GI-Cube: An Architecture for Volumetric Global Illumination and Rendering

Frank Dachille IX and Arie Kaufman
Center for Visual Computing (CVC)
and Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA

Abstract

The power and utility of volume rendering is increased by global illumination. We present a hardware architecture, GI-Cube, designed to accelerate volume rendering, empower volumetric global illumination, and enable a host of ray-based volumetric processing. The algorithm reorders ray processing based on a partitioning of the volume. A cache enables efficient processing of coherent rays within a hardware pipeline. We study the flexibility and performance of this new architecture using both high and low level simulations.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: Volume rendering, volumetric ray tracing, volumetric global illumination, volume processing, hardware accelerator

1 Introduction

Volume rendering is an eminently useful technique whose unequaled ability permits interior examination of amorphous, complex, and sampled volumetric datasets. Its voracious appetite for memory, bandwidth, and processing power has finally been met by advances in technology. General purpose graphics hardware now empowers interactive volume rendering [2] and special purpose, real-time volume rendering hardware has made it from the realm of research [3, 13, 17] into the hands of consumers [16]. Commercial use of real-time volume rendering coupled with useful interaction has the ability to transform visualization in the same way that polygon graphics libraries and hardware acceleration have fueled the recent explosion in graphical applications.

Volumetric global illumination [21] goes beyond the usual direct volume rendering to boost the realism of a volumetric scene. By considering not only the direct component of the illumination, but also the indirect, we permit a range of natural phenomena to occur, including shadows, reflections, radiosity, and caustics. While sometimes considered fanciful, these phenomena often enhance a visualization by adding perceptual clues [10]. For example, soft shadows cast by a complex object from an area light source permit us to use our innate ability to judge relationships via size, sharpness, and shape of cast shadows. A few carefully-placed reflective surfaces allow to naturally increase the area of visualization. Diffuse inter-reflection among various objects provides subtle, but effective perceptual clues to spatial arrangement. Coupling real scanned volumetric datasets with physically-based rendering enables highly detailed and realistic rendering.

While direct volume rendering challenges even the most advanced hardware, global illumination imposes an additional significant computational penalty. Algorithmic enhancements are the usual method devised to accelerate slow volumetric global illumination algorithms. Consider volumetric shadow casting in which shadow rays are cast from each visible sample point toward the light source(s) [21]. This multiplies the original effort unless coherence is used to simplify the computation. For example, Kajiya and von Herzen [7] processed the illumination through the volume slice-by-slice, effectively coalescing collinear shadow rays into one (cf., [1]). A restricted form of this technique is used to provide shadowing in the Heidelberg ray tracing model [12].

While volumetric shadowing has a recognizable coherence, general global illumination as a solution to the volumetric transport operator [9, 15] contains little apparent coherence. Radiosity simplifies the general nature of reflections (i.e., presumes a diffuse BRDF) to allow efficient re-use of light transport sub-paths. When efforts to re-use computation are impossible, significant savings can be had by re-organizing the computation for good cache locality. For example, Pharr et al. [18] accelerated complex global illumination calculations by reordering ray tracing computations by sub-volume, reducing the number of sub-volume cache loads, enabling computation on a small memory machine, and significantly improving the run time. Vettermann et al. [23] managed to hide the latency of volume rendering memory accesses by multi-threading multiple rays in a round robin fashion. Despite these advances, global illumination is generally non-interactive. With advanced ray tracing hardware, however, it enters the realm of possibility.

This paper describes a new hardware architecture, GI-Cube, that accelerates volumetric global illumination as well as standard direct volume rendering. The design is based on a single-chip ASIC design from which an entire system can be constructed on a single PCI board. The core ASIC is a parallel, pipelined, ray integration processor. The processor re-organizes the computation in order to extract greater coherence from the usual disorderly volume accesses, whether as part of a standard direct volume rendering or a complex volumetric global illumination rendering. With its ability to simultaneously and efficiently trace arbitrary rays through a volume it can be used to accelerate various ray-based volume graphics algorithms when used as a coprocessor (see Section 5.6).

In the next section we describe the various algorithms which this architecture supports. In Section 3 we present the details of the hardware design. Section 4 describes how we simulated the algorithms and architecture. We present our results and discussion in Section 5 and conclude the paper with Section 6.

2 Algorithms

The GI-Cube architecture can be described as a cache-conscious volume ray tracing coprocessor. As such, it has the capability to

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

HWWS 2000, Interlaken, Switzerland
© ACM 2000 1-58113-257-3/00/08 ...$5.00

accelerate a variety of algorithms, such as

- basic volume rendering with Phong shading and local illumination,
- volume rendering with global illumination including shadow casting, reflections, glossy scattering, and radiosity, and
- generalized volumetric ray tracing acceleration support for various algorithms including hyper-texture, photon maps, polygonal global illumination, tomographic reconstruction, bidirectional path tracing, volumetric textures, and BSDF evaluation.

The remainder of this section describes these algorithms in detail.

2.1 Volume Rendering

In its most basic algorithm, GI-Cube operates as a standard volume rendering accelerator, producing images comparable in quality to Cube-4 [17] and VolumePro [16] since its pipeline is loosely derived from these. Actually, GI-Cube is part of Cube-5, the next generation of the Cube architectures. The basic pipeline of GI-Cube takes volume samples along a ray using trilinear sampling of the density and gradient, performs some per-sample shading operation, and composites the shaded sample onto the ray. GI-Cube radically departs from these prior architectures by generally processing rays in image order thus permitting great flexibility (e.g., perspective projection, global illumination). However, the processing order is not strictly image order; it is more accurately a hybrid order. Groups of rays are spatially partitioned into blocks, then blocks are processed one (or several in parallel) at a time. Rays are passed between blocks (and sometimes processors) at boundaries.

At a coarse level the processing is image order (i.e., individual rays may be submitted at any time during the computation).

At a medium level the processing is object order. All the rays within a block are processed together.

At a fine level the processing is image order. Samples are taken along each ray in round robin fashion using a cache to exploit coherence among rays.

Prior to rendering, the N^3 volume data is organized into cubic blocks of size n^3 (determined by the hardware implementation: in our example N=256 and n=32). The data may be stored on disk blocked or else assembled into blocks on the fly. The blocks are distributed among p multiple processors according to some function. For the best load balance, N is related to pn by some integer. We studied three possible distributions (see Figure 1):

Simple slab Cut the volume into p axis aligned slabs and assign one slab to each processor.

Repeated slab Cut the volume into slabs of n voxels thick and assign slab i to processor i mod p.

Skewed block Assign cubic block (i,j,k) to processor $(i + j + k)$ mod p.

The first option minimizes inter-processor communication at the expense of load balancing. The second option increases inter-processor communication but improves load balance, although certain viewing directions still balance poorly (i.e., the image plane oriented parallel to the slabs). The third option, which skews blocks the same way that Cube-4 skews voxels, has the best load balance but the most inter-processor communication [11]. Rays may pass between blocks three processors away, while the first two options always communicate locally. None of these schemes is necessary in the uniprocessor case [18].

To render an image, a set of image rays is generated on the interface to the GI-Cube board and clipped with the volume extents. Each ray is a data structure containing the (u,v) image coordinate, contribution to the image, starting point, direction, color, opacity, lifetime, generation, interaction value, and type (see Figure 2). Ray creation can be made very efficient by treating the intersection coordinates as texture coordinates and utilizing an incremental texture mapping scan conversion approach. Generating the volume intersection coordinates can then be as simple as scan converting the six faces of the volume. Cut planes are implemented using one additional polygon scan conversion per plane. Rays are then bucketed into queues representing the volume blocks. Some portion of the queues is loaded onto the fixed size hardware queues.

