediate computation results produced during volume traversal, it naturally maps well to data parallel strategies. The techniques described here utilize data parallelism.

Our work is built on shear-image order volume ray casting [WBLS03] for rectilinear volumetric datasets. Shear-image order VRC provides a high image quality that is equivalent to the full image order VRC, but it provides this at a lower computational cost. Also, its memory access has a spatial coherence that is similar to the shear-warp factorization [Lac95]. In shear-image order VRC with unit-sampling, samples are taken only on "faces" of the cells that make up the data lattice (e.g., if ray direction is nearly parallel to the z-axis, samples will be taken at $z = 0, z = 1, z = 2, \cdots$).

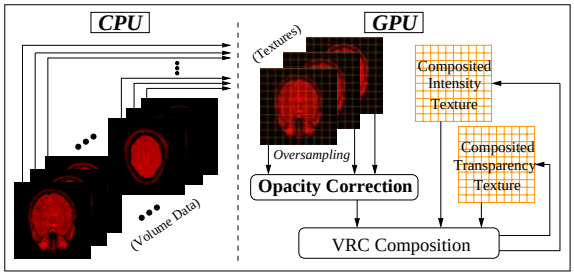


Figure 2: *Illustration of GPU-based opacity-corrected VRC*

Our techniques include early ray termination and zero-cell skipping features. To our knowledge, these features were not part of the original reports of opacity correction (e.g., in [LCN98] or [LN07]). Also, dependancy analysis followed by term rearrangement and term reductions are used in the CTF's quadratic fitting of the correction factors for both GPU- and cluster-based techniques so that the performance is not inhibited by the dependency or redundancy.

3.1. Programmable GPU-based Opacity Correction

Standard volume ray casting on a CPU typically involves interpolating data values at samples that do not fall exactly on dataset elements. Often, the interpolation is done by linear (or trilinear) interpolation. One computational advantage of shear-warp/image order style VRC over standard VRC is that its linear-class interpolations can be computed more quickly due to the known patterns of sample positions.

Using shear-image order VRC, the process of traversing rays through the volume can also be done straightforwardly in a slice-by-slice way. (Here, each "slice" is a cross-sectional plane of the data lattice.) Slice-by-slice processing is advantageous for the GPU since it can be achieved by texture-based operations; the GPU's high bandwidth and computational power in texture-based operations enables fast performance.

Our GPU-based opacity-corrected VRC is performed in an iterative way in which three dataset slices are used in VRC composition at each iteration. Initially, the compositions for the samples between the first two slices are computed by taking the first three dataset slices and mapping them to textures on the GPU. Two 2D arrays are also mapped to textures on the GPU to store the rays' composited intensities (which are initialized to 0.0) and transparencies (which are initialized to 1.0). Then, using the first two slice textures, over-sampling is performed via interpolation on the GPU to create $N-1$ intermediate textures, where N is the over-sampling rate. (Trilinear interpolation is done for non-axial-perpendicular rays, otherwise linear interpolation is done.) Next, opacity-corrected VRC composition is applied using the first slice texture and the $N-1$ intermediate textures. The opacity-corrected VRC composition includes opacity correction, (central differencing) gradient computation, and transparency and intensity composition. (Later in this section, we describe how the opacity correction activity is achieved. We note that the third slice texture is used only in central differencing gradient computation and that the zero-cell skipping and opacity- and geometry-based early ray terminations are employed in the transparency and intensity compositions.) For the rest of the slices, these steps are iterated such that one additional dataset slice is mapped to a GPU texture per iteration. In each new iteration, the prior iteration's second and third slice textures are re-used as the new iteration's first and second slice textures, and one new slice texture is added. The composited intensity and

opacity textures are also fed back for continued composition. The iterative slice-to-texture mapping allows our GPU-based opacity-corrected VRC to process volumetric datasets that are too large to be mapped onto the GPU textures at the same time.

