Progressive CPU Volume Rendering with Sample Accumulation

W. Usher[1,2]   J. Amstutz[2]   C. Brownlee[2]   A. Knoll[1]   I. Wald[2]

[1]SCI Institute, University of Utah   [2]Intel Corp.

(a) *SAVR, 1 frame*   **(b)** *SAVR, 16 frames*   **(c)** *SAVR, 32 frames*
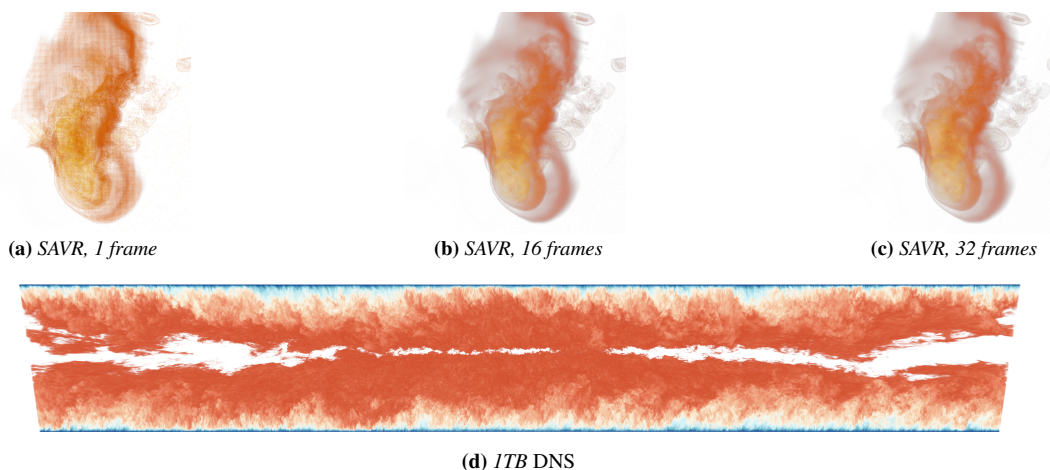

(d) *1TB* DNS

Figure 1: *(a-c) Progressive refinement with Sample-Accumulation Volume Rendering (SAVR) on the 40GB* Landing Gear *AMR dataset using a prototype AMR sampler. The SAVR algorithm correctly accumulates frames to progressively refine the image. After 16 frames of accumulation the volume is sampled at the Nyquist limit, with some small noise, by 32 frames the noise has been removed. SAVR extends to distributed data, in (d) we show the 1TB DNS dataset, a $10240 \times 7680 \times 1536$ uniform grid, rendered interactively across 64 second-generation Intel®* Xeon Phi™ *"Knights Landing" (KNL) processor nodes on* Stampede 1.5 *at a 6144×1024 resolution. While interacting, our method achieves around 5.73 FPS.*

Abstract

We present a new method for progressive volume rendering by accumulating object-space samples over successively rendered frames. Existing methods for progressive refinement either use image space methods or average pixels over frames, which can blur features or integrate incorrectly with respect to depth. Our approach stores samples along each ray, accumulates new samples each frame into a buffer, and progressively interleaves and integrates these samples. Though this process requires additional memory, it ensures interactivity and is well suited for CPU architectures with large memory and cache. This approach also extends well to distributed rendering in cluster environments. We implement this technique in Intel's open source OSPRay CPU ray tracing framework and demonstrate that it is particularly useful for rendering volumetric data with costly sampling functions.

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

1. Introduction

Over the last decade, with growing computing power, direct volume rendering has become increasingly interactive and widespread. However, large, entropic or unstructured volume data, higher-order reconstruction filters, and costly classification can still hamper performance and image quality. The typical approach to reducing the cost per frame in order to remain interactive is to decrease the sampling rate along a ray. However, since the volume rendering integral is noncommutative, old samples must be thrown away when new higher frequency samples are computed. Consequently, most progressive refinement schemes for volume rendering either recompute redundant samples prior to blending or employ separate image-space progressive rendering.

In this paper, we introduce an approach for sample-space progressive refinement in the context of direct volume rendering. In the spirit of Ohbuchi and Fuchs [OF91], we separate the *evaluation* of interpolated samples from the *integration* of samples, and implement progressive refinement by accumulating volume samples over successive frames. Each frame adds new samples to a list of samples, stored in a streaming- and vector-friendly format, and then

re-integrates this list of samples every frame. With this approach, after N rays with M samples each, we have the equivalent of a single ray with $N \times M$ samples, converging to the equivalent image as a single finely sampled frame, see Figure 2 and Figure 3. Our contribution is an analysis of this approach in practice in a modern parallel volume rendering system. Our method is independent of how samples are computed and is thus applicable to any volumetric data type that can return a sample for a 3D point in space, such as structured volumes, adaptive mesh refinement data, radial basis function particle data, unstructured volumetric data, or other representations. We show this technique can be extended to both image-parallel and data-distributed sort-last rendering on a cluster with no additional communication or impact on scalability.

One drawback of this approach—the memory required to store the samples—is feasible with modern HPC nodes and workstation memory budgets, and in fact mostly disappears in an image-parallel or data-distributed rendering context, where each node stores only the samples it needs for its portion of the image and dataset. The sample accumulation volume rendering method we propose in this paper:

- Performs correct progressive refinement of volumetric data while allowing transfer function changes during refinement
- Is capable of rendering any sample-able volumetric data
- Extends well to distributed rendering on a cluster.

2. Related Work and Background

Direct volume rendering (DVR) is a method for directly rendering a 3D scalar field through a process of sampling, classification, and integration [DCH88, Sab88, Lev88]. In recent years, ray casting [Lev88] has become the dominant method, displacing other approaches such as slicing [CN93] and splatting [Wes90]. State-of-the-art GPU volume rendering systems now employ ray casting or ray-guided approaches [BHP15, HBJP12, CNLE09]. To improve interactivity, LOD approaches have been employed in several popular GPU volume renderers such as Voreen [MSRMH09] and Para-View [Aya15]. Generally, these approaches reduce the number of samples (or slices) by some fixed fraction when the user moves the camera, resulting in visible artifacts, and then render a higher quality version when interaction has ceased. Image-space progressive rendering of volume data was first explored by Levoy [Lev90], and more recently by Frey et al. [FSME14] for time-sensitive applications (i.e., video streaming).