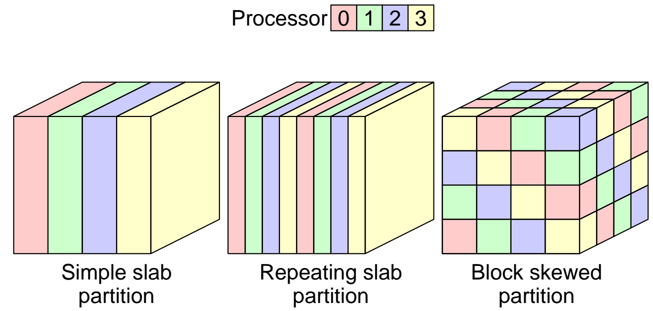

Figure 1: *Three volume partitioning strategies for multiprocessing.*

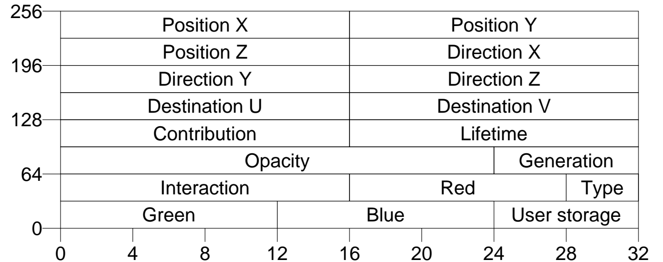

Figure 2: *Bit widths in the 32 byte ray packet.*

Each hardware processor (pipeline) selects a queue which contributes most toward completion. Two policies have been studied:

1. The simplest policy to implement in hardware is to select the queue with the most rays.
2. Also possible to implement in hardware is to select the queue with the most contribution to the image. Contribution can be measured by summing the individual contribution of the rays in the queue.

Each pipeline processes all the rays in its active queue simultaneously in a round-robin fashion (see Figure 3).

A volume cache large enough to hold an entire block is used to capitalize on spatial coherence among rays. To trilinearly sample the first ray, eight neighboring voxels need to be fetched into the cache. The direct mapped, eight-way interleaved volume cache provides the ability to read out an entire trilinear neighborhood of voxels in each cycle. The next ray, assuming it is spatially coherent, should be able to share some (perhaps four) of the fetched voxels depending on the inter-ray spacing. Assuming a ray and sampling density of one ray sample per voxel (just adequate for volume rendering) then processing the entire block generates only $(n + 1)^3$ cache misses although the total bandwidth for trilinear interpolation is eight times that. Supersampling along the ray increases coherence; the second ray sample in a trilinear neighborhood has a 100% hit ratio. While there are some stalls at the start of processing a block, the number of voxels requested from the main volume memory never exceeds the block size. Note that the volume cache

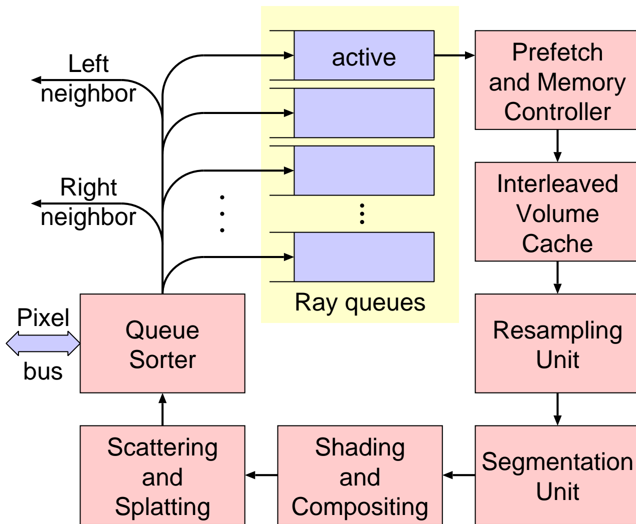

Figure 3: *An overview of a block processor showing a circular ray integration pipeline fed by multiple queues.*

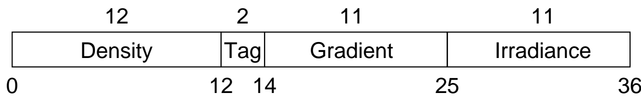

Figure 4: *The voxel format.*

actually holds one extra slice of voxels in each dimension to account for the overlap of the trilinear neighborhood.

In addition to 12 bits of density, voxels contain gradient information, a tag field for segmentation, and irradiance data which is unused for standard volume rendering (see Figure 4). The gradient is pre-computed and quantized into discrete angular bins similar to [13], although vector quantization could yield improved results for some datasets. The eight neighboring voxels are read from the volume cache, decoded, and interpolated. The density, three gradient components, and possibly three color components are trilinearly interpolated, and the tag field is zero-order interpolated (i.e., nearest neighbor).

The final sample is passed down the pipeline for shading and compositing. If no color information is supplied, a transfer function is used to convert the density into color and opacity, otherwise the transfer function only supplies the opacity. The material tag is used to provide additional dimensionality (e.g., useful coexistence of sampled and scanned datasets often requires separate tailored opacity transfer functions). The inter-sample distance d is multiplied by a random number r selected between 1/2 and 1 to provide jittering as a compromise between noise and aliasing. The opacity is modulated accordingly: $\alpha' = 1 - (1 - \alpha)^{dr}$.

Local shading is performed by utilizing a reflectance map [22]. While this technique is limited to distant light sources only, the results are adequate for most visualizations. The pipeline next performs compositing using the **over** operator [19] and stores the result back into the ray data structure which traveles down the pipeline. All the while, new rays have been accepted into the top of the pipeline resulting in the production of one new ray sample (on different rays) with each hardware cycle, neglecting any stalls due to cache misses.

Next, the ray is advanced by distance dr. The new start position determines the appropriate queue for the ray. First, the appropriate processor is selected. If the ray now belongs to a neighboring processor, the ray data structure is queued for horizontal communication to the appropriate neighbor. If the ray has exited the volume or reached full opacity, it is queued for vertical communication with the board interface. If the ray remains in the same processor, it is further scrutinized to determine the appropriate block for subsequent queuing.

As rays are received by the board interlaced, they are assembled into a composite image. Sub-pixel sampling may be used to improve the quality of the image. In this case, sub-pixel coordinates are stored in the generated rays and the composite image is accumulated by appropriate filtering. Once complete, the final image is transferred to main memory for display on the framebuffer.

Space leaping and early ray termination accelerate rendering of most datasets. Early ray termination is employed to halt processing when the ray reaches some opacity threshold. Because some blocks may contain no visible data an empty flag is associated with each block. A special part of the pipeline computes the nearest edge intersection of each ray. Based on the ray direction, axial distances to the pertinent three block faces are computed. The actual distance of intersection is computed by dividing by the appropriate component of the ray direction. If the empty flag is set the ray is advanced by the minimum of the three distances, but at least dr. Note that some blocks may be rendered invisible by the current transfer function although they contain nonzero data. It is the job of the driver software to detect this condition and set empty flags as necessary (perhaps asynchronously after editing the transfer function).

2.2 Global Illumination

To enable volumetric global illumination, a number of changes are required to the algorithm. First, rays must be generalized to be either lighting rays or rendering rays. The primary difference between them is that lighting rays distribute energy along the ray while rendering rays accumulate it (cf., [2]). In our design, energy is distributed trilinearly (splatted) to the irradiance field of voxels along the ray. Second, we must adopt a two-pass (bidirectional) methodology in which energy is distributed from the light sources in the first pass, and irradiance energy is gathered toward the eye point in the second pass. Third, we must permit scattering of rays within the volume (e.g., reflections, caustics). Next, we examine the two passes in more detail and explore its design implications.

In the first pass of global illumination, light energy is allocated among a number of rays, then traced through the scene. Instead of generating a ray from a point light source to each of the N^3 voxels we only fire rays to the face voxels of the volume, allowing these rays to automatically sample the interior voxels. Up to $6kN^2$ rays sample the surface of the volume, where k is a super-sampling factor. Borrowing from radiosity literature (e.g., [20]), we use a point-to-finite-area form factor based on the distance r to the light source to compute the energy of the light source E_l distributed among the surface voxel rays:

$$E_v = \frac{E_l \cos\theta}{k\pi r^2} \quad (1)$$

If the light source is outside the volume, rays are only generated when $\cos\theta > 0$, where θ is the angle between the inward volume surface normal and the light source direction at the finite area v corresponding to a voxel on the volume face. The k rays per voxel are distributed across the exterior voxel surface using a quasi-random scheme.

Area light sources are handled slightly differently. For each of the surface voxels, k random points on the area light source surface are selected. The energy is determined using Equation 1 multiplied by $\cos\psi$, where ψ is the angle between the inward volume surface normal direction and the light source normal at the random point on the light source. These lighting rays are queued on blocks and scheduled in the first pass just like rendering rays.