The zero-cell skipping is handled by not performing opacity-corrected VRC composition within a cell when all texture elements within it are zeros. Early ray terminations are handled by not performing the opacity-corrected VRC composition for rays that leave the volume or when a ray's transparency reaches 0.0001. For exiting rays, padding of textures with dummy elements is applied to allow coherent operations on all texture elements. The incorporation of zero-cell skipping and early ray termination speeds up performance only marginally.

Once slices have been considered, the composited intensity texture result is displayed as the final rendering on the GPU.

Fig. 2 illustrates some details of the GPU-based processing. The figure shows three slices mapped at a time to GPU textures. (GPU textures contain orange grid overlays in the figure.) The over-sampling and the opacity correction (e.g., classic or correct-to-first correction) followed by the VRC composition steps are applied using textures on the GPU.

The key part of our technique is performing the mathematical operations necessary to correct the opacities on the GPU. For the CTF opacity correction, all elements of the N textures within a cell are used to determine the coefficients for the formula in Eqn. 3. The formula is fitted by a second degree polynomial and then the fitted polynomial is solved for the correction factor p on the GPU. Since the correction factor p requires all the between-slice samples along one ray for its computation, all elements of N textures are used at the same time to correct the opacity on the GPU. In short, since the same processing operations for the fitting and solving can be applied on each texel, the operations map naturally to and run quickly on the GPU. Here, we note that conditional expressions to prevent divide-by-zero and a negative discriminant in the quadratic formula are included in the CTF correction. For the classic correction, since each corrected opacity

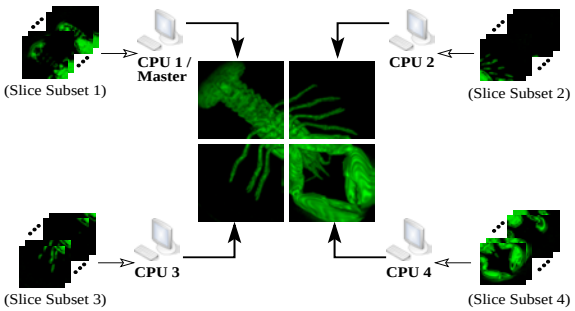


Figure 3: *Illustration of static load balanced opacity-corrected VRC*

is independent of other samples, the correction is applied one texture at a time, with all of each texture's elements being computed at the same time on the GPU (i.e., opacities are adjusted using the Eqn. 2, which is able to well-leverage the hardware support for the N^{th} root operation).

3.2. Cluster Computer-based Opacity Correction

The second set of techniques described here considers opacity correction on cluster computers lacking programmable GPUs. One of these techniques uses a static load balancing scheme. The other uses a dynamic load balancing scheme. Other VRC parallelizations using opacity corrections are possible. Our emphasis here is enabling parallel opacity correction and measuring its performance.

It is natural to utilize data parallelism for opacity-corrected VRC since it involves ray composition steps in which the rays are independent of each other; each ray can apply opacity correction and composition independently of other rays using only the portion of the volume that the ray passes through. Thus, our cluster-based techniques divide the volumetric dataset among the CPUs. As suggested earlier, the subdivision is dependent on ray direction (i.e., viewing direction) since each ray accesses only the portion of the volume it passes through. Our subdivision scheme divides the volumetric dataset in the dimensions that are perpendicular to the ray direction. (If the ray direction is not perpendicular to any of dataset dimensions, the volume is resampled with respect to the ray direction. This step is done as a pre-processing step, for reasons which will be discussed in Sec. 4.) Data for a contiguous bundle of rays are then sent to each processor. Each processor performs opacity-corrected VRC and returns its portion of the 2D rendering to a master processor. The master gathers all portions of the rendering and outputs the rendering to display. The work assignment is governed by either of two load balancing schemes, as described next. Neither of the schemes requires pre-processing steps to estimate work load.

```
1.  Assign a data subset to each slave
2.  Do {
2a.     Wait to receive a subset result
2b.     Assign new data subset to idled processor
2c.     Gather returned results
2d. } until no subsets remain
3.  For all slaves :
3a.     Send terminate message to slave
4.  Render final image
```