Ray casting approaches have generally sought to minimize the computational costs of volume rendering through methods such as space-skipping, early ray-termination, data bricking, and instruction-level optimization of the sampling process [PPL*99, KTW*11]. Levoy presented a refinement technique that sub-sampled at the pixel level, with successive frames filling in empty pixel regions of an image [Lev90]. Multi-resolution refinement techniques employ hierarchical representations of the volume, with subsequent frames rendering finer resolution representations of the data [LH91]; however, these require preprocessing the data and additional storage. OSPRay's CPU volume ray caster [WJA*16] currently employs a combination of bricking, space-skipping, early ray-termination, and SIMD-level optimizations of the in-

terpolation routine via the open source Intel® SPMD compiler (ISPC) [PM12].

Special-case volume rendering systems have employed separated approaches similar to ours. On the GPU, Nelson et al. [NLKH12] used the OptiX ray tracing engine to traverse a high-order spectral finite element volume and separate CUDA kernels to sample and integrate. Sakamoto et al. [SKKK09] perform sort-free progressive refinement of tetrahedral element volumes by rendering points with probability equal to the opacity of the sample, avoiding the cost of storing all the samples at the cost of requiring many iterations to converge. Orthomann et al. [OKK10] used a similar two-pass approach for sampling SPH/RBF volumes on the GPU. On the Intel® Xeon® processor and Intel® Xeon Phi™ co-processor (KNC), Knoll et al. [KWN*14] employed a similar two-pass algorithm per-packet for fast rendering of RBF volume data. Storing samples along the ray and deferred integration have been previously employed on the GPU in the form of volumetric depth images [FSE13] for more flexible classification, batch rendering, and *in situ* (co-processing) visualization [FFSE14]. Our approach bears some resemblance to "deep A-buffer" approaches [Car84, CICS05] and "deep images" [LV00], and to visualization preview approaches such as explorable images [TCM10] and ParaView Cinema [OAJ*16]. There are some differences: our approach computes deep samples per packet as opposed to over the entire frame buffer, and most importantly none of these works have leveraged progressive sampling in object space. However, this technique could be useful for the applications of those works.

The specific method proposed in our work is similar to the ray cache of Ohbuchi and Fuchs [OF91] and the sample buffer introduced by Ke and Chang [KC93]. Both methods perform similar refinement to ours by storing samples along a ray; our technique extends these methods to SIMD and distributed rendering in a modern volume renderer that generalizes to different types of data with different filters. There are some differences as well: in our approach we perform a new pass of samples through the entire volume and do not store approximate or lower resolution samples. Moreover, our work seeks to show the tradeoffs of these methods in practice, and expose where they may be helpful for large-scale visualization.

2.1. Volume Rendering

In Equation (1), a Riemann sum of the volume rendering integral is shown, where N is the number of samples taken along the ray and t is the distance between each sample [KM05]. $C(x)$ and $\alpha(x)$ compute the color and opacity at distance x along the ray. The product of $(1 - \alpha(j \cdot t))$ terms attenuates samples further along the ray by the opacity of closer ones, modeling light absorption by the volume. The accuracy of the discretization is dependent on the step size t. The ideal sampling rate is at the Nyquist frequency of roughly twice a voxel, but in practice is often several times the size of a single voxel to maintain interactive framerates with large or costly data. For final renderings the sampling rate is increased to the Nyquist limit, at the cost of interactivity.

$$L = \sum_{i=0}^{N} \left(C(i \cdot t)\alpha(i \cdot t) \prod_{j=0}^{i-1} (1 - \alpha(j \cdot t)) \right) \qquad (1)$$
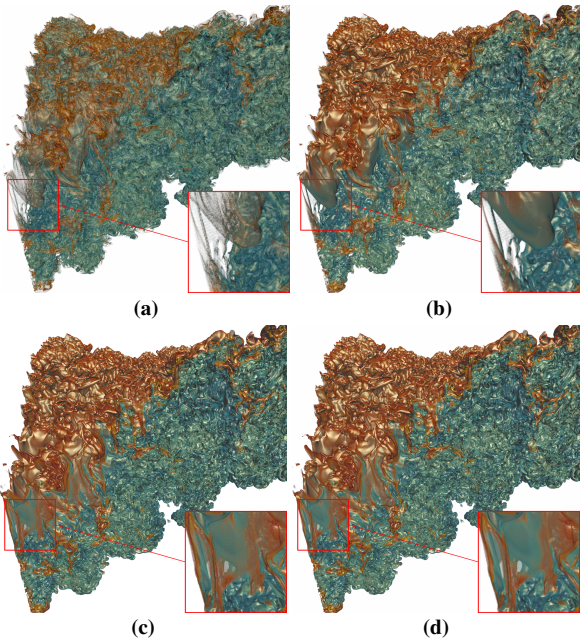
(a) **(b)** **(c)** **(d)**

Figure 2: *Convergence of DVR (a), DVR with adaptive sampling and preintegration (b), and our method (c) vs. ideal sampling (d) on the* Richtmyer–Meshkov *dataset. (a-c) Use a sampling step of 8 voxels, the adaptive sampling in (b) refines to a 0.5 step size where needed. After 16 frames (a-c) have taken an equivalent number of samples as (d), one frame of DVR with two samples per voxel. (d) Renders the image in 752ms while our method (c) takes 3.357s for the 16 frames; however, it is important to note (c) provides better interactivity during rendering at a mean framerate of 4.77 FPS, vs. 1.33 FPS for (d), on a KNL.*

To achieve smooth rendering, progressive refinement is done by averaging M frames following Equation (1), which results in Equation (2). Each frame is computed by traversing a new ray with a different starting offset through the volume, with the goal of taking a new unique set of samples with each ray. We multiply total radiance by $1/M$ to unbias opacity (note that irrespective of a correct blend order, separately-integrated samples $(1 - \alpha(j \cdot t))$ result in over-weighting of each sample per frame). Moreover, as usual, insufficient sampling of volumes with high-frequency data or transfer functions results in missing features, as shown in Figure 2.

$$L_{\text{DVR}} = \frac{1}{M} \sum^{M} \left(\sum_{i=0}^{N} \left(C(i \cdot t) \alpha(i \cdot t) \prod_{j=0}^{i-1} (1 - \alpha(j \cdot t)) \right) \right) \quad (2)$$

