IEEE Access
practical innovations : open solutions

Received June 21, 2013, accepted August 1, 2013, date of publication September 9, 2013, date of current version September 20, 2013.

Digital Object Identifier 10.1109/ACCESS.2013.2281080

GPU-Accelerated Visualization of Scattered Point Data

THOMAS L. FALCH[1], JOSTEIN BØ FLØYSTAD[2], DAG W. BREIBY[2], AND ANNE C. ELSTER[1] (Senior Member, IEEE)

[1]Department of Computer and Information Science, Norwegian University of Science and Technology, N-7491 Trondheim, Norway
[2]Department of Physics, Norwegian University of Science and Technology, N-7491 Trondheim, Norway

Corresponding authors: T. L. Falch (thomafal@idi.ntnu.no), A. C. Elster (elster@ntnu.no)

ABSTRACT As data sets continue to grow in size, visualization has become a vitally important tool for extracting meaningful knowledge. Scattered point data, which are unordered sets of point coordinates with associated measured values, arise in many contexts, such as scientific experiments, sensor networks, and numerical simulations. In this paper, we present a method for visualizing such scattered point data sets. Our method is based on volume ray casting, and distinguishes itself by operating directly on the unstructured samples, rather than resampling them to form voxels. We estimate the intensity of the volume at points along the rays by interpolation using nearby samples, taking advantage of an octree to facilitate efficient range search. The method has been implemented on multi-core CPUs, GPUs as well as multi-GPU systems.[1] To test our method, actual X-ray diffraction data sets have been used, consisting of up to 240 million data points. We are able to generate images of good quality and achieve interactive frame rates in favorable cases. The GPU implementation (Nvidia Tesla K20) achieves speedups of 8–14 compared with our parallelized CPU version (4-core, hyperthreaded Intel i7 3770K).

INDEX TERMS GPGPU, multi-GPU, reciprocal space maps, scattered point data, volume visualization, volume ray casting, X-ray scattering.

I. INTRODUCTION

Due to improving sensor technologies, as well as increasing size and fidelity of numerical simulations, scientific datasets are growing dramatically in size. Often, the only viable way of interpreting such datasets is through visualization. Volumetric data based on unstructured grids and scattered point data are becoming increasingly important. Here, the locations of the available data points (hereafter: *samples*) from an underlying (spatial) distribution are irregularly distributed throughout the volume. This differs from the traditional voxel format, which requires samples on a rectilinear, uniform grid, as exemplified by a stack of MRI images.

Scattered point data arises in many areas, including sensor networks, which are used to measure physical and environmental conditions at various locations. Examples include sensors that measure precipitation at various geographical locations, or the temperature at various locations in a furnace. There is usually no predefined or implied connectivity between the sensor sites. The output of several numerical simulations, such as smoothed particle hydrodynamics (SPH) and n-body simulations can also be regarded as scattered point data. Finally, even if the data is originally grid based, the structure can be lost if it is post-processed, of if the data is streamed over a network.

The result of X-ray diffraction experiments, which are used to study the structure, chemical composition and physical properties of materials, is another example of scattered point data, which motivated this work. In a generic diffraction experiment, measurements of how a material *scatters*, or changes the direction of, X-rays are made. The outcome of these experiments is samples of the diffracted X-ray intensity—a three-dimensional scalar field—at locations determined by details of the experimental setup. To extract information about the material being studied, it is often helpful to create high quality 3D visualizations of this data. This is just one example of the increasing importance of volume visualization in science.

Well established methods exist for visualizing volumetric data on structured as well as unstructured grids, but scattered

[1]Our source code is available under a BSD license at https:// github.com/acelster/scatter-pt-viz

point data remains a challenge. One approach is to resample the data on a uniform grid, or tetrahedralize it to create an unstructured grid, and then visualize the resulting grid. This can, however, be problematic. If resampling is done, a grid with sufficiently high resolution will require vast amounts of memory. On the other hand, using a low resolution grid is equivalent to low-pass filtering, and will cause details to be lost, resulting in poor image quality. With tetrahedralization, one can typically only perform interpolation at the cell faces, which may cause poor results with cells of widely varying sizes.