(a) Master

```
1.  While terminate message is not received :
1a.     Blocking receive a data subset
1b.     Apply opacity−corrected VRC
1c.     Blocking send processed image to master
```

(b) Slaves

Figure 4: *Steps of dynamic load balanced opacity-corrected VRC: (a) on master and (b) on slaves*

Static Load Balancing. The static load balancing scheme divides the volume into equal-sized subsets, and each processor performs opacity-corrected VRC for one subset. In particular, one processor divides the data into subsets and assigns subsets to other processors (while keeping a subset for itself). Then, each processor performs opacity-corrected VRC for its assigned subset. Thus, there is only one data subset assigned to each processor.

Fig. 3 illustrates the static load balancing for a 4-processor configuration. In the figure, one equal-sized data subset is assigned to each processor. Then, each processor performs opacity-corrected VRC to produce a portion of the final rendering. The master gathers the portions for final display.

We use blocking sends and receives for the synchronizing communications. Although this entails some communications overhead between processors, the total processing time is strongly dominated by the computation time for the opacity-corrected VRC. In addition, the gathering communications depend only on the 2D rendering's size rather than dataset size or over-sampling rate.

Dynamic Load Balancing. The dynamic load balancing scheme involves first dividing the dataset into small subsets (e.g., $8\times8\times Z$ or $16\times16\times Z$ subsets, where Z is the number of data slices along the rays). The subsets are assigned one at a time, each to an idle processor. Processors perform the opacity-corrected VRC and return their portion of the final rendering to the master processor. As each processor becomes idle, a new subset is assigned, until all subsets have been processed. The master gathers the renderings and displays the final rendering once all subset results are produced.

Fig. 4 outlines the dynamic load balancing scheme's steps. We use blocking sends and non-blocking receives on the master processor and blocking sends and receives on the slave processors for the communications. The approach has been implemented using MPI which allowed use of the MPI_Waitsome() on the master to handle multiple return communications from the slaves.

4. Experimental Results

In this section, experiments to benchmark the techniques' performances are described. First, the GPU-based technique is considered. Then, the cluster computation techniques are considered. In the results presented here, for the GPU-based technique, interpolation is done on the GPU, and the time to interpolate is included in the reported times. Interpolation time is not included for any CPU-based technique results, though. For the GPU, performing interpolation on the GPU results in fastest overall opacity-corrected VRC times. (It is slower to do interpolation on the CPU first due to the time to transfer the interpolation textures onto the GPU.) For the CPU-based approaches, we report just opacity-corrected VRC times (exclusive of interpolation, with interpolation done as a pre-processing step).

4.1. GPU-based Acceleration

The GPU-based technique was tested on a 512 MB NVIDIA GeForce 7950 GT. Cg version 1.4.1 was used for the implementation. For comparison, CPU-based implementations were executed on the (unloaded) CPU (Intel Core Duo 3.20 GHz) of the same PC. The PC has 3.2 GB RAM and runs Linux. Executables were built using gcc version 4.0.2. Both classic and CTF opacity corrections were tested. Five real datasets were tested: a $256\times256\times72$ dataset called Brain 1, a $256\times256\times128$ dataset called Brain 2, a $256\times256\times62$ dataset called Monkey, a $256\times256\times128$ dataset called Head, and a 256^3 dataset called Engine. The last three are from Roettger's Volume Library.

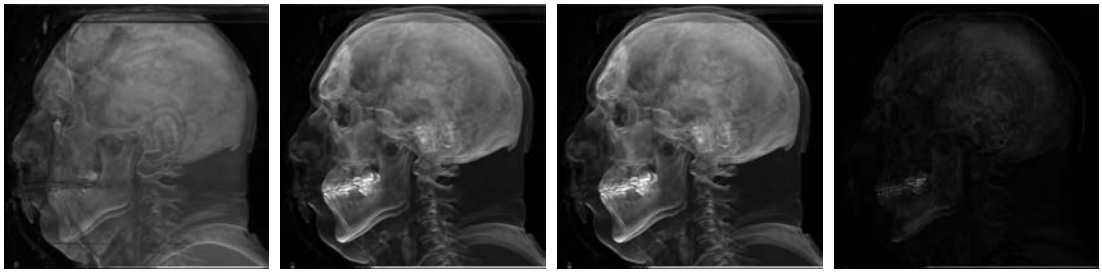
Tables 1 and 2 show the execution times of the opacity-corrected VRC composition on the CPU and the GPU, respectively, for the five datasets. Here, ray directions were axially-perpendicular. Datasets were over-sampled N= 2, 3, 4, and 5 times. For the N=5 case, the number of samples ranged from 19,988,480 (for the Monkey dataset) to 83,558,400 (for the Engine dataset). For these datasets, the CTF opacity correction was about 2 to 4 times faster than