2.2. Distributed Rendering

Data parallel rendering in the context of large volume datasets has been widely studied, producing work covering data distribution, rendering, and compositing. Early work by Hsu and Ma et al. [Hsu93, MPHK94] distributes bricks of the volume data among multiple processes, with each process rendering an image with its assigned data. These partial images are then composited together to produce the final frame in a sort-last compositing step. Another approach employed be DeMarle et al. [DPH*03] uses image-parallel work distribution where each node is assigned a subset of
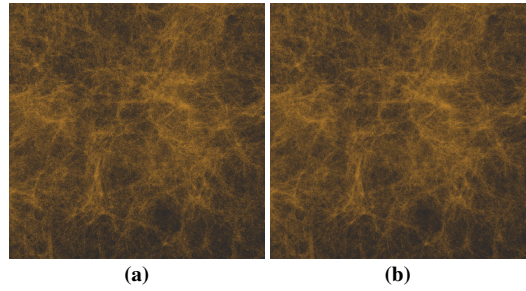

(a) **(b)**

Figure 3: *Convergence of our method (a) vs. vs. ideal sampling (b) on the* Cosmic Web *particle RBF model. (a) Uses a sampling step of 80 "voxels" to remain interactive, (b) is one frame of DVR at the ideal sampling rate of two samples per "voxel". After 160 frames of accumulation (a) has taken an equivalent number of samples as (b). Although (a) takes a total of 156.8s and (b) takes 62.76s for the single fully sampled frame, our method provides better interactivity, averaging 0.98 FPS vs. 0.0159 for (b) on a KNL.*

the framebuffer to render and creates a distributed shared memory buffer to make the volume available to all nodes. On each node the data is fetched from remote nodes as needed and cached locally into an octree for rendering.

We implement our method in OSPRay [WJA*16], which provides functionality for image-parallel rendering with duplicated data, and data-distributed volume rendering with sort-last compositing. In the image-parallel case, the scene being rendered is copied onto the nodes and each is responsible for rendering a subset of the image tiles. OSPRay's data-distributed DVR renderer is an extension of *segmented ray casting* [Hsu93] where bricks of volume data are assigned to nodes. Each node is then responsible for rendering the tiles to which its bricks project and compositing some subset of tiles.

3. SAVR – Sample-Accumulation Volume Rendering

Sample-Accumulation Volume Rendering (SAVR) decouples the *evaluation* (sampling) and *integration* of samples for the purpose of progressive refinement. With this decoupling, SAVR increases N in Equation (1) to correctly refine its approximation of the integral over successive rendered frames. As shown in Figure 2 and Figure 3 our method converges to the equivalent of the Nyquist limit sampled image while remaining interactive during rendering. However, in order to refine the integral properly, all samples taken by previous rays are retained in memory to update the approximation by recomputing Equation (1) for each frame with a finer sampling of the integral.

An overview of the algorithm is given in Algorithm 1. For each frame, a new set of samples are accumulated along each ray, starting from offsets taken from a precomputed Halton sequence (lines 7-13). These samples are merged into a buffer of previously taken samples. The buffer of all samples for the pixel, *list*, is maintained in sorted order by depth along the ray. The merge process is a variant of merge sort, which works well in practice because the sample depths from the two merged buffers are already partially sorted. To produce a color for each pixel, we color and blend all the accumulated samples by traversing the list in order and applying the transfer function.

Algorithm 1 The SAVR algorithm.

```
 1: function RENDERPIXEL(ray, volume, tfcn)
 2:     if ISVIEWCHANGED() then
 3:         list ← {}
 4:     end if
 5:     tNear ← FINDENTRY(ray, volume)
 6:     tFar ← FINDEXIT(ray, volume)
 7:     t ← tNear + GETOFFSET()
 8:     samples ← {}
 9:     while t < tFar do
10:         s ← TAKESAMPLE(volume, ray, t)
11:         APPEND(samples, s)
12:         t ← ADVANCE(ray, t)
13:     end while
14:     MERGE(list, samples)
15:     color ← BLENDALL(list, tfcn)
16:     return color
17: end function
```

We note that both evaluation and integration of samples are data-parallel problems relying on coherent ray casting of neighboring pixels. These characteristics make it much easier to map SAVR to SIMD hardware than other multi-sampled ray-based techniques such as multi-hit ray tracing, allowing us to avoid the more complicated sorting and integration methods found in multi-hit ray tracing kernels [AGGW15]. Moreover, we store all samples along the ray from entry to exit per packet of rays, distinguishing this approach from similar frame-wide methods on the GPU [NLKH12].

SAVR allows changes to the transfer fuction used for integration without the need to retake samples. In Algorithm 1, the call to `blendAll` can use different transfer functions, should the application want it to be changed. In typical DVR techniques, the progressive refinement accumulation must be restarted when a new transfer function is introduced because the refinement is performed on the final blended colors instead of the samples themselves. Our algorithm is also independent of how samples are taken in `takeSample` and can be used with more advanced sampling schemes such as adaptive sampling, or sample other volumetric data types like adaptive mesh refinement volumes or unstructured particle RBF models.

4. Implementation

The SAVR algorithm is agnostic toward hardware platform and to the choice of data distribution, tasking, and compositing algorithms. In practice, its effectiveness depends on the size and type of data, transfer function, relative costs of sampling and classification, lighting, and implementation choices. We describe the latter in this section.

4.1. Single Node SAVR

The SAVR algorithm itself is implemented using ISPC [PM12] within the OSPRay [WJA*16] framework, to take advantage of vector instructions on Intel Xeon and Intel Xeon Phi processors. Our initial implementation utilized scalar code within ISPC (serializing

with the `foreach_active` statement), and we found the vectorized ISPC version provided approximately $3\times$ speed-up over the scalar code with AVX2. We observe that volume rendering with ray casting is a relatively coherent problem, where vectorization is important for good performance. Our evaluations are with the ISPC vectorized implementation on each platform.