In this paper, we present a method for visualizing scattered point data, primarily developed for X-ray diffraction data, but applicable also in other contexts. Our method is based on the well established technique of volume ray casting [1], [2]. However, our method differs by not resampling the data on a uniform grid, but instead operating directly on the scattered point data. Due to the highly parallel and computationally intensive nature of the method, we have implemented it on single- and multi-GPU systems for increased performance.

The remainder of this article is structured as follows: the following section presents previous work on volume visualization of scattered data. Section III provides background information about GPU computing, scattered point data, and X-ray diffraction. Section IV describes our visualization method, as well as modifications and optimizations made for the GPU version. Results are presented in Section V, while Section VI concludes and describes possibilities for further research.

II. RELATED WORK

An overview of direct volume visualization can be found in the book by Hansen and Johnson [1]. The volume ray casting algorithm was originally proposed by Levoy [2] for regular grids, and extended by Garrity [3] to irregular grids. In addition to these image order techniques, the object order technique of splatting was introduced by Westover [4]. With the advent of programmable GPUs, this platform was adopted for volume visualization. Early efforts emulated ray casting with texture mapping [5], [6]. As the flexibility of GPUs increased they were also used for full ray casting [7]–[9], as well as the related technique of ray tracing [10].

Most volume visualization techniques require the underlying function to be reconstructed based on the samples. This is fairly straightforward in the case of regular grids, where trilinear interpolation is used. In addition, several techniques for modeling and interpolating scattered point data also exist, for instance the works by Nielson [11] and Amidror [12].

Much work has also been done on visualizing scattered point data. Some approaches simply resample the data on a regular grid, and then render this grid using standard techniques [13]–[16]. Other approaches operate directly on the scattered data. This includes techniques that are based on splatting, such as the one adopted by Hopf and Ertl [17].

Techniques based on ray casting have also been employed to directly render scattered point data; Chen [18] used an approach based on radial basis functions, with one radial basis function per point. At equidistant positions along the rays, neighbouring points were found using an octree, and their radial basis functions were evaluated to find the value of the underlying function. Jang *et al.* [19] also used radial basis functions, but with fewer radial basis functions than points. Their approach was based on texture mapping, but rather than using a 3D texture, the radial basis functions were evaluated when the volume slices were rasterized. Ledergerber *et al.* [20] proposed a unified framework for both structured and unstructured datasets, based on moving least squares. At equidistant positions along the rays, close samples were found and used to compute a weighted least squares approximation. The method also supports anisotropic weights. A GPU implementation was also provided.

While our approach resembles these three last mentioned, it differs in several ways: We use inverse distance weighing for interpolation; have developed a novel empty-space skipping technique to improve performance; and adapted a filtering approach which can dramatically reduce the size of the input dataset, without compromising image quality. We have implemented our method for both single- and multi-GPU systems, and have developed a load balancing scheme for the multi-GPU case. Finally, we provide detailed results, for experimental X-ray diffraction data, for both image quality and performance.

III. BACKGROUND

In this section, we will provide the background information about GPU computing, scattered point data and X-ray diffraction necessary for the full appreciation of our work.

A. GPU COMPUTING

GPUs were originally developed as dedicated coprocessors for 3D graphics, but their increasing programmability, combined with high performance, low cost and low energy consumption, have made them highly popular for general purpose high performance computation [21], [22]. For instance, GPUs have been used to speed up SPH simulations [23], as well as applications in biology [24] and medicine [25]–[27]. In the following, we will give a brief introduction to the architecture of typical high-end GPUs.

A GPU consists of a number of *multiprocessors*, each of which consists of several *streaming processors*. Each of the streaming processors of a multiprocessor works in a SIMD fashion, executing the same instruction in lock-step. Hence, GPUs are well suited for data parallel problems, where each thread executes the same program, but with different input data. Furthermore, the high number of streaming processors, which range from several hundred to several thousand for current GPUs, requires a high degree of inherent parallelism in the problem for GPUs to achieve good efficiency.

However, even high-end GPUs typically lack advanced features such as branch prediction and out-of-order execution. In addition, more of the die is devoted to computational units

and less to caches, compared to CPUs, and they thus typically run at lower clock frequencies.