As the lighting rays travel through the volume they deposit energy to voxels along the ray. The reduction in radiance along a differential distance dx is given by $\kappa_t(x)dx$ where $\kappa_t(x)$ is the coefficient of extinction at point x and κ_t is the sum of the the coefficients of absorption κ_a and scattering κ_s. $\kappa_s(x)$ is determined by $K_s\alpha'(x)$, where K_s is a constant defined per material tag, and therefore $\kappa_t(x) = (1 - K_s)\alpha'(x)$. At each sample point along the ray, the energy E_r stored in the density field of the ray data

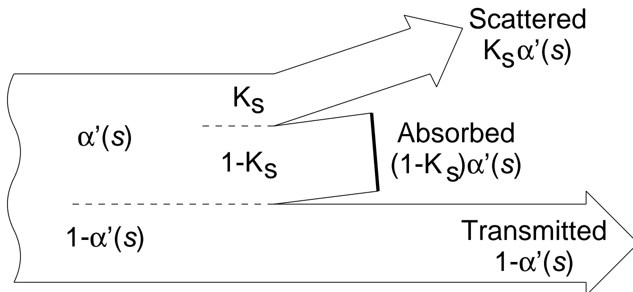

Figure 5: *The splitting of energy at a volume sample s.*

structure is split into scattered energy $E_s = E_r K_s \alpha'(x)$, absorbed energy $E_a = E_r(1 - K_s)\alpha'(x)$, and the transmitted final ray energy $E'_r = E_r(1 - \alpha'(x))$ (see Figure 5). Part of the ray energy is eventually stored in the volume data as a view independent estimate of the irradiance [14].

Two modes of global illumination are available, low albedo and high albedo. In low albedo mode, optically thin objects (e.g., light fog) are lighted with a small number of light rays (actually bundles of rays), while in high albedo mode many rays are used to stochastically sample the complex transport space of high albedo objects. In low albedo mode, light bundles distribute part of their energy at all samples with nonzero opacity, similar to the *absorption suppression* method of Pattanaik et al. [14]. In this mode, the exiting bundle energy is E'_r and the absorbed energy E_a is trilinearly distributed (splatted) to the irradiance fields of the eight neighboring voxels. The ray direction is only scattered when the accumulated ray opacity computed incrementally with $\alpha_{r,i+1} = 1-(1-\alpha_{r,i})(1-\alpha'(x))$ reaches some uniform random value which is constant per ray and is stored in the "interaction" field of the ray format. In this way, the light bundle energy is continuously distributed to the voxels along the way.

In high albedo mode, scattering is an important source of illumination so many light rays are necessary to sufficiently sample the illumination transport equations. In this mode, rays model photons which carry an indivisible amount of energy which may scatter (bounce) at many difference locations before finally being absorbed. As such, they only deliver their energy once during a photon capture event (absorption). As in low albedo mode, rays interact with the medium when the accumulated opacity reaches the interaction value. However, rays either scatter or absorb depending on the scattering albedo κ_s/κ_t and a uniform random value. If scattering is selected, the ray direction is modified based on the material's BRDF and the accumulated opacity is set to zero. The ray then continues in a new direction having gained a generation. Whenever the photon is absorbed, the full energy E_r is splatted to the irradiance fields of the neighboring eight voxels. The two methods attempt to achieve the same distribution of energy in two different ways which are tuned to the characteristics of the medium. With a large number of photons, the law of large numbers succeeds in generating a continuous distribution of illumination. The high albedo mode can also be used for shooting a large number of rendering rays. In this case, color is not accumulated continuously along the ray, but only when the ray is absorbed.

Both lighting rays and rendering rays scatter according to the phase function or bidirectional scattering function (BSDF) of the material. The tag of each voxel determines the material type. Among other things, materials define the color, the scattering constant, and the BSDF. The specification of the BSDF is flexible. Certain BSDFs are easily specified in hardware (e.g., isotropic scattering) and others (e.g., Fresnel, refraction, point sampled automobile paint) are more suitable for software implementation. Complex BSDFs are trapped by the hardware and redirected to software for further processing. The hardware supports specular reflection, dull reflection, dull scattering, isotropic scattering, and ideal diffuse reflection. All other BSDFs intercept the ray processing in hardware and pass it to the software for processing. After the software scatters the ray, it re-queues the ray on the appropriate hardware queue for further processing. Note that both lighting and rendering rays can be scattered.

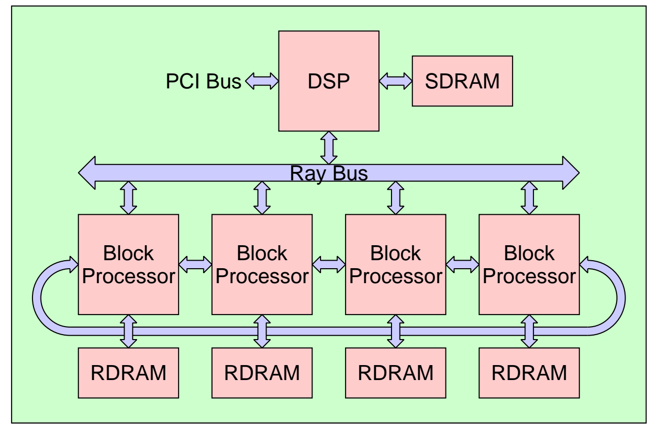

Figure 6: *An overview of the GI-Cube design.*

Because of the apparent randomness introduced by perspective projection, the global illumination lighting method, and scattering, rays no longer travel in coherent groups as in orthographic volume rendering. The block reordering and scheduling algorithm automatically attempts to maximize coherence among computations [18].

3 Hardware

The architecture is designed to render 256^3 volumes in real time (30 Hz) using a single PCI board (see Figure 6). Global illumination and other algorithms are accelerated to interactive rates, depending on the desired quality of results. The flexibility of the generalized ray tracing approach and the pipelined hardware makes this all possible.

3.1 DSP

The board is composed of three major components. The first component is a digital signal processor (DSP) which acts as an interface to the board and a controller of the individual processors. It handles loading the dataset, generating lighting and viewing rays, controlling the operation and I/O of the processors, assembling the final image, and reporting the results back to the host over the PCI interface. It is directly connected to an SDRAM for scratch memory storage and the framebuffer.

3.2 Processors

The second component is a set of hardware pipelines called processors; each follows the general layout of Figure 3. Each pipeline is designed to take one sample along a ray in each hardware cycle, barring stalls due to cache misses.

The first responsibility of the processors is to maintain and sort a group of fixed size hardware queues of rays. As new rays are accepted from either the DSP or neighbor processors, they must be bucketed into queues depending on the starting point of the ray. Simple addition, shift, and modulo operations are sufficient to select the appropriate queue. Given a ray start position (x, y, z), block size of 32, volume size of 256, and the simple slab volume distribution scheme of Figure 1, the internal queue number q is determined

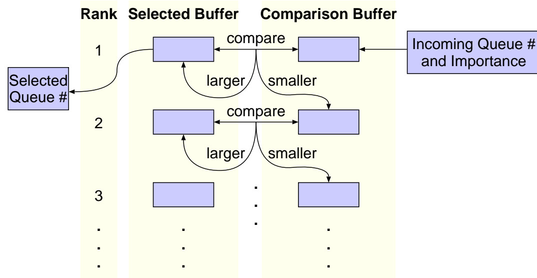

Figure 7: *A pipelined insertion sorter selects the most important queue.*

using C shift notation by:

$$x' = (x >> 5) \mod 2$$
$$y' = y >> 5$$
$$z' = z >> 5$$
$$q = (x' << 6) + (y' << 3) + z$$

Due to the simplicity of queue selection and the potential bottleneck, the bucketing operation is over-clocked by a factor of two.