In OSPRay, a renderer implements a `renderSample` method that receives a packet of rays for sampling and shading. Packets are a machine vector width number of rays and are traced by OSPRay in Z-order through a tile. For each Z-order packet in the tile, we store the depth and sample pairs for the rays in the packet in an array. Each index in this array are the M samples for the ith sample point for each ray in the packet, where M is the vector width of the target CPU. With this streaming and vectorization friendly layout, operations to load/store all samples at index i are made with a single vector load/store instead of separate scalar load/stores or gather/scatters. Each ray also stores a small header tracking the number of samples currently in its buffer for determining when it is full. We could also optionally track the effective sampling rate of the ray and use this to determine when each pixel is finished.

To reduce the number of sample buffers allocated, and thus the memory required by our method, buffers are allocated on demand by a cache for each image tile that views a portion of the volume. This cache allocates the sample buffers for an entire tile once a packet requests to write to a buffer in a tile that has not yet been allocated.

4.2. Image-Parallel SAVR

In image-parallel volume rendering, the volume data is replicated on each node, and work is divided across multiple nodes by partitioning the image tiles to be rendered among the workers. When rendering work is distributed among multiple nodes in this way, each node needs to store the buffers only for the portion of the framebuffer it renders, reducing memory requirements per-node. This reduction is handled transparently by our caching system. In image-parallel rendering OSPRay assigns different image tiles to the workers, each worker only samples rays in its assigned tiles and thus only requests buffers from the cache which it needs for those tiles.

This reduction in memory requirements enables end applications to be more flexible in tuning the number of samples stored in each buffer and to render larger images. In an extreme case of 4K rendering (4096×2160) with 2048 samples per buffer, 145GB of space is required for the samples alone. If rendering work is distributed among 16 nodes in this scenario, each node would need to store only 9.06GB of the sample buffers, easily fitting into the memory budgets of recent HPC nodes or even in a KNL's MCDRAM or GPU's VRAM.

The sorting and blending of samples in the SAVR algorithm is independent per ray-packet, and since packets are localized to tiles, SAVR introduces no extra communication and thus no decrease in scalability over existing image-parallel direct volume rendering techniques.

4.3. Data-Distributed SAVR

OSPRay supports a method of sort-last tile-based compositing [WJA*16]. Volume data is partitioned into disjoint blocks, typically one per node, and assigned to worker nodes for rendering. Each block projects to some subset of the tiles in the frame buffer which the worker is then responsible for rendering. As tiles are finished, they are handed off to a sort-last compositing algorithm to combine the worker's results into the final image.

Similar to the image-parallel case, only tiles onto which a node's data projects are rendered, and the resulting reduction in memory requirements is similar as well, though somewhat less predictable due to reliance on the underlying data distribution. Some nodes may render overlapping tiles, where each has a subregion of the data projecting to the tile. In this case the nodes can use smaller buffers respective to their data region's depth along the camera axis, effectively splitting the memory cost of storing the samples for that region of data.

Although not explored in this paper, implementations have the flexibility to send the volume samples themselves, if the added bandwidth requirement is acceptable. Such techniques may allow for global sample blending operations to be performed. On demand paging of remote data from other nodes as needed could also be implemented with our technique, with more data being streamed as needed over the course of several frames to alleviate costly data transfers. These options are left for future work.

5. Results

We evaluate the convergence of our method's progressive refinement and performance compared to naïve "unsorted" DVR with a large sampling step size of 8 voxels (80 on *Cosmic Web*) and DVR sampling at the Nyquist limit of two samples per voxel to render the same quality as SAVR, listed as "Ideal Sampling". Though incorrect from the standpoint of order-dependent integration, unsorted DVR provides an upper bound on performance given a certain step size.

To demonstrate the flexibility of our approach, we run on a variety of different volumetric datasets, as well as an AMR volume dataset using a prototype AMR sampling method and a particle RBF model of cosmology data. The latter two datasets are costly to sample, and are where our method is most applicable, as mentioned in Figure 3. In the case of uniform grid volumes with trilinear interpolation the sampling can be cheap enough that rendering at the ideal sampling rate gives acceptable performance. The datasets used for benchmarking are shown in Figure 4. The uniform grid volumes use fast trilinear interpolation, whereas computing a sample for the RBF model requires a costly k-d tree query and combination of Gaussians. We also measure performance in image-parallel and data-distributed volume rendering applications and demonstrate that SAVR extends well to these contexts.

Our benchmarks were run on the *Stampede 1.5 KNL* partition and *Lonestar5* cluster at the Texas Advanced Computing Center. Single node performance for a *Stampede 1.5* KNL node and for a *Lonestar5* node is shown in Table 1. Evaluation of single node performance and convergence quality can be found in Section 5.1,

image-parallel and data-distributed performance are discussed in Section 5.2.

5.1. Single Node Performance

For the purposes of classification, we break the SAVR rendering sequence into three phases: interacting, accumulating, and filled. These phases can be seen in Figure 5a and Figure 5b, which show our method's quality and performance dependence on the number of samples stored per ray.

In the interacting phase, the user is moving the camera and we keep only a single pass of samples, as each new camera position invalidates our previous set. At this stage, the rendering quality of our algorithm is similar to that of DVR. Additionally, the merging of samples after traversal is not required, and we find performance is around $2\times$ to $3\times$ slower than unsorted DVR, see max FPS for unsorted and SAVR columns in Table 1. This reduction in performance is at least partially due to loss of early ray termination in our current implementation. However, without removing the ability to interactively change the transfer function while refining, using early ray termination would give incorrect results. Another option to address this issue would be to use unsorted DVR when interacting, and switch to SAVR when the camera motion has stopped.

In the accumulating phase, the user has stopped moving the camera, and samples are accumulated each frame to refine the rendering. Here SAVR drops in framerate as the sample merging step impacts performance. There are different options for implementing this merge, and it should be possible to alleviate this performance issue in the future by using a faster sorting method. Further, for non-uniform grid volumes and unstructured data this merging step is less expensive relative to sampling as shown on the *Cosmic Web* and *Landing Gear* benchmarks in Table 1. This decrease can be seen in Figure 5a and Figure 5b where we start accumulating at frame 20, and the framerate decreases as the merge becomes increasingly expensive over a greater number of accumulated samples.

As the sample buffers fill to capacity, the algorithm switches to the filled phase. In this phase the sample accumulation stage is skipped, and the previously accumulated samples are used to compute the pixel color. This is shown in Figure 5a and Figure 5b where after some number of frames of accumulation the framerate improves, depending on how many samples are being blended each frame. Optionally the end application could detect when the volume is sampled sufficiently and stop rendering new frames, e.g. after reaching the vertical line in each plot.