B. SCATTERED POINT DATA

A *scattered point dataset* can be defined as a set of samples $S = \{s_1, s_2 \dots s_N\} = \{(p_1, f_1), (p_2, f_2) \dots (p_N, f_N)\}$ where $p_i \in \mathbb{R}^m$ are coordinate vectors in m-dimensional space with associated scalar values $f_i \in \mathbb{R}$. In the case of $m = 3$, we deal with *volumetric* scattered point data.

In general, the sample locations are arbitrarily distributed, and there is no connectivity between samples. However, two special cases frequently arise in practice: the sample locations may be on a uniform, rectilinear grid, commonly referred to as *voxel data*, or the samples may be located on a $m - 1$ dimensional manifold (*e.g.* on the surface of some object in 3D space), commonly referred to as *point clouds*.

The data can often be regarded as being samples of a continuous function f. Reconstructing this function from the samples is often necessary for visualization. Many methods exist for this problem, see *e.g.* [11]. Trilinear interpolation is frequently used with grid data, due to its conceptual and computational simplicity. In the case of scattered point datasets where the sample locations do not exhibit any particular structure, *inverse distance weighing* (IDW) [28] is an applicable method. In its simplest form, the estimated function value $f_e(x)$ at position x is

$$f_e(x) = B^{-1} \sum_{i=1}^{N} d(p_i, x)^{-u} f_i, \tag{1}$$

where $u > 0$ is an adjustable parameter, $d(x, y)$ is the Euclidean distance between x and y, and $B = \sum_{i=1}^{N} d(p_i, x)^{-u}$. It is clear that the samples located closest to x has far stronger influence on the estimated function value $f_e(x)$, thus the sums in equation (1) can be truncated to include only those samples located close to x.

If it is known that the underlying function varies more rapidly in one direction than others, it will generally make sense to weigh samples along the direction of slow change more heavily compared to samples along the direction of rapid change [29]. When IDW is used, this can be achieved by using an anisotropic distance measure, instead of the Euclidean distance [30]. The anisotropic distance $d_a(x, y)$ between two points x and y can be defined as

$$d_a(x, y) = \sqrt{(x - y)^T A\,(x - y)}, \tag{2}$$

where the symmetric $m \times m$ matrix A describes the equidistance ellipsoid.

C. X-RAY DIFFRACTION

To explain the nonuniform distribution of the X-ray data, used as our motivational test case, we will provide a brief introduction to X-ray diffraction [31].

The wavelength λ of X-rays (0.01–10 nm) has the same order of magnitude as the distance between atoms and molecules in condensed matter. X-rays passing through a material will be scattered, that is, spread in new and different directions, and also interfere with each other. We emphasize that the dataset being scattered point data is not caused by the X-rays being scattered, but by the way the scattered X-rays are measured. Measurements of the resulting intensity distribution as function of direction can be used to extract detailed information about the structural arrangements inside the material specimen [31].

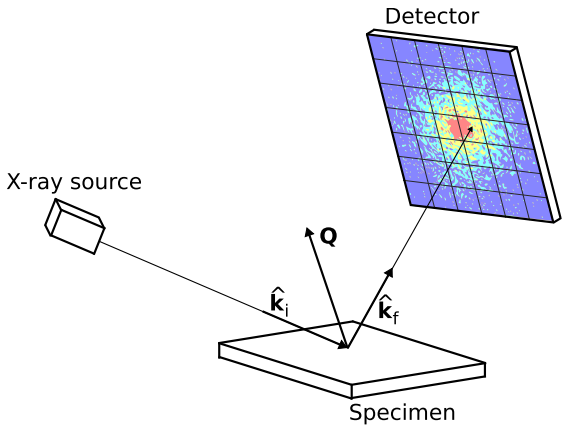


FIGURE 1. Generic X-ray scattering experimental setup. X-rays from the source (direction $\hat{\mathbf{k}}_i$) hit the sample, and are scattered into direction $\hat{\mathbf{k}}_f$. The scattered X-rays are measured with a pixelated area detector. The measured intensity of each pixel I, combined with the corresponding scattering vector $\mathbf{Q}$ results in a sample $(\mathbf{Q}, I)$.