Embedded DRAM (eDRAM) is used to maintain ray data in the queues since the data access is highly regular, the amount of storage is large, and the required bandwidth is high. A fixed number and size of ray queues simplifies the bookkeeping allowing constant time access to active and sorted queues in every cycle. For our reference design with 256^3 voxels, blocks of 32^3, and four processors, each processor has 128 queues. With each queue having a length of 256 rays and width of 32 bytes, the total eDRAM memory on a four processor ASIC is 4MB. Because the sorting operation is over-clocked, up to two rays need to be written into queues at the same time. For that reason, each queue is implemented as a separate eDRAM so multiple queues may be written simultaneously, unless of course both rays belong in the same queue. At the same time, the active ray queue supplies new rays to the pipeline at the rate of one ray per cycle.

One issue which arises with fixed sized ray queues is exhausting available space. This can happen when rays converge at a point (near a light source, near the camera, or with focused caustics). When this occurs, overflow rays are returned to the DSP over the ray bus. This does not generally hamper throughput since overflow occur when there is *too* much work in the ray queue. The only detriment is the consumption of resources on the DSP.

Another responsibility of the processors is to determine the active ray queue. Each ray queue is assigned a scalar importance based on one of the criteria discussed in Section 2.1. Each of these criteria can be incrementally maintained by simply adding any incoming ray importance and subtracting any outgoing ray importance. To sort these importances in hardware, we utilize a novel pipelined insertion sort (see Figure 7). A priority heap would suffice, but be more complex to design.

This hardware inserts each modified importance at the top and allows it to filter down through the ranks incrementally. When a modified queue appears at the top, it immediately wipes out its old location in the ranks. That way, only one copy ever appears in the list. Each rank contains two items: the *selected* and the *comparison*. Items from above are always first placed in the *comparison* buffer. Then at each cycle, the two buffers are compared and the larger is moved to the *selected* buffer. The smaller is moved to the *comparison* buffer of the next lower rank. The active queue is always processed until it is emptied, so it must remain at the top rank until it becomes zero. Therefore, the importance of the active queue is always infinity. When the active queue empties, all lower ranks are simultaneously moved up one rank. The queue number of the active queue is used to control a multiplexer which selects the active queue output.

3.3 Memory

The third major component of our design is the memory. The memory is generally the key component of any volume rendering architecture since that is the usual bottleneck. In our design, we rely on parallel, distributed memory (cf., [17]) and high bandwidth memories. Parallel, distributed memory permits size and bandwidth scalability by the simple addition of identical components. The disadvantage of distributed memory is the difficulty of sharing data for dynamic load balancing, which is not attempted in this architecture. We utilize RAMBUS memory (RDRAM) because of its high sustained bandwidth for random accesses. Depending on market conditions, we could equally well use double data rate (DDR) DRAMs. At 800 MHz, one RDRAM can supply 1.6 GB/s bandwidth. In our system, standard volume rendering at our design point requires an average of 2.8 GB/s and global illumination 4.6 GB/s.

A cost effective implementation could share one RDRAM between every two processors with a noticeable degradation in performance. Such sharing increases the latency of cache misses, but increases utilization and helps avoid internal bank contention on the RDRAM. (RDRAMS require a unique bank address in every four consecutive accesses.)

Because RDRAMs currently operate at 800 MHz and supply 18 bits (1/2 a voxel) every cycle, they can deliver four voxels to the pipeline, which conservatively runs at about 100 MHz (cf., [16]). To hide latency, voxels are prefetched earlier in the pipeline (see Figure 8). After address decoding, the eight tags associated with the trilinear neighborhood are checked. Up to eight misses are queued by the miss scheduler for serial RDRAM access. The Rambus ASIC cell (RAC) streams voxel read/write operations to and from the RDRAM. Retrieved voxels are received into a queue which feeds the volume cache.

Each of the eight banks is able to read or write one voxel per clock cycle. Up to four voxels are retrieved each cycle from the over-clocked RDRAM. As long as there are no bank contentions, all are written to their respective banks during the cycle. But, to perform the trilinear interpolation using the newly retrieved voxels, they are routed around the cache through a bypass that also writes them directly to the resampling unit. To ensure that all the voxels have reached the cache, a simple counter is used that stalls each ray until the specified number of voxels have been retrieved into the cache.

Before interpolation, the gradient index of each of the eight voxels must be decoded into three components. The decision was made to cache the 11-bit gradient index and decode eight indices in parallel using eight identical gradient look-up tables (LUTs). It would cost over 50% more storage to cache the actual gradient components (3×10 bits per index).

3.4 Pipeline

The fourth major component of the processors is the pipeline itself (see Figure 3). The resampling unit can be considered the top of the pipeline. It accepts one sample location along a ray and the eight nearest voxels. It trilinearly interpolates the density, gradient, and irradiance using seven linear interpolators per channel. The tag is nearest-neighbor sampled. The sample data is passed to the segmentation unit which looks up the color, opacity, and shading coefficients (total of 36+16+32=84 bits) in a density+tag indexed SRAM LUT.

The segmented and classified sample is passed to the spacing unit which randomly jitters the sample location along the ray. The sample location and image destination bits are mangled to select a random factor from an SRAM LUT representing uniformly random numbers between 1/2 and 1. The sample opacity is then modulated, using another LUT to assist in computing the power function. Then, the opacity of the ray is updated by using the compositing equation.

If the ray is a lighting ray, the amount of energy lost to the surrounding voxels is computed as described in Section 2.2. If any energy should be deposited, the amount is forwarded to the miss scheduler of the volume cache in order to be added to the nearest eight voxels using the weights interpolated in the resampling unit.

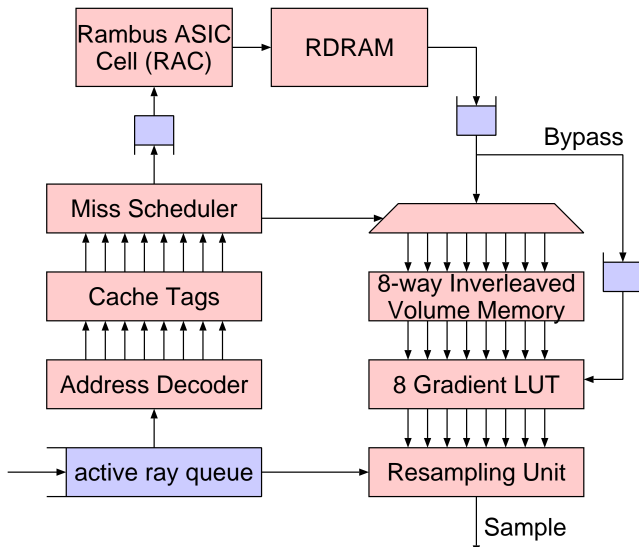

Figure 8: *The volume memory cache design including the prefetch architecture, RDRAM interface, and 8-way interleaved volume memory.*

This reduces the available bandwidth and can lead to stalls, especially in low albedo mode. The only complication is when a sample borders other blocks: because the voxels on one edge are shared, up to seven other blocks may contain copies of the voxels. To maintain consistency among different copies, the energy is packaged up and queued on the neighboring blocks.

A special flag in the ray is used to mark them as irradiance carriers. When the volume cache detects an irradiance carrier, it retrieves the current irradiance for each voxel from the cache, adds to them the trilinearly interpolated energy, and writes them back to the cache and memory. The main additions required to accommodate this read/modify/write behavior are a datapath from the pipeline to the volume cache, a datapath from the pipeline to the dispatcher, and a recirculation from the data cache to the miss scheduler to write the voxels.

If the ray is a rendering ray and global illumination is turned off, the sample is shaded using the reflectance map. The resolution of the reflectance map is 128^2 for each of the six faces, but discretization artifacts are minimized by using bilinear interpolation. For globally illuminated rendering rays, illumination is computed based on the BSDF. The diffuse component is estimated by using the sampled irradiance. The specular component is resolved only if the ray is scattered and happens to hit the light source, which is only feasible for area light sources. To evaluate the specular component of the Phong lighting model would require casting a ray to each light source, multiplying the number of rays and flooding the fixed length ray queues. Alternatively, the reflectance map can be loaded with the specular component of the point light sources (rather than the usual black) and the result summed into the sample color.