Interacting performance is not as high as the sparsely sampled unsorted DVR; however, our method will converge to the correct result and here performance is a great improvement over ideally sampled DVR which renders the same quality result as SAVR. For challenging data such as the AMR dataset and cosmology RBF model we reach nearly the same framerate as unsorted DVR, while converging to a significantly better result. In all cases when moving the camera, our method remains interactive and only when stopping motion to focus on a feature and resolve the image does rendering performance decrease. When accumulating to the final image the user is also still able to change the transfer function without
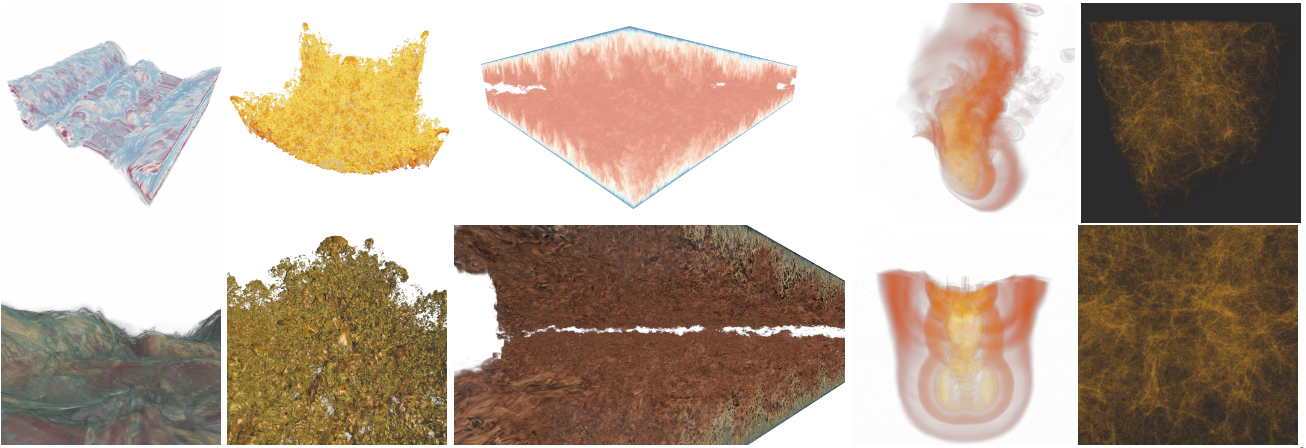
Figure 4: *Datasets used for benchmarking and convergence comparisons, from left to right:* Magnetic Reconnection *(512MB, 512^3),* Richtmyer–Meshkov *(8GB, $2048 \times 2048 \times 1920$),* DNS *(1TB), the* Landing Gear *AMR dataset (40 GB) [KBH*14], and an unstructured particle RBF model of the* Cosmic Web *early universe simulation with 178.9M particles. The top row shows the far viewpoint images, marked (F), and the bottom row shows the close viewpoint images, marked (C).*

			DVR		
Dataset	SAVR	SAVR + Gradients	Ideal Sampling	Ideal Sampling + Gradients	Unsorted
Single *Stampede 1.5* node, Intel® Xeon Phi™ 7250 processor in cache mode with 96GB of RAM					
Magn. Recon. (F)	28.62 (3.36)	16.93 (0.92)	6.78 (6.71)	2.66 (2.62)	59.46 (56.10)
Magn. Recon. (C)	15.40 (2.77)	8.91 (0.73)	4.01 (3.99)	1.49 (1.58)	36.99 (35.43)
R–M (F)	14.72 (5.31)	9.61 (0.83)	6.43 (6.32)	2.23 (2.19)	32.58 (30.94)
R–M (C)	6.67 (3.62)	4.69 (0.60)	3.27 (3.24)	1.18 (1.18)	18.01 (17.28)
Landing Gear (F)	21.78 (6.19)	N/A	4.60 (4.54)	N/A	36.13 (31.89)
Landing Gear (C)	12.16 (3.99)	N/A	2.09 (2.08)	N/A	20.05 (18.75)
Cosmic Web (F)	2.63 (2.09)	N/A	0.026 (0.025)	N/A	2.69 (2.65)
Cosmic Web (C)	1.13 (0.98)	N/A	0.016 (0.0159)	N/A	1.29 (1.27)
Single *Lonestar5* node, dual socket Intel® Xeon® E5-2690 v3 processor with 64GB of RAM					
Magn. Recon. (F)	19.54 (1.81)	10.02 (0.42)	3.91 (3.88)	1.49 (1.47)	37.41 (36.85)
Magn. Recon. (C)	10.46 (1.44)	5.44 (0.32)	2.18 (2.17)	0.79 (0.79)	22.53 (22.35)
R–M (F)	9.60 (2.95)	6.74 (0.56)	4.51 (4.47)	1.73 (1.71)	22.69 (22.53)
R–M (C)	4.23 (1.93)	3.13 (0.39)	2.22 (2.21)	0.84 (0.84)	11.83 (11.75)
Landing Gear (F)	24.66 (3.71)	N/A	4.07 (4.02)	N/A	40.35 (38.59)
Landing Gear (C)	11.74 (2.22)	N/A	1.71 (1.71)	N/A	20.29 (19.81)
Cosmic Web (F)	0.13 (0.12)	N/A	0.0195 (0.0191)	N/A	0.12 (0.11)
Cosmic Web (C)	0.25 (0.21)	N/A	0.0130 (0.0127)	N/A	0.25 (0.21)

Table 1: *Framerate of DVR methods vs. SAVR. We show the max framerate (interactive phase) and mean in parentheses (expected framerate across all phases). Ideal sampling is DVR at two samples per voxel, rendering equivalent quality to SAVR's converged result. Unsorted does not converge correctly, but provides a useful performance baseline.*

needing to recompute the volume samples, allowing for interactive exploration of the final rendering.

5.2. Distributed Performance

As discussed in Section 4.2, each image tile being rendered with SAVR is independent of others and as a result SAVR follows scaling trends similar to OSPRay's existing image-parallel volume rendering. In Figure 6 we compare both the max framerate, our performance while interacting, and the median framerate, our expected performance across all phases, against ideally sampled DVR on the *Landing Gear*.