Correct.	N	Brain 1	Brain 2	Monkey	Head	Engine
without	2	0.296	0.548	0.215	0.392	1.296
classic		2.020	4.152	0.646	3.210	3.513
CTF		0.533	0.956	0.296	0.759	1.525
without	3	0.386	0.760	0.238	0.536	2.573
classic		3.015	6.213	1.001	4.814	6.102
CTF		0.843	1.524	0.429	1.181	3.115
without	4	0.481	1.175	0.276	0.728	3.028
classic		4.006	8.521	1.245	6.537	7.764
CTF		0.873	1.711	0.493	1.333	3.525
without	5	0.700	2.132	0.351	1.200	3.688
classic		5.069	11.217	1.593	8.512	9.665
CTF		1.219	2.987	0.660	2.189	4.385

Table 1: *Opacity-corrected ray composition execution times (in seconds), N=over-sampling rate, on the CPU*

Correct.	N	Brain 1	Brain 2	Monkey	Head	Engine
without	2	0.0183	0.0318	0.0157	0.0317	0.0627
classic		0.0192	0.0336	0.0166	0.0334	0.0661
CTF		0.0199	0.0349	0.0172	0.0346	0.0687
without	3	0.0196	0.0342	0.0169	0.0341	0.0675
classic		0.0214	0.0374	0.0184	0.0373	0.0738
CTF		0.0220	0.0385	0.0190	0.0383	0.0760
without	4	0.0211	0.0369	0.0182	0.0367	0.0728
classic		0.0221	0.0388	0.0191	0.0387	0.0769
CTF		0.0257	0.0446	0.0208	0.0451	0.0879
without	5	0.0215	0.0375	0.0185	0.0374	0.0741
classic		0.0280	0.0492	0.0241	0.0491	0.0984
CTF		0.0329	0.0599	0.0268	0.0581	0.0994

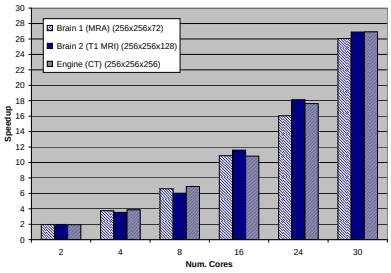
Table 2: *GPU-based opacity-corrected ray composition execution times (in seconds), N=over-sampling rate*



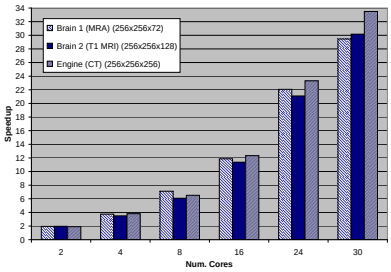
(a) (b) (c) (d)

Figure 5: *GPU-based VRC renderings of (human) Head for over-samplings (5 times): (a) without opacity correction, (b) with classic correction, (c) with CTF correction, and (d) scaled absolute difference image of (b) and (c).*

classic opacity correction on the CPU. On the GPU, the classic and CTF opacity corrections executed in about the same amount of time, although the classic correction was slightly faster than the CTF correction. The overhead of incorporating opacity correction into GPU-based VRC ranged from about 5% more time (for N=2) to about 40% more time (for N=5). The GPU is substantially faster than the CPU at opacity-corrected VRC; it was on the order of 100 times faster at classic opacity correction and about 30 times faster at CTF opacity correction. While GPU-based opacity correction is well-suited for both correction mechanisms, it appears to be modestly better for the classic correction. We believe that the absence of conditional expressions in the computation of classic correction resulted in fewer pipeline stalls on the GPU. Also, the classic opacity correction benefits from the GPU's built-in support for its expensive N^{th} root operation. Both opacity corrections are reasonably well-suited for the GPU, though.