The final stage of the pipeline scatters the ray based on the BSDF. If the BSDF of the sample's material is too complex for the hardware to compute, a flag is set in the ray data structure and the ray is directed to the ray sorter to be forwarded to the DSP for further processing. In the DSP, a user-specified routine computes the new ray direction and the ray is sent back to the processors via the bus. Simple BSDF's are evaluated in hardware. For example, isotropic scattering is computed by selecting a direction δ from a random direction LUT. The random index is selected by again mangling the ray position and destination bits. Glossy scattering is performed using:

$$D' = \frac{D + \beta\delta}{\mid D + \beta\delta \mid}$$

where D is the original direction and β some fraction controlling the glossiness. Glossy reflections are simulated by reflecting the original ray direction about the sample normal followed by glossy scattering. Careful selection of β permits simulation of perfectly sharp to diffuse materials.

3.5 Board Design

The layout of the board is much simplified by grouping all the processors into a single ASIC. The rest of the board is composed of a DSP, some SDRAM for the framebuffer, and two RDRAMs for volume memory. Of course, additional RDRAMs can be easily daisy chained to meet increased memory needs. With this chip count, a single PCI board implementation is entirely feasible. The primary difficulty is most likely designing to avoid heat buildup.

The host is freed by the board of most rendering responsibilities. Besides modeling, manipulation, and final display, the host is largely available for other processing. The exception is when other algorithms (see Section 5.6) are performed which rely on the host for most of the algorithmic work, while the board acts as a coprocessor.

The DSP carries the bulk of the rendering management responsibilities. Prior to rendering, the DSP coordinates loading the volume data and LUT onto the ASIC. During rendering, the DSP continually generates new rays according to the viewing parameters and assembles the processed rays into an image in the attached SDRAM. At the end of rendering a frame, it transmits the image back to the host's main memory for display. The processors require next to no control coordination during the actual rendering; they autonomously process and return rays.

Because most of the ASIC pipeline is is similar to other volume rendering pipelines, it appears to be reasonable to implement in silicon. The primary difference with this design is the eDRAM reordering queues. Based on eDRAM manufacturers' guidelines, 32Mb of eDRAM is not too aggressive and presents no stumbling block.

3.6 Major datapaths

The bandwidth of the PCI interface (132 MB/s) to the board can become a bottleneck in some cases. For usual volume rendering and global illumination at 30 Hz, it is possible to transmit 512^2 images at 36 bits per pixel (8-8-8-12 rgbα) without saturating the bus. However, other algorithms (e.g., volumetric texturing) which require the full participation of the host may flood the interface. If all the rays are generated on the host, transferred over the PCI bus, and returned over the bus, flooding may occur. In this mode, the frame rate or resolution might be reduced, unless the AGP interface which provides four times the bandwidth is used instead.

The pipelines are controlled through simple logic connections. There is no need to coordinate shared activity among the processors over the bus. Therefore, the ray bus can be physically implemented as two unidirectional connections. The DSP to the processors is just a broadcast connection, while the processors to the DSP is a many-to-one connection. The latter can be implemented as a binary tree of merging nodes.

The size of a ray packet is 32 bytes. The width of the bus can accommodate this size since it is on-chip. The bus frequency matches the processor frequency (100 MHz) so each processor receives one new ray every p cycles. As p grows beyond four, this can become a limitation, particularly during the start-up period when the queues are first filled. To overcome this, ray differentials can be used to compress a sequence of rays into a smaller data structure. Just as the host instructs the DSP to generate all the rays for a specific viewpoint, the DSP can instruct the processors to extrapolate a number of rays given the differential (e.g., in direction and destination).

The communication between processors is also in the form of ray packets of 32 bytes. Since this communication is local and presumably on-chip, it can be easily implemented. Testing with the simulator has shown that processors communicate with each other only 2 to 4 out of 100 cycles. Therefore, each ray packet can be

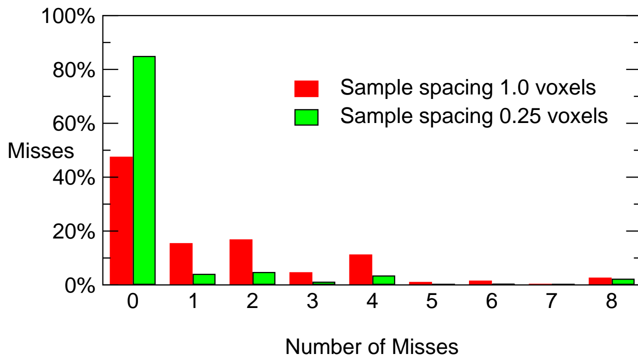

Figure 9: *Histogram of the number of misses incurred per sample.*

broken into several smaller parts and transmitted over a narrower bus at the same speed.

However, partitioning the volume with repeating slabs increases the communication to about 7% and block skewing about 20%. Block skewing resulted in a bisection bandwidth of about 12 MB/s, while simple slabs had about 1 MB/s or less.

Tests show that the memory interface is used 37.5% of the time during rendering and 92% of the time during the lighting pass of global illumination. The increased utilization is due to the irradiance splatting operation which requires a read-modify-write cycle.

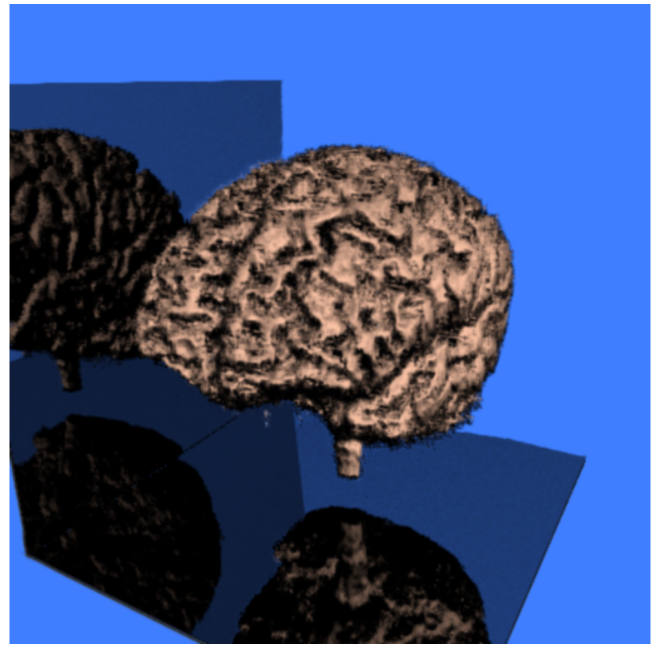
Figure 10: *Reflections naturally increase the area of visualization.*

4 Simulation

4.1 Algorithmic Simulator

Two software simulations were written in C++. The first implemented the algorithms at a high level. While it was not written for speed, it was multi-threaded to take advantage of a parallel processing machine with one thread per processor, one thread for the DSP and one thread for the host. Two-pass global illumination algorithms tended to achieve near linear speedup up to two or three simulated processors, but almost no speedup beyond that. This is likely due to bus saturation because of the eight node symmetric multiprocessor topology which does not match the ring topology of the architecture.

An API was created to assist in generating scenes and controlling the operation of the simulator. The API allows both programmatic creation of scenes and the ability to read a scenegraph. The scenegraph, related to [8], permits the creation of dynamic volumetric scenes with hierarchical modeling, flexible parameter control through engines, and voxelized geometry.

The algorithms were tested with a variety of scenes. The majority of the renderings were performed at the design point of N=256, n=32, p=4, queue size 256, pipeline frequency 100 MHz, and four 800 MHz RDRAMs. Images sizes were 256^2 unless noted. Scenes were composed using the API. Gradients were computed using a Sobel filter for scanned datasets and the analytical normal for voxelized geometry.

Volume rendering with reflections was tested on a segmented brain dataset with mirrors (see Figure 10). This 300^2 image was computed in 22.7ms leading to a frame rate of over 44 Hz. Caustics were tested on a golden ring scene in which light reflected from the reflective ring focused on the floor (see Figure 11). Global illumination was further tested using a scene containing a buckyball on a mirrored floor with shadow casting (see Figure 12). The high resolution illumination pass required 361ms, but subsequent images were rendered at 39 Hz.