We find that SAVR follows scaling trends similar to DVR and, as in the single node case, provides better performance than ideal sampling when interacting while not being significantly slower while accumulating to the final result. The drop in median framerate at 64 workers may be due to some machine or compositing performance fluctuations, we found the standard deviation of the frame time for this run was large (± 22 FPS). This case warrants some further investigation though is likely unrelated to SAVR, as it does not modify the compositing code used by the renderer.

Our data-distributed implementation of the SAVR algorithm fits well into OSPRay's existing sort-last distributed volume rendering, as described in Section 4.3. In the data-distributed case, we partition the volume with a grid such that we assign one brick to each
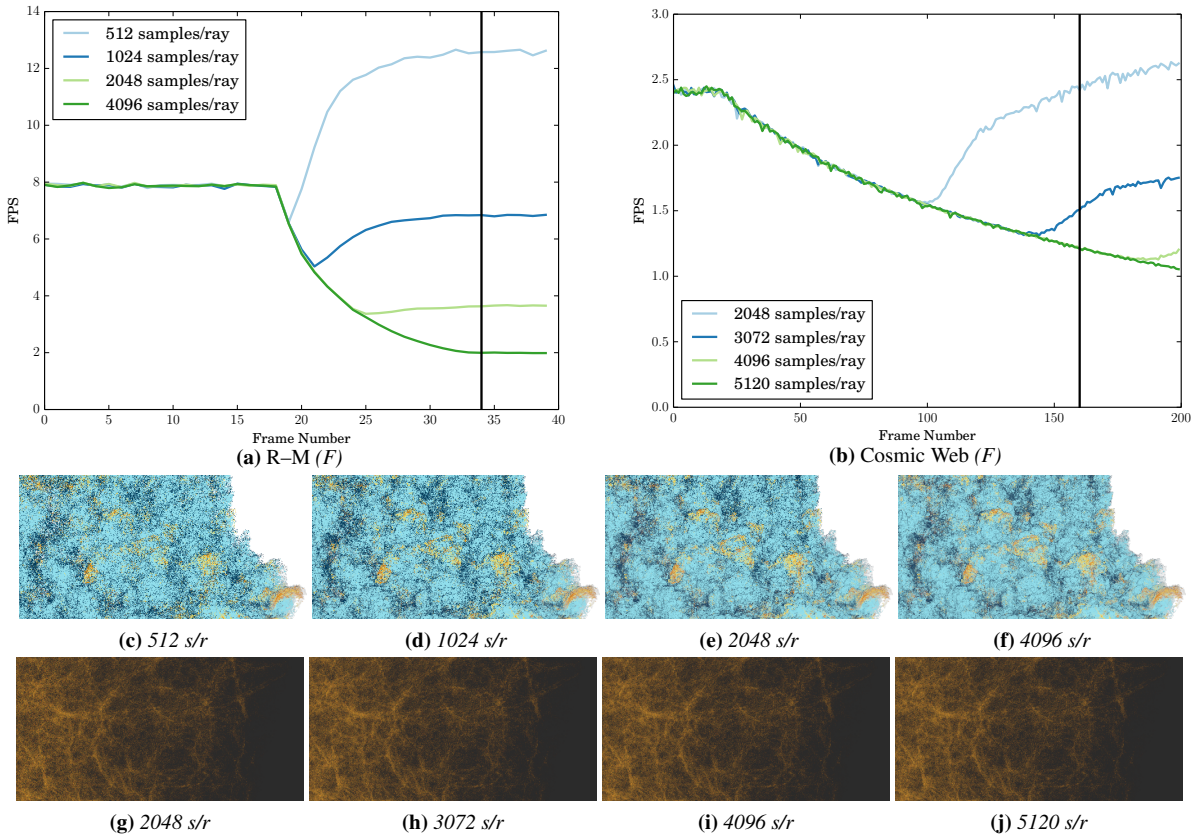
(a) R–M *(F)*

(b) Cosmic Web *(F)*

(c) *512 s/r*

(d) *1024 s/r*

(e) *2048 s/r*

(f) *4096 s/r*

(g) *2048 s/r*

(h) *3072 s/r*

(i) *4096 s/r*

(j) *5120 s/r*

Figure 5: *Performance and quality vs. number of samples/ray. During the first 20 frames the user is interacting and no accumulation is performed. From frame 20 on we accumulate samples and see a dip in performance until the buffers are filled. At the vertical line on (a) and (b) the volume is sampled to the Nyquist limit. (c-j) Show rendered results for both datasets, for* R–M *512 or 1024 samples are insufficient to sample the volume, for* Cosmic Web *2048 is insufficient.*
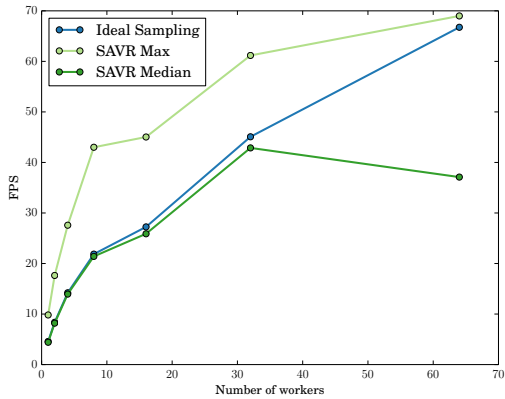

Figure 6: *Image parallel scaling on* Stampede 1.5 *KNL nodes rendering the far viewpoint of the* Landing Gear.
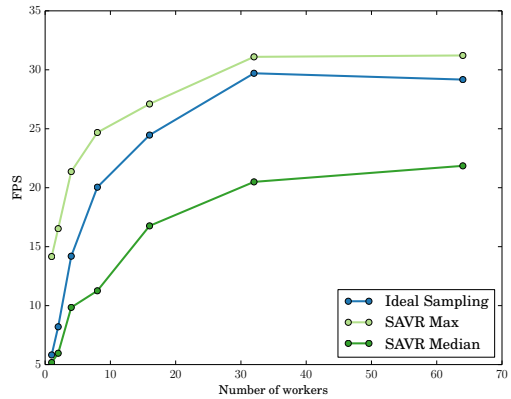

Figure 7: *Data distributed on* Stampede 1.5 *KNL nodes rendering the far viewpoint of the* Richtmyer–Meshkov *dataset.*

worker for rendering. We compare scaling vs. DVR on the *R–M* dataset in Figure 7 and the *DNS* in Table 2. We find that in the case of the *R–M* dataset, we see scaling similar to DVR, as in the image parallel case. On the *DNS* dataset, we see better scaling than DVR when increasing from 32 workers to 64.