(a) Speedups using classic correction



(b) Speedups using CTF correction

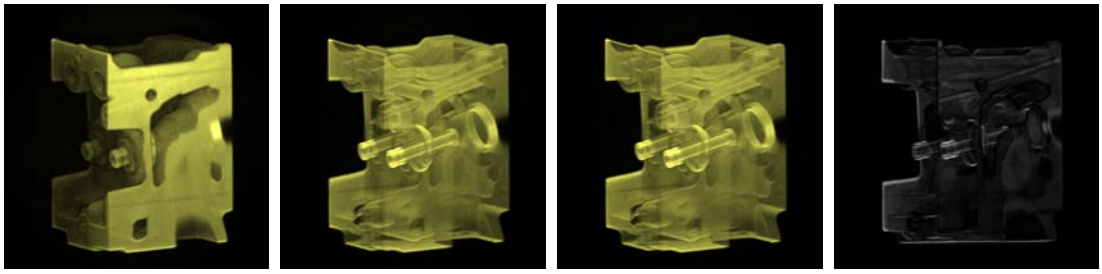
Figure 6: *Speedups for opacity-corrected VRC, static load balanced cluster computation, using (a) classic opacity correction and (b) CTF opacity correction.*

Fig. 5 shows example GPU-based VRC renderings of the Head (CT) dataset. The figure shows VRC renderings without opacity correction and with each of the correction mechanisms. While the VRC rendering without any opacity correction (Fig. 5 (a)) suffered from over-composited opacities, the VRC renderings using the opacity corrections (Fig. 5 (b) and (c)) suffer less; they allow more internal details to be observed. Fig. 5 (d) shows the scaled absolute difference image of the opacity-corrected renderings. The differences are most noticeable near the teeth and top back of head. The GPU-based renderings in Fig. 5 are visually indistinguishable from results produced using the single CPU system (i.e., there were no visible differences between the renderings of the CPU-based and GPU-based VRCs). Some pixels had very small measurable differences, however—at the 6th digit after the decimal point.

4.2. Cluster-based Acceleration

The cluster computation techniques were tested on a cluster computer with 16 nodes. Each node consists of an Intel Core Duo 3.00 GHz CPU with 1 GB RAM and runs Linux. Executables were built using mpicc of LAM version 7.1.1. Our experiments used 2-, 4-, 8-, 16-, 24-, and 30-core configurations. (To avoid inter-communication effects on the node with the master, only less than 32 cores were tested.) Tests on our cluster-based approaches enable study of opacity correction's scale-up.

Fig. 6 shows, for 5 times over-sampling of 3 datasets, the speedups for the static load-balanced technique for the classic and CTF opacity corrections. The speedup increases approximately linearly as the number of cores increases for these datasets. For 24 cores, speedups of about 17.6 for classic correction-based VRC and of about 22.0 for CTF-based VRC were observed. We also note that the CTF-based VRC was consistently about 3.0 times faster than the classic



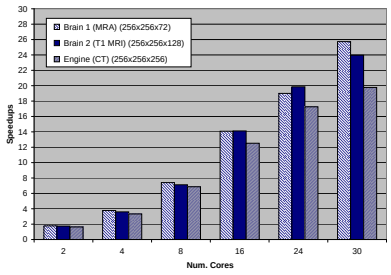
(a) (b) (c) (d)

Figure 7: *VRC renderings of Engine on cluster computer system for over-samplings (5 times): (a) without opacity correction, (b) with classic correction, (c) with CTF correction, and (d) scaled absolute difference image of (b) and (c).*

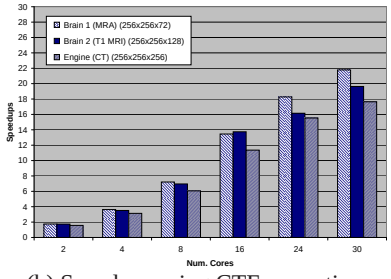
correction-based VRC. For 30 cores, a superlinear speedup was observed for CTF-based VRC (from caching effects).