4.2 Hardware Simulator

The second simulator implemented a bit and cycle accurate simulation. This uniprocessor version took much longer to run, but produced exact statistics for every part of the design. The volume cache logic, all queues, and registers were all simulated faithfully. However, the latency of the pipeline was not modeled exactly but only approximated with a queue, because a gate level design was not appropriate for all the mathematical functions. All datapaths were bit restricted to the given widths. For example, dataflow of color components was restricted to 12 bits, opacities to 24 bits, and irradiances to 11 bits. The ray data format was wide enough to accommodate extra precision including space for user data, but the voxel format was compressed to fit into 36 bits.

The amount of energy carried by a lighting ray depends on the total number of rays shot into the scene. With approximately eight rays shot per voxel, rays carry 8 bits of energy so that a single voxel's irradiance could accumulate for eight impinging rays before saturation would occur with the 11 available bits. In high albedo mode in which hundreds or thousands of rays could be shot per visible voxel, rays only carry 4 bits worth of energy, but each rays energy is only delivered to a single 8-voxel neighborhood rather than distributed along the ray.

5 Results and Discussion

5.1 Speed

Figure 13 demonstrates the effectiveness of the cache with varying parameters. Obviously, the larger the size of the cache (the block size), the greater the cache hit ratio. But the cost of a larger cache is a larger and more expensive chip. Even the largest cache blocks do not achieve 100% hit ratio because of unavoidable compulsory misses. For a given cache block size, the hit ratio increased with increasing ray queue sizes. This is because larger ray queues permit a block to be re-used for a greater number of rays. The efficiencies jump higher beyond ray queue size 16 because of the latency of the pipeline. With too small a ray queue the queue empties before rays recirculate out of the pipeline and a new queue must be selected. For any given ray queue size, a block size of 32 (our design point) strikes a good balance between cache size (cost) and efficiency. Moving to a block size of 64 increases the SRAM size by a factor of eight with only a small percentage improvement in efficiency.

Figure 14 plots the number of hardware cycles required to render a typical image with varying parameters. With a block size the

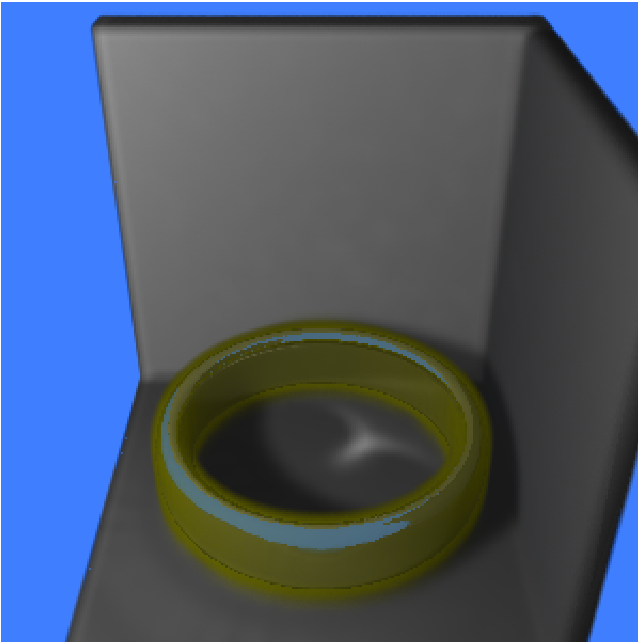

Figure 11: *Simulated caustics via reflection of lighting rays.*

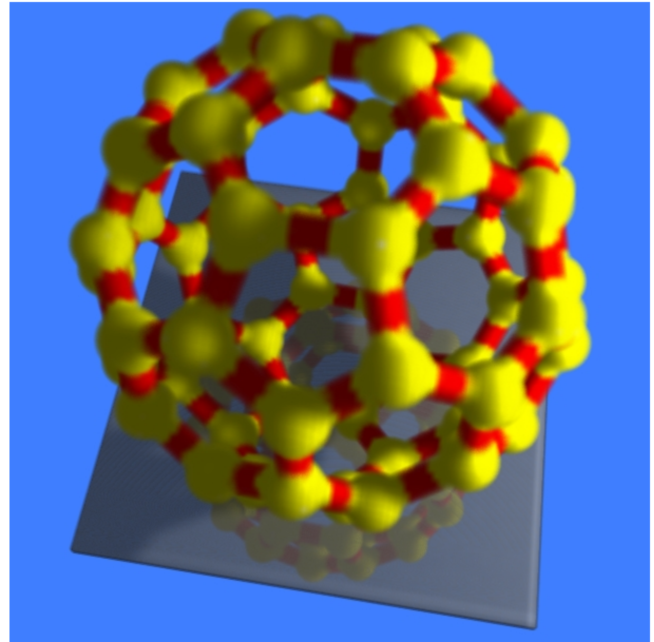

Figure 12: *Shadows, reflections, and caustics demonstrated with a volumetric buckyball model (C_{60} molecule).*

same as the volume size, queue size makes no difference in rendering time. Interestingly, smaller block sizes render more quickly for reasonable queue sizes. This helps to show that reordering is beneficial. As with efficiency, the ray queue size must be above the latency of the pipeline to realize the most benefit. Examining the values for our design point, 256 ray queue length, we see that a block size of 32 is the minimum. In general, we found that an 8×8×8 grid of blocks is the most efficient reordering granularity for a wide variety of scenes, image sizes, and datasets. This is borne out by the graph.

If VolumePro ran at 100 MHz, the design speed studied in this paper, it could process approximately 301M samples/s [16]. This design averaged 122M samples/s during lighting and 231M samples/s during rendering. Even with some stalls due to cache misses, this design was able to outperform pure object-order volume rendering because it could often take advantage of space leaping and early ray termination.

To evaluate the effectiveness of the architecture, we made a comparison of volumetric global illumination between the reordered algorithm and simple image-based ray tracing with caching. The alternative processor utilized a single queue and a direct mapped cache. Each voxel had its own cache tag for fine grained caching. We located the ideal queue size for a scanline image based renderer and gathered statistics. The simple cache processor had a poor throughput of only 52%, memory bandwidth of 1.9 GB/s, and a frame rate of 9.2 Hz. Using the same size cache, multiple queues, and intelligent reordering we achieved a throughput of 93%, a reduced memory bandwidth of 0.8 GB/s, and a frame rate of 16.3 Hz. Thus, reordering benefits by reducing the memory bandwidth and increasing the frame rate.

Figure 15 shows the use of the cache during global illumination. Ray scattering (e.g., reflections) cause blocks to be loaded more than once. The reordering algorithm helps to minimize the block loads, as it did in Pharr et al. [18]. The difference in our system is that block loads are not necessarily expensive because of lazy filling. In fact, we have only shown the first 99.6% of the computation which is dominated by large ray queues. The final 0.4% of computation exhibits rapid cache invalidation due to very short ray queues which do match the latency of the pipeline. In this condition, a small number of rays enter the pipeline but do not recirculate before another set of rays from another queue enter the top of the pipeline. Thrashing of the cache occurs because only one sample is taken per ray in round-robin style. Fortunately, this occurs for an insignificant percentage of the time and hardly affects the performance. This can potentially be avoided by increasing the associativity of the cache, decreasing the pipeline latency, or artificially increasing the queue size.

5.2 Scalability

The architecture demonstrates near linear speedup with the addition of processors. Sixteen processors achieve a speedup within 14% of ideal rendering a 256^2 image (see Figure 16). The sub-linear performance is partially related to contention on the ray bus and also the pixel-to-voxel ratio. This is explained below in Section 5.3.

There are various ways to measure scalability. This particular test measured the speedup of a fixed size problem with additional processors. Another test examined the ability to tackle larger problems with additional identical processors, in which case the overall throughput of each processor should ideally remain constant. In actuality, the throughput per processor varied linearly from 99.9% throughput with a single processor to 80.7% throughput with eight processors. However, with a 512^3 volume and only a 256^2 image, 16 processors averaged a paltry 29.1% throughput. By increasing the image size to 512^2 achieved an improved throughput of 76.1%. The cause of this is described next.

5.3 Efficiency

The relative sizes of voxels and pixels (thus rays) has a significant effect on the efficiency of processing. Cache, and thus, overall efficiency improves with increasing sampling density. Sampling density increases with super-sampling along the ray (see Figure 9) and with increasing image size (see Figure 17). The greater the number of rays and samples per scheduled block, the greater the potential re-use of cache. A very low pixel-to-voxel ratio means that the volume is sub-sampled by the rays and that a lower resolution volume is more appropriate for both sampling and efficiency.