The *DNS* benchmarks demonstrate a large volume being rendered at a high resolution, 1920×1080, storing 3000 samples per buffer, which results in demanding memory and compute require-

ments. While the volume is large, in the data-distributed case each worker only needs to store enough samples for its block, reducing the memory requirements of SAVR. When the number of nodes is doubled, the amount of volume data for which each is responsible is cut in half, from 30.2GB to 15.1GB, and the number of tiles to which each brick projects also decreases, reducing memory demands and work required per node. The interacting framerate (max), and often the median framerate as well, are higher than

Rendering Mode	32 Workers	64 Workers
Far viewpoint		
SAVR	9.24 (8.66)	15.81 (8.29)
SAVR + Gradients	6.58 (1.49)	10.01 (1.32)
Ideal Sampling	5.26 (5.16)	5.56 (5.37)
I.S. + Gradients	1.97 (1.92)	2.06 (2.01)
Unsorted DVR	20.69 (15.85)	20.17 (5.28)
Close viewpoint		
SAVR	4.28 (3.26)	8.73 (2.62)
SAVR + Gradients	2.84 (0.53)	4.37 (0.43)
Ideal Sampling	1.81 (1.77)	2.02 (1.97)
I.S. + Gradients	0.65 (0.64)	0.72 (0.70)
Unsorted DVR	13.69 (12.67)	14.41 (13.21)

Table 2: *Framerate for data-distributed rendering of the* DNS *on* Stampede 1.5 *KNL nodes rendering at* 1920×1080. *We show both the max framerate (interactive phase) and the mean in parentheses (expected framerate across all phases).*

DVR at the ideal sampling rate while our method converges to the same result.

5.3. Limitations and Comparison to Other Approaches

The main performance bottleneck of our current implementation is the merging algorithm used to maintain the sorted sample buffers, as shown in the drop at frame 20 in Figure 5a and Figure 5b. This merge could be parallelized or better vectorized. Alternatively, it may be possible to not merge the individual ray's samples at all, but instead perform an ordered traversal of the samples while blending. We also find that at larger sample buffer sizes, classification and blending steps incur significant cost (see the tails of Figure 5a and Figure 5b at high sample counts).

Moreover, when comparing the framerate of SAVR while interacting, performance does not match unsorted DVR, likely due to buffer copies when merging samples and loss of early ray termination, which cannot be implemented without the full set of classified samples. The maximum required set of samples could be deduced during sample accumulation for some given transfer function; however, this optimization would require the sample cache to be recomputed upon modifications to the transfer function. Even with these limitations, however, SAVR remains a compelling method for progressively rendering large volume data, especially when computing samples is costly.

We have deliberately evaluated SAVR in the context of uniformly sampled volume rendering. It would, however, be desirable to combine this technique with adaptive sampling approaches, which would better leverage empty space and heterogeneous data. Adaptive volume rendering approaches have recently been added to OSPRay, and have previously proven powerful on both GPU [LLYM04, HSS*05] and CPU [KTW*11]. These adaptive techniques are orthogonal to the progressive rendering method of SAVR; full consideration of these combined approaches remains as future work.

Out-of-core approaches (see, e.g., the survey of Beyer and Had-

wiger [BHP15]), in particular on GPU architectures, are a compelling alternative to volume rendering of full-resolution data using the techniques described in our work. In particular, new sparse octree capabilities on GPUs allow for straightforward adoption of ray-guided LOD techniques in the spirit of Fogal et al. [FSK13]. Similarly, use of volumetric LOD techniques are orthogonal to our progressive rendering method. Recent GPU work has shown framerates higher than our approach by combining efficient LOD and progressive refinement [HBJP12, CNLE09]. An advantage of the SAVR approach is that it provides a progressive rendering solution that always operates on native data resolution, assuming the architecture has sufficient memory. When that is not the case, it would be possible (and desirable) to pair SAVR with out-of-core techniques, regardless of the underlying architecture (CPU or GPU).

6. Discussion and Future Work

We have presented sample-accumulation volume rendering, a novel method for progressive refinement of volume data in object space, which supports arbitrary structured or unstructured volume data and extends to both image-parallel and data-parallel distributed rendering configurations. Unlike naïve averaging and image-space progressive refinement, our method ensures the refined image converges to the correct result while still maintaining an interactive framerate and providing a smooth transition from low to high quality. This technique allows for terascale data to be rendered progressively and interactively on few compute resources, and is particularly well-suited for large-memory CPU architectures.

The main drawback of our approach, the memory required to store samples and additional time required to sort them, is less of a concern in a distributed setting; moroever our method scales similarly to standard DVR techniques. SAVR is also independent of how samples are computed, and does not prevent one from taking advantage of more advanced adaptive sampling schemes or LOD.

Although we have focused our evaluation of SAVR on CPU architectures, there is nothing inherent in our method that prevents it from being applied in a GPU volume rendering context. Recent GPUs such as the Tesla P40 have 24GB of memory, and a single one could handle small to medium datasets at reasonable framebuffer sizes, and switch to data-distributed or image-parallel rendering to handle larger data or framebuffers. We leave exploring GPU implementation possibilities for future work.

Finally, we would like to examine possibilities for reducing the memory required by SAVR. Even though the cost per node is reduced in distributed environments, on many current HPC nodes with limited memory it can still be an issue. Although the *Stampede 1.5* KNL nodes have 96GB of RAM, this amount is high by HPC standards. Reducing memory requirements further would allow SAVR to be used in more memory-constrained environments, and may improve performance due to reduced memory bandwidth.