The setup of a generic X-ray scattering experiment is shown in Fig. 1. A 2D sensor array is used to measure the intensity distribution of scattered X-rays from the beam incident on the material specimen. For each pixel of the detector, the corresponding scattering vector $\mathbf{Q} \equiv 2\pi(\hat{\mathbf{k}}_f - \hat{\mathbf{k}}_i)/\lambda$ can be computed to obtain one sample $(\mathbf{Q}, I)$. The relative orientation of the detector, material specimen and incoming beam can be varied to cover the region of $\mathbf{Q}$-space of interest.

If the components of the scattering vector are interpreted as 3D space coordinates in $\mathbf{Q}$-space, all samples from a single detector frame will be placed on the same curved 2D surface. Measuring multiple frames with different sample-detector configurations will result in multiple surfaces, each with different curvature, orientation and position. Such a set of frames can be used to effectively map the diffracted intensity in a volume of $\mathbf{Q}$-space, resulting in a *reciprocal space map* [32]. The distance between pairs of surfaces in $\mathbf{Q}$-space is typically different from the distance between samples located on the same surface, and surfaces may intersect. Although the structure of the sensor array and the curvature of the $\mathbf{Q}$-space surface corresponding to a particular frame may be used to acquire some connectivity between the samples, shadowing effects from the experimental setup combined with masked or insensitive detector areas complicate strategies utilizing connectivity information, suggesting that regarding it as an unstructured scattered point data set is a viable option.

IV. RAY CASTING FOR SCATTERED POINT DATA

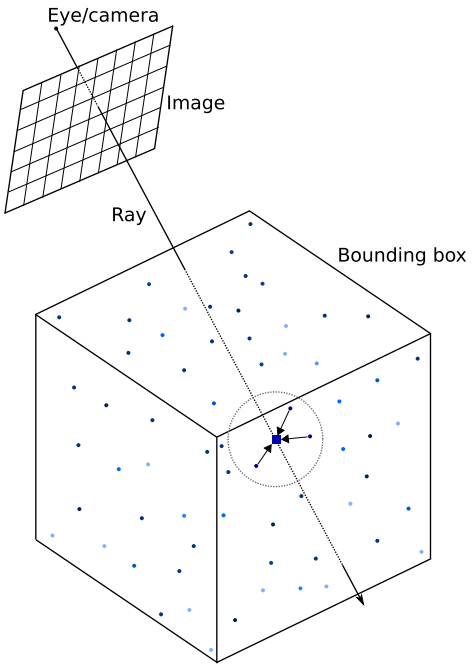


FIGURE 2. Overview of implementation. The input is a set of samples, here shown as dots. Rays are cast from the eye/camera, through each pixel, and into the volume. The value of the underlying function is estimated at points along the ray, by interpolating among neighbouring samples. Here, we show one such ray, and one point, where three samples are contributing to the local function value.

Fig. 2 illustrates how our ray casting method is used to generate a single image. A ray is created for each pixel of the output image. The direction of the ray is defined by the camera position and the center of its pixel. For each ray, starting at the camera, we move along the ray, and estimate the value of the underlying continuous distribution f at positions separated by a user-defined distance δ. How the estimation is performed, is covered later in this section. Next, the estimated value is mapped to a color and opacity value. This is done using a user-defined transfer function. Finally, the color and opacity values of all the positions along a ray are used to evaluate the volume rendering integral, as described in [33], in order to find the final color of the pixel. Starting at the camera, for each point along the ray, we update the color C_i and opacity A_i using the formulae:

$$C_i = C_{i-1} + (1 - A_{i-1})c_i \tag{3}$$

$$A_i = A_{i-1} + (1 - A_{i-1})a_i, \tag{4}$$

where $c_i = c_i(f_e(x_i))$ and $a_i = a_i(f_e(x_i))$ are the color and opacity contribution of the i'th point, which are functions of the estimated value $f_e(x_i)$ of the underlying function at the position of the i'th point x_i, and C_i and A_i are the color and opacity after processing the i'th point. The initial values of the color and opacity are $C_0 = A_0 = 0$. The final color of the pixel in the rendered image is $C_N \cdot A_N$, where N is the number of positions along the ray at which the underlying function is estimated.