Example renderings using a yellow material color (and a difference image) for the static load balanced cluster computation technique are shown in Fig. 7. Differences are especially noticeable near structure boundaries.

Fig. 8 shows, for the same cases in Fig. 6, the speedups for the dynamic load-balanced technique for the classic and CTF opacity corrections. The number of cores shown in Fig. 8 are the number of (slave) cores performing the opacity-corrected VRC composition. For 24 cores, speedups of about 15 to 19 were observed. For 30 cores, speedups of 23.2 for classic correction-based VRC and of 19.7 for CTF-based VRC were observed. While the classic correction-based VRC achieved higher speedups than the CTF-based VRC, the CTF correction was always faster than the classic correction (e.g., it was 2.6 times faster using 16 cores). As shown in Fig. 8, the dynamic load balancing scheme had better performance improvement for smaller datasets.



(a) Speedups using classic correction



(b) Speedups using CTF correction

Figure 8: *Speedups for opacity-corrected VRC, dynamic load balanced cluster computation, using (a) classic correction and (b) CTF correction.*

We note that the master processor in the dynamic load balancing scheme does not perform any opacity-corrected VRC composition. However, it is possible to have the master do a little VRC composition work for a small region. We have found that adding work to the master provides a substantial benefit only for configurations with few cores. For example, having the master processor perform some VRC composition improved performance by about 10 to 20% using 2 to 4 cores and by about 1 to 2% using 8 to 16 cores. We believe that the time the master processor spends handling both the communication and some VRC composition resulted in some overload on it as well as synchronization issues (when there were many cores), delaying processing of slave results. The implementation of such a dynamic load balancing scheme is also complex; it may not be worthwhile to assign work on the master for such small improvements.

While both load-balancing schemes are well-suited for the opacity-corrected VRC, the static load-balancing scheme has less overhead; the communication overhead to achieve a balanced load in dynamic load-balancing was more than its benefit of more evenly-balanced load.

5. Conclusion

We have presented efficient techniques for over-sampled opacity-corrected volume ray casting. One class of technique achieves efficiency using fine-grained parallelism on a programmable GPU on a single CPU system. For this technique, classic correction is marginally faster than CTF but both achieve about 15 f.p.s. for a scenario of a 256^3 dataset with two times over-sampling. A second class of techniques exploits less finely-granular parallelism using static and dynamic load balanced cluster computation. For this class of technique, CTF can achieve about 14 f.p.s. for the same

scenario. Known opacity correction mechanisms in over-sampled volume ray casting were shown to achieve efficient performance on a GPU and clustered CPUs. This paper is, to our knowledge, the first demonstration of parallel opacity correction for over-sampled VRC.

In the future, we hope to achieve further efficiency improvements using a cluster computer system that contains programmable GPUs and to test use of higher-order approximating polynomials.

Lastly, we note reviewer comments improved this paper.

References

[CDM06] CHILDS H., DUCHAINEAU M., MA K.-L.: A scalable, hybrid scheme for volume rendering massive data sets. In *Proc., EG Par. Graphics and Vis. '06* (2006), pp. 153–161.

[CMCL06] CASTANIÉ L., MION C., CAVIN X., LÉVY B.: Distributed shared memory for roaming large volumes. *IEEE Trans. Vis. and Comp. Graphics 12*, 5 (2006), 1299–1306.

[EKI06] ES A., KELES H., ISLER V.: Accelerated volume rendering with homogeneous region encoding using extended anisotropic chessboard distance on gpu. In *Proc., EG Par. Graphics and Vis. '06* (2006), pp. 67–73.