Another question is how to support rendering larger datasets with fixed hardware. Standard volume rendering can just be performed in separate blocks and composited in screen space. However, global illumination requires global information and non-uniform data access. The solution is to break up the full dataset into smaller blocks

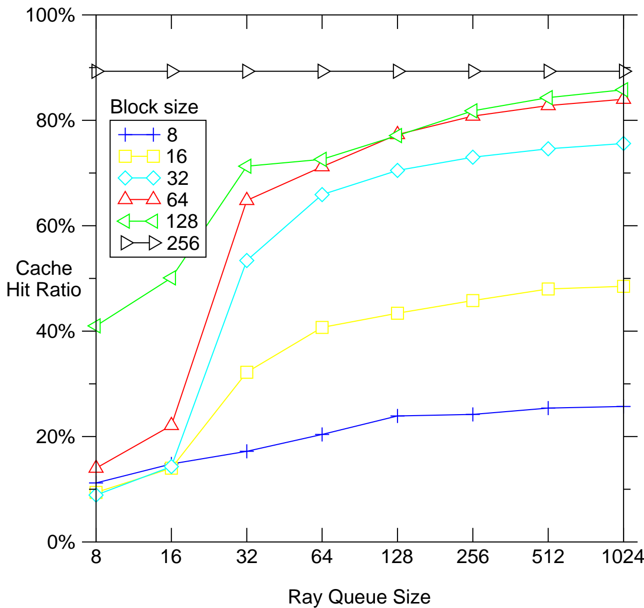

Figure 13: *Cache efficiency with varying block and queue sizes.*

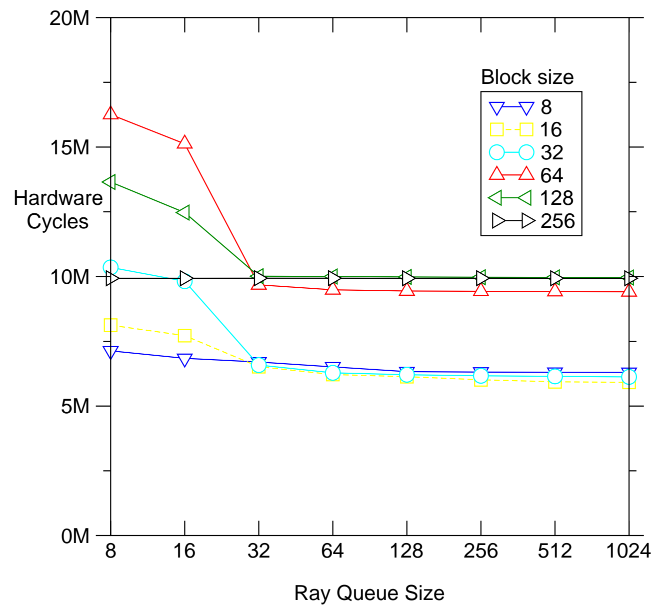

Figure 14: *The hardware time required to render with varying block and queue sizes.*

which fit into the hardware and utilize intelligent scheduling at a higher level, in effect a hierarchical scheduling algorithm with a hierarchical importance queue.

5.4 Load Balance

Because of the static memory distribution, no dynamic load balancing is possible. The first partitioning strategy, equal-sized slabs, performs adequately for usual scenes when viewed perpendicular to the slabs. However, viewing parallel to the slabs tends to take twice as long to render because each ray pierces all the slabs, imposing a temporal dependence. The second partitioning strategy has largely the same results, although there is less data dependent load imbalance due to the finer partitions. The third strategy, block skewing, evenly distributes the volume data, but adds a 19% premium on the average rendering time due to higher communication costs. Using our test scenes, we found that the simple slab partition resulted in a load balance of 91%, repeated slab partition 92%, and block skewed partition 76%. The load balance values were computed by normalizing the time each processor was busy over the lifetime of the computation. Although the block skewed partition had the most uniform nonzero voxel distribution, its load balance value was adversely affected by the high communication among processors which led to stalling of certain pipelines and overall greater time.

Performance can be increased in a number of ways. For simple ray casting (the primary rays), some temporal dependencies exist which can hamper efficiency. At the cost of complexity and some bandwidth, all segments of a ray could be cast in parallel and compositing segments later in software. This enhancement was not tested and requires future work.

Space leaping was incorporated and resulted in a <min,avg,max> performance increase of <0%,21%,34%>. Some scenes achieved no speedup because there were no empty blocks. Such is often the case in scientific visualization. Early ray termination resulted in a performance increase of <0%,12%,24%>. Again, some datasets never reach full opacity (e.g., galaxy formation simulations) and can not take advatage of this optimization. Nevertheless, these techniques are designed into the hardware and are essentially free.

5.5 Implementation

This hardware implementation is feasible, based on a direct comparison with the VolumePro implementation of 1999: 3.2M transistors and 2Mb SRAM in 0.35μ process silicon [16]. Rough estimates of our designs transistor count and memory sizes with four processors on one ASIC are presented in Table 1. By moving to current 0.20μ eDRAM technology, a chip of similar die size and cost could be produced with 9.5M transistors, 2Mb SRAM, and 32Mb eDRAM in less than one year from the date of publication, and sooner with a more expensive chip.

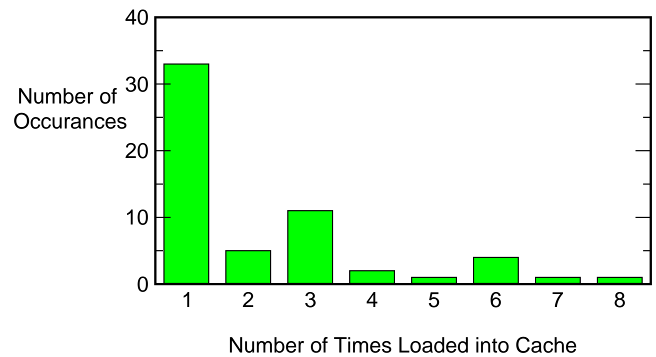

Figure 15: *Histogram of block loads during global illumination.*

5.6 Future Work

Our future work includes VHDL simulation, prototyping, and feature enhancements. Additional features considered are supporting pre-classified data (e.g., visible human RGBα data) either through overloading the voxel format and/or by expanding the bit width, intermixing surface-based geometry, and intermixing multiple volumes. An expanded voxel format could include, for example, tristimulus irradiance for colored shadows and precise color bleeding.

We also plan to explore various uses for a generalized volumetric ray integration accelerator such as GI-Cube. For instance, the most intensive portion of hierarchical volumetric radiosity requires computing the form factor between pairs of voxels. GI-Cube can be used to compute the visibility function between two voxels by creating a ray between them, and later recovering the opacity when the ray exits the hardware. Similarly, a ray can be cast for each pixel which intersects a volumetric texture. More advanced volu-

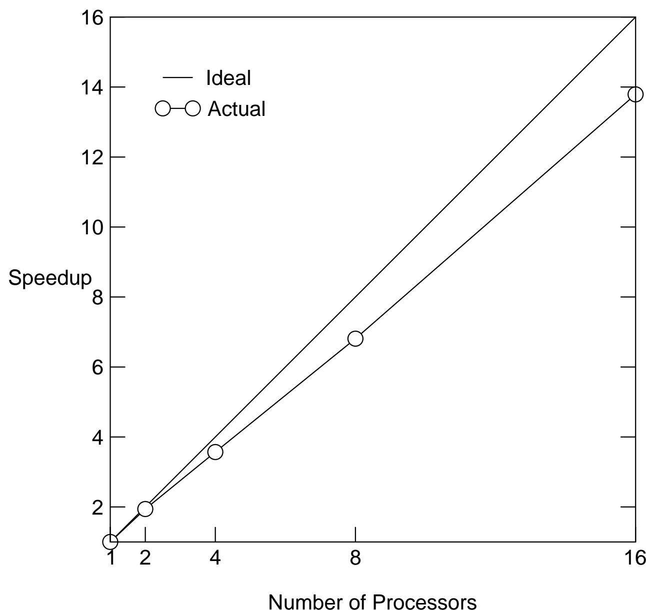

Figure 16: *Result of scalability test rendering of a 256^3 volume with 32^3 blocks.*

Table 1: *Hardware cost estimates.*

Item	*Bits/Chip*	*Total*
Ray Queues	$512\times256\times256$	32Mb eDRAM
Volume Cache	$4\times33^3\times36$	4.9Mb SRAM
Gradient LUT	$4\times2^{11}\times8\times30$	1.9Mb SRAM
Registers	$32\times256\text{b}+20\times55$	10Kb SRAM
Segmentation LUT	$4\times2^{14}\times84$	5.3Mb SRAM
Reflectance Map	$4\times128^2\times6\times8$	3Mb SRAM
Total		15Mb SRAM
Queues		2M transistors
Volume Cache		2M transistors
Pipeline		4M transistors
Control, etc.		1.5M transistors
Total		9.5M transistors

metric subsurface scattering can be computed by GI-Cube to simulate skin [6] and stone [4]. We have used lighting rays to perform volumetric backprojection in medical tomographic reconstruction. Because the hardware natively supports perspective projection and backprojection, it can be used to reconstruction data from newer and safer cone-beam scanners [5].