1) FILTERING

Before the ray casting starts, the data is filtered using a simplified version of the filtering proposed by Ljung *et al.* [34]. For the X-ray data, low intensity values are often noise (or indistinguishable from noise), making them less interesting from a physics perspective. Therefore, a transfer function that suppresses the samples with low values will typically be used. We can achieve the same effect by simply discarding all samples whose value lies below a user defined threshold when the data is loaded, while at the same time treating regions void of samples as transparent. This will not affect image quality, but can dramatically reduce the size of the input data set, and thereby increase performance. This filtering does not require any changes to our rendering algorithm, since it makes no assumptions about structure or connectivity of the input data points.

2) INTERPOLATION

To estimate the value of the underlying function $f(x)$ at point x, we use IDW interpolation. Rather than using all the samples of the entire dataset for each point, we only use those samples closer to the interpolation point than a user specified search radius r_s. How these samples are found is described later in this section. The user-specified matrix A specifies the equidistance ellipsoid for anisotropic distance calculation. Regions with no samples are treated as transparent, *i.e.,* as if the intensity is zero.

3) NEIGHBOR SEARCH

An accelerating data structure is used to greatly speed up the search for neighboring samples during interpolation. Many data structures have been proposed for this problem [35], [36]. We have chosen to use an octree [37] due to its conceptual simplicity, ease of implementation, intuitive and predictable structure, and good performance. In an octree, each node in the tree corresponds to a cube, and the children of a node are the eight octants of the cube. The root node of our octree is the bounding box of all the samples. Leaf nodes contain those samples that lie in their corresponding cube, and may be empty if no such samples exist. In practice, we store the samples in a separate array, and the leaf nodes contains pointers to the samples.

During search, we want to find all samples within a sphere[2] of radius r_s centered at the search point. At each node, we find the child nodes intersecting the bounding box of the sphere, and search those nodes recursively. Thus the search returns all the leaf nodes of the tree where the intersection between the node and the bounding box of the search sphere is nonempty. Finally, all the samples of each of these leaf node must be checked to see if they fall within the search sphere.

The last step can be made faster by reducing the size of leaf nodes. There is, however a trade-off, as reducing the

[2]When anisotropic distances are used, coordinates can be transformed such that the region to search remains a sphere in the search space.

size of the leaf nodes will increase the depth of the tree, making it more costly to find the leaf nodes in the first place, and also increasing memory overhead. In our implementation, we set the maximum tree height to the empirically chosen

$$h_{\max} = \lfloor \log_2(R/r_s) \rfloor + 2, \quad (5)$$

where R is the smallest dimension of the bounding box of all the samples. For a cubic bounding box, the volume occupied by a leaf node will then be from $(r_s/4)^3$ to $(r_s/2)^3$. To avoid unnecessarily deep trees in sparse regions, we allow leaf nodes to contain up to 8 samples.

A. OPTIMIZATIONS

To improve performance, we have adapted two common optimizations techniques for volume ray casting [2].

First, we use early ray termination. Rather than estimating the value at all n points along the ray, we stop when A_i becomes sufficiently close to 1, which indicates that the volume between the camera and the current position is completely opaque. Hence, the value at further points will not contribute significantly to the color of the pixel.

Second, the filtering described above leaves large regions of the volume empty. Ideally, no search should be performed in such regions, since they are treated as transparent. This could be done by taking advantage of the acceleration data structure. If an empty leaf node was encountered, one could simply jump to the end of it. However, due to the way we store samples in the tree, this might lead to incorrect results, as illustrated in Fig. 3. While this problem can be resolved by organizing the tree differently, we have instead developed a novel empty-space skipping algorithm. If the result of a search is empty, we increase the step size, the distance between positions at which the value of the volume is sampled, by a factor. After each subsequent empty search the step size is increased, until a threshold is reached. When a non-empty search is encountered, we move back to the previous point, reset the step size to its default value, and proceed.