7. Acknowledgements

References

[AGGW15] AMSTUTZ J., GRIBBLE C., GÜNTHER J., WALD I.: An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (2015), 72–88. 4

[Aya15] AYACHIT U.: *The Paraview Guide: A Parallel Visualization Application.* Kitware, Inc., 2015. 2

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-Art in GPU-Based Large-Scale Volume Visualization. In *Computer Graphics Forum* (2015), vol. 34, Wiley Online Library, pp. 13–37. 2, 8

[Car84] CARPENTER L.: The A-buffer, an antialiased hidden surface method. *ACM Siggraph Computer Graphics 18*, 3 (1984), 103–108. 2

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295. 2

[CN93] CULLIP T. J., NEUMANN U.: Accelerating Volume Reconstruction with 3D Texture Hardware. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games* (2009), ACM, pp. 15–22. 2, 8

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume Rendering. In *ACM Siggraph Computer Graphics* (1988), vol. 22, ACM, pp. 65–74. 2

[DPH*03] DEMARLE D., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE PVG* (2003), pp. 87–94. 3

[FFSE14] FERNANDES O., FREY S., SADLO F., ERTL T.: Space-time volumetric depth images for in-situ visualization. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on* (2014), IEEE, pp. 59–65. 2

[FSE13] FREY S., SADLO F., ERTL T.: Explorable volumetric depth images from raycasting. In *2013 XXVI Conference on Graphics, Patterns and Images* (2013), IEEE, pp. 123–130. 2

[FSK13] FOGAL T., SCHIEWE A., KRÜGER J.: An analysis of scalable GPU-based ray-guided volume rendering. 8

[FSME14] FREY S., SADLO F., MA K.-L., ERTL T.: Interactive progressive visualization with space-time error control. *IEEE transactions on visualization and computer graphics 20*, 12 (2014), 2397–2406. 2

[HBJP12] HADWIGER M., BEYER J., JEONG W.-K., PFISTER H.: Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012), 2285–2294. 2, 8

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312. 8

[Hsu93] HSU W. M.: Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering* (1993). 3

[KBH*14] KIRIS C. C., BARAD M. F., HOUSMAN J. A., SOZER E., BREHM C., MOINI-YEKTA S.: The lava computational fluid dynamics solver. In *52nd Aerospace Sciences Meeting* (2014), p. 0070. 6, 9

[KC93] KE H.-R., CHANG R.-C.: Sample buffer: A progressive refinement ray-casting algorithm for volume rendering. *Computers & Graphics 17*, 3 (1993), 277–283. 2

[KM05] KAUFMAN A., MUELLER K.: Overview of Volume Rendering. *The Visualization Handbook 7* (2005), 127–174. 2

[KTW*11] KNOLL A., THELEN S., WALD I., HANSEN C. D., HAGEN H., PAPKA M. E.: Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *Proceedings of the 2011 IEEE Pacific Visualization Symposium (PacificVis)* (2011), pp. 3–10. 2, 8

[KWN*14] KNOLL A., WALD I., NAVRATIL P., BOWEN A., REDA K., PAPKA M. E., GAITHER K.: RBF Volume Ray Casting on Multicore and Manycore CPUs. *Computer Graphics Forum 33* (2014). 2

[Lev88] LEVOY M.: Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37. 2

[Lev90] LEVOY M.: Volume Rendering by Adaptive Refinement. *The Visual Computer 6*, 1 (1990), 2–7. 2

[LH91] LAUR D., HANRAHAN P.: Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1991), SIGGRAPH '91, ACM, pp. 285–288. 2

[LLYM04] LJUNG P., LUNDSTROM C., YNNERMAN A., MUSETH K.: Transfer function based adaptive decompression for volume rendering of large medical data sets. In *Volume Visualization and Graphics, 2004 IEEE Symposium on* (2004), IEEE, pp. 25–32. 8

[LV00] LOKOVIC T., VEACH E.: Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 385–392. 2

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel Volume Rendering Using Binary-Swap Compositing. *IEEE Comput. Graph. Appl. 14*, 4 (1994), 59–68. 3

[MSRMH09] MEYER-SPRADOW J., ROPINSKI T., MENSMANN J., HINRICHS K.: Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations. *IEEE Computer Graphics and Applications 29*, 6 (2009). 2

[NLKH12] NELSON B., LIU E., KIRBY R. M., HAIMES R.: ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions. *IEEE Transactions on Visualization and Computer Graphics 18*, 12 (2012), 2325–2334. 2, 4

[OAJ*16] O'LEARY P., AHRENS J., JOURDAIN S., WITTENBURG S., ROGERS D. H., PETERSEN M.: Cinema image-based in situ analysis and visualization of MPAS-ocean simulations. *Parallel Computing 55* (2016), 43–48. 2

[OF91] OHBUCHI R., FUCHS H.: Incremental volume rendereing algorithm for interactive 3D ultrasound imaging. In *Biennial International Conference on Information Processing in Medical Imaging* (1991), Springer, pp. 486–500. 1, 2

[OKK10] ORTHMANN J., KELLER M., KOLB A.: Topology-Caching for Dynamic Particle Volume Raycasting. In *Proceedings of Vision, Modeling and Visualization 2010, Siegen, Germany* (2010), Eurographics, pp. 147–154. 2

[PM12] PHARR M., MARK B.: ISPC: A SPMD Compiler for High-Performance CPU Programming. In *Proceedings of Innovative Parallel Computing (inPar)* (2012), pp. 184–196. 2, 4

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3 (1999), 238–250. 2

[Sab88] SABELLA P.: A Rendering Algorithm for Visualizing 3D Scalar Fields. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 51–58. 2

[SKKK09] SAKAMOTO N., KAWAMURA T., KUWANO H., KOYAMADA
    K.: Sorting-free pre-integrated projected tetrahedra. In *Proceedings of
    the 2009 Workshop on Ultrascale Visualization* (2009), ACM, pp. 11–18.
    2

[TCM10] TIKHONOVA A., CORREA C. D., MA K.-L.: Explorable im-
    ages for visualizing volume data. In *PacificVis* (2010), Citeseer, pp. 177–
    184. 2

[Wes90] WESTOVER L.: Footprint Evaluation for Volume Rendering.
    ACM Siggraph Computer Graphics 24, 4 (1990), 367–376. 2

[WJA*16] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C.,
    KNOLL A., JEFFERS J., GÜNTHER J., NAVRATIL P.: OSPRay — A
    CPU Ray Tracing Framework for Scientific Visualization. *IEEE Trans-
    actions on Visualization and Computer Graphics PP*, 99 (2016), 1–1. 2,
    3, 4, 5