6 Conclusion

We have presented a hardware architecture capable of flexible and scalable volumetric ray tracing. We designed it to accelerate standard volume rendering, ray tracing, and various other ray-based volume sampling algorithms. It is further designed to permit volumetric global illumination by treating light rays as another class of volume sampling primitives which have the ability to disseminate light energy rather than gathering it. The capabilities of the algorithm have been tested by producing various images and animations.

The architecture reorganizes computation for better efficiency by grouping coherent operations. The basic reordering algorithm has been shown to be a significant improvement over näive image-order rendering with straightforward caching. The reordering algorithm nearly doubled the frame rate from 9.2 to 16.3 Hz.

The architecture scales in near linear fashion for large problems.

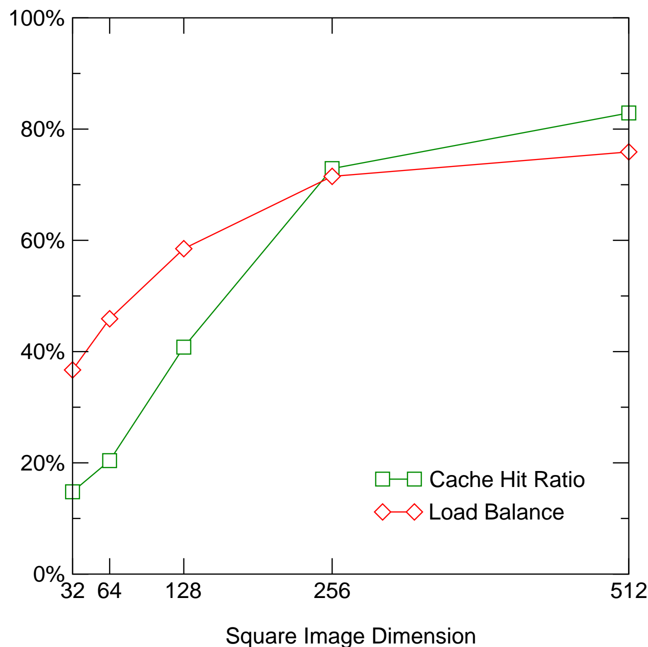

Figure 17: *Demonstration of the efficiency of rendering larger images.*

Small volumes and/or small images do not scale as well, nor do they need to scale to achieve real-time performance. The architecture successfully blends a reordering algorithm and multiprocessing to achieve reasonable scalability.

The board and chip design are reasonably similar in scale to other volume rendering architectures. Pieces of this design, the multiprocessing configuration, ray integration pipeline, volume cache, simple ray queues, can all be found in other architectures. The primary difference is the structuring of the ray queue for reordering and the generalized ray handling of both rendering and lighting rays. Our design is feasible because the parts have already been implemented elsewhere and the conglomerate algorithm has been extensively and successfully simulated.

Acknowledgments

This work has been supported by ONR grant N000149710402. The authors wish to thank Justine Dachille, Manjushree Nulkar, Klaus Mueller, Ingmar Bitter, and Kevin Kreeger for their assistance, reviews, and discussion.

References

[1] U. Behrens and R. Ratering. Adding shadows to a texture-based volume renderer. In *1998 Symposium on Volume Visualization*, pages 39–46, 1998.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, Oct. 1994.

[3] M. de Boer, A. Gröpel, J. Hesser, and R. Männer. Latency- and hazard-free volume memory architecture for direct volume rendering. In *Proceedings of the 11th Eurographics Workshop on Graphics Hardware*, pages 109–119, 1996.

[4] J. Dorsey, A. Edelman, H. W. Jensen, J. Legakis, and H. K. Pedersen. Modeling and rendering of weathered stone. *Proceedings of SIGGRAPH 1999*, pages 225–234, Aug. 1999.

[5] R. Gordon, R. Bender, and G. T. Herman. Algebraic reconstruction techniques (ART) for three-dimensional electron micropscopy and X-ray photography. *Journal of Theoretical Biology*, 29:471–481, 1970.

[6] P. Hanrahan and W. Krueger. Reflection from layered surfaces due to subsurface scattering. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 165–174, Aug. 1993.

[7] J. T. Kajiya and B. P. Von Herzen. Ray tracing volume densities. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 165–174, July 1984.

[8] A. Kaufman, F. D. IX, B. Chen, I. Bitter, K. Kreeger, N. Zhang, and Q. Tang. Real-time volume rendering. *International Journal of Imaging Systems and Technology*, 2000. to appear.

[9] W. Krueger. The application of transport theory to the visualization of 3D scalar fields. *Computers in Physics*, 5:397–406, 1991.

[10] M. Levoy, H. Fuchs, S. M. Pizer, J. Rosenman, E. L. Chaney, G. W. Sherouse, V. Interrante, and J. Kiel. Volume rendering in radiation treatment planning. In *Proceedings of the First Conference on Visualization in Biomedical Computing*, pages 4–10, May 1990.

[11] J. Lichtermann. Design of a fast voxel processor for parallel volume visualization. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware*, pages 83–92, Aug. 1995.

[12] H.-P. Meinzer, K. Meetz, D. Scheppelmann, U. Engelmann, and H. J. Baur. The heidelberg ray tracing model. *IEEE Computer Graphics and Applications*, 11(6):34–43, Nov. 1991.

[13] M. Meißner, U. Kanus, and W. Straßer. VIZARD II, a PCI-card for real-time volume rendering. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware '98*, pages 61–67, Aug. 1998.

[14] S. N. Pattanaik and S. P. Mudur. Computation of global illumination in a participating medium by Monte Carlo simulation. *The Journal of Visualization and Computer Animation*, 4(3):133–152, July–Sept. 1993.

[15] F. Pérez, X. Pueyo, and F. X. Sillion. Global illumination techniques for the simulation of participating media. In *Eurographics Rendering Workshop 1997*, pages 309–320, June 1997.

[16] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The VolumePro real-time ray-casting system. In *Proceedings of SIGGRAPH 1999*, pages 251–260, Aug. 1999.

[17] H. Pfister and A. E. Kaufman. Cube-4 - a scalable architecture for real-time volume rendering. In *1996 Volume Visualization Symposium*, pages 47–54, Oct. 1996.

[18] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 101–108, Aug. 1997.

[19] T. Porter and T. Duff. Compositing digital images. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[20] F. X. Sillion and C. Puech. *Radiosity and Global Illumination*. Morgan Kaufmann, San Francisco, CA, 1994.

[21] L. M. Sobierajski and A. Kaufman. Volumetric ray tracing. In *1994 Symposium on Volume Visualization*, pages 11–18, Oct. 1994.

[22] J. van Scheltinga, J. Smit, and M. Bosma. Design of an on-chip reflectance map. In *Proceedings of the 10th Eurographics Workshop on Graphics Hardware '95*, pages 51–55, Aug. 1995.

[23] B. Vettermann, J. Hesser, and R. Manner. Solving the hazard problem for algorithmically optimized real-time volume rendering. In *Proceedings of the International Workshop on Volume Graphics*, pages 171–184, Mar. 1999.