The optimal values for the threshold and factor depend, in the same way as the step size itself, upon the dataset. Care must be taken to avoid setting the threshold to high, as the resulting coarseness of the sampling might miss small structures in the dataset. Currently, these parameter values must be determined through trial and error.

B. PARALLELIZED CPU IMPLEMENTATION

Since all the rays can be processed independently, ray casting is regarded as an embarrassingly parallel problem. We have implemented a parallel version of our method. Rather than distributing the rays evenly between the threads, which might cause problems with load balancing as some rays are more work-intensive than others, we have adopted a work queue and thread pool approach, which is a technique we have used successfully in other contexts [38].

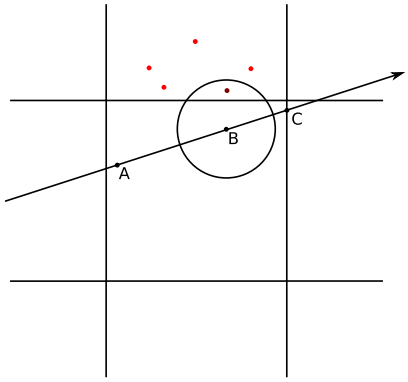


FIGURE 3. Illustration of why skipping a empty node might cause incorrect results. When A is reached and the empty node detected, we could skip it by jumping to C. This would lead to incorrect results, as B would be treated as transparent.

C. GPU IMPLEMENTATION

Ray casting is a compute-intensive and highly parallel task, and is therefore ideally suited for GPUs. Furthermore, the task of moving the finalized image to the GPU for display is made superfluous when the entire computation is done on the GPU itself.

We have created an implementation of our visualization method where the ray casting is performed on the GPU, using Nvidia's CUDA platform [39]. We have implemented a kernel that processes a single ray. Multiple instances of this kernel are then run on the GPU in parallel, with one thread for each ray. The kernel implements the algorithm described above.

Since CUDA allows using a subset of the C programing language, we only needed to make a few modifications to our code in order to port it from the CPU to the GPU. Each of these modifications will be described in the following paragraphs.

1) REMOVING RECURSION

Older GPUs do not support recursion [39]. Since supporting a broad range of hardware was a priority for us, we replaced the recursive octree range search algorithm by iteration in combination with a manually managed stack. Finding all the samples within the search sphere is done by initially pushing the root node of the tree onto the stack. At each step of the iteration, one node is popped of the stack. If it is a leaf node, its samples are added to those returned. Otherwise, those child nodes of the popped node intersecting with the bounding box of the search sphere are pushed onto the stack. The loop runs until the stack is empty. The depth-first nature of the algorithm ensures that the stack size is bounded by $O(h_{\max})$ where $h_{\max}$ is the maximum depth of the tree.

In an effort to reduce the memory footprint of the stack, we used the following compaction technique: When nodes are pushed on the stack, we always push all the children of a node that intersects the search box at the same time. Therefore, rather than pushing a pointer to each of these child nodes, we can push a pointer to the parent node, along with a description of which of its child nodes that are pushed. This idea is

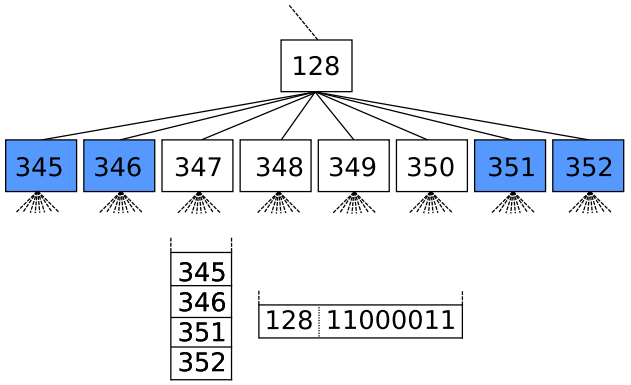


FIGURE 4. Stack optimization. On the top, a portion of a octree is shown. The shaded child nodes are to be pushed onto the stack. In the bottom left is the original stack, and in the bottom right is the optimized stack, after the shaded nodes have been pushed. The stack grows downwards.

illustrated in Fig. 4. The description can be encoded in one byte, where each bit indicates whether a child is pushed or not. By reducing the size of the node pointer to 24 bits, we can combine the parent node pointer, and child descriptor in a single 4 byte integer.