[GM96] GOEL V., MUKHERJEE A.: An optimal parallel algorithm for volume ray casting. *The Visual Computer 12* (1996), 26–39.

[GW06] GEORGII J., WESTERMANN R.: A generic and scalable pipeline for gpu tetrahedral grid rendering. *IEEE Trans. Vis. and Comp. Graphics 12*, 5 (2006), 1345–1352.

[KEI07] KELES H., ES A., ISLER V.: Acceleration of direct volume rendering with programmable graphics hardware. *The Visual Computer 23*, 1 (2007), 15–24.

[Kni00] KNITTEL G.: High-speed software raycasting on a dual-cpu using cache optimizations, mmx, streaming simd extensions, multi-threading and directx. In *Proc., SPIE Visual Data Exp. and Anal. VII* (2000), pp. 164–174.

[KSSE05] KLEIN T., STRENGERT M., STEGMAIER S., ERTL T.: Exploiting frame-to-frame coherence for accelerating high-quality volume raycasting on graphics hardware. In *Proc., Vis. '05* (2005), pp. 223–230.

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proc., Vis. '03* (2003), pp. 287–292.

[Lac95] LACROUTE P.: *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. Doctoral Dissertation (Tech. Rep. CSL-95-678), Stanford Univ., 1995.

[LCN98] LICHTENBELT B., CRANE R., NAQVI S.: *Introduction to Volume Rendering*. Prentice Hall, U. Saddle Riv., NJ, 1998.

[Lev88] LEVOY M.: Display of surfaces from volume data. *IEEE Comp. Graphics and Apps. 5*, 3 (1988), 29–37.

[LN07] LEE J. K., NEWMAN T.: New method for opacity correction in oversampled volume ray casting. *J. WSCG 15* (2007), 1–8.

[MHB*00] MEISSNER M., HUANG J., BARTZ D., MUELLER K., CRAWFIS R.: A practical evaluation of popular volume rendering algorithms. In *Proc., IEEE Symp. on Vol. Visualization* (2000), pp. 81–90.

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proc., EG Par. Graphics and Vis. '06* (2006), pp. 59–66.

[Pfi05] PFISTER H.: Hardware-accelerated volume rendering. In *The Visualization Handbook* (C. Hansen and C. Johnson (eds.), Elsevier, New York, 2005), pp. 229–258.

[RGW*03] ROETTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *Proc., IEEE Symp. on Data Vis. '03* (2003), pp. 231–238.

[RPSC99] RAY H., PFISTER H., SILVER D., COOK T.: Ray casting architectures for volume visualization. *IEEE Trans. Vis. and Comp. Graphics 5*, 3 (1999), 210–223.

[SKLE03] SCHULZE J., KRAUS M., LANG U., ERTL T.: Integrating pre-integration into the shear-warp algorithm. In *Proc., Third Int'l Work. on Vol. Graphics* (2003), pp. 109–118.

[SL02] SCHULZE J., LANG U.: The parallelization of the perspective shear-warp volume rendering algorithm. In *Proc., EG Par. Graphics and Vis. '02* (2002), pp. 61–69.

[SMW*04] STRENGERT M., MAGALLON M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical visualization and compression of large volume datasets using gpu clusters. In *Proc., EG Par. Graphics and Vis. '04* (2004), pp. 41–48.

[WBLS03] WU Y., BHATIA V., LAUER H., SEILER L.: Shear-image order ray casting volume rendering. In *Proc., 2003 Symp. on Interact. 3D Graphics* (2003), pp. 27–30.

[Wei06] WEISKOPF D.: *GPU-based interactive visualization techniques*. Springer, New York, 2006.

[Wit98] WITTENBRINK C.: Survey of parallel volume rendering algorithms. In *Proc., Par. and Dist. Proc. Techs. and Apps. (PDPTA) '98* (1998), pp. 1329–1336.

[WMG98] WITTENBRINK C., MALZBENDER T., GOSS M.: Opacity-weighted color interpolation for volume sampling. In *Proc., IEEE Symp. on Vol. Visualization* (1998), pp. 135–142.