2) MEMORY OPTIMIZATIONS

To achieve optimal performance, memory operations must take the GPU's explicit memory hierarchy into account [39], [40]. *Texture memory* is a logical memory space which physically resides in the GPUs device memory (RAM). This memory is designed to store textures for graphics applications, and is therefore optimized for 2D spatial locality and streaming fetches. Furthermore, it is typically cached. We stored the input samples in this memory, which can increase performance for a number of reasons.

Firstly, all the samples of a leaf node will be accessed sequentially, to check if they actually lie within the search sphere, and if that is the case, be used in the interpolation. As mentioned, texture memory is optimized for such streaming access patterns. We therefore sort the samples, so that all samples belonging to the same leaf node are placed together, before transferring the data to the GPU. Secondly, special access patterns are required to get good performance for global memory. Since these access patterns cannot always be achieved in our case, texture memory might give better performance. Finally, as opposed to global memory, texture memory has its own cache, also on older GPUs without L2 and L1 caches. Since a thread might access the same samples at consecutive points along the ray, or neighbouring threads might access the same samples, this can also increase performance.

3) MULTI-GPU AND LOAD BALANCING

Multiple GPUs can be used together to increase performance [41]. The embarrassingly parallel nature of ray casting makes this easy, and allows us to further speed up the rendering. Each GPU can simply be assigned a fraction of the rays/pixels, and process them independently. Achieving good load balancing, so that all the GPUs finish at the same time, is challenging, however, for two reasons. Firstly, the amount of work per ray/pixel is not constant. Hence, distributing the number of rays/pixels evenly will not cause work to be distributed evenly. Secondly, different GPUs with different performance might be used together.

To address these issues, we have developed a load balancing scheme that uses two techniques. The first is based on the realization that our application typically will be used to generate a large number of frames, where each frame is quite similar to the previous, because the camera will often be moved smoothly around the object. The relative performance of each GPU for one frame can therefore be used to decide how to distribute work for the next frame. The second technique uses the length of a ray, that is, the distance the ray intersects the bounding box of all the samples, as a proxy for the amount of work required to process it. While clearly inaccurate, this is a better assumption than that of uniform ray/pixel work amount. The ray lengths can be computed fairly cheaply on the CPU prior to work distribution.

V. RESULTS AND DISCUSSION

Two carefully chosen and qualitatively different experimental X-ray datasets were used to test our method.

Dataset A was obtained for a $PbTiO_3$ thin film on a $SrTiO_3$ substrate [42], measured at the Swiss-Norwegian Beamline (BM01A) of the European Synchrotron Radiation Facility using a six-circle κ diffractometer, a 1024 by 1024 pixel CCD detector, and an X-ray wavelength of 0.097 nm. It consists of 5.6 million datapoints and was zoomed in on a single feature known as a crystal truncation rod [43]. A 3D rendering is provided in Fig. 5a. The central, green appearing, crystal truncation rod displays clear intensity variations consistent with the crystalline structure and thickness of the film. The surrounding magenta torus of lower intensity arises from, and thus contains information about, ferroelectric domain structures in the $PbTiO_3$ film [44].

Dataset B is from a single crystal of *diaquabis(salicylato) copper(II)* [45], [46], consists of 243 million samples, and was measured over a much larger region of **Q**-space than dataset A. The data was measured using a rotating anode X-ray source emitting Cu Kα radiation (wavelength 0.154 nm), a four-circle diffractometer and a Dectris Pilatus 1M pixelated detector [47]. A 3D rendering can be found in Fig. 5c, clearly demonstrating the presence of both regularly spaced sharp Bragg peaks consistent with a single-crystalline structure, and the presence of modulated lines of diffuse intensity in certain directions coinciding with high-symmetry directions in the sample.[3]

For both datasets a filtering threshold was applied to remove samples from low-intensity regions where no mea-

[3]This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors. This includes two mp4 format movie clips showing 3D renderings of the two datasets described here, generated with our method. This material is 36.9 MB in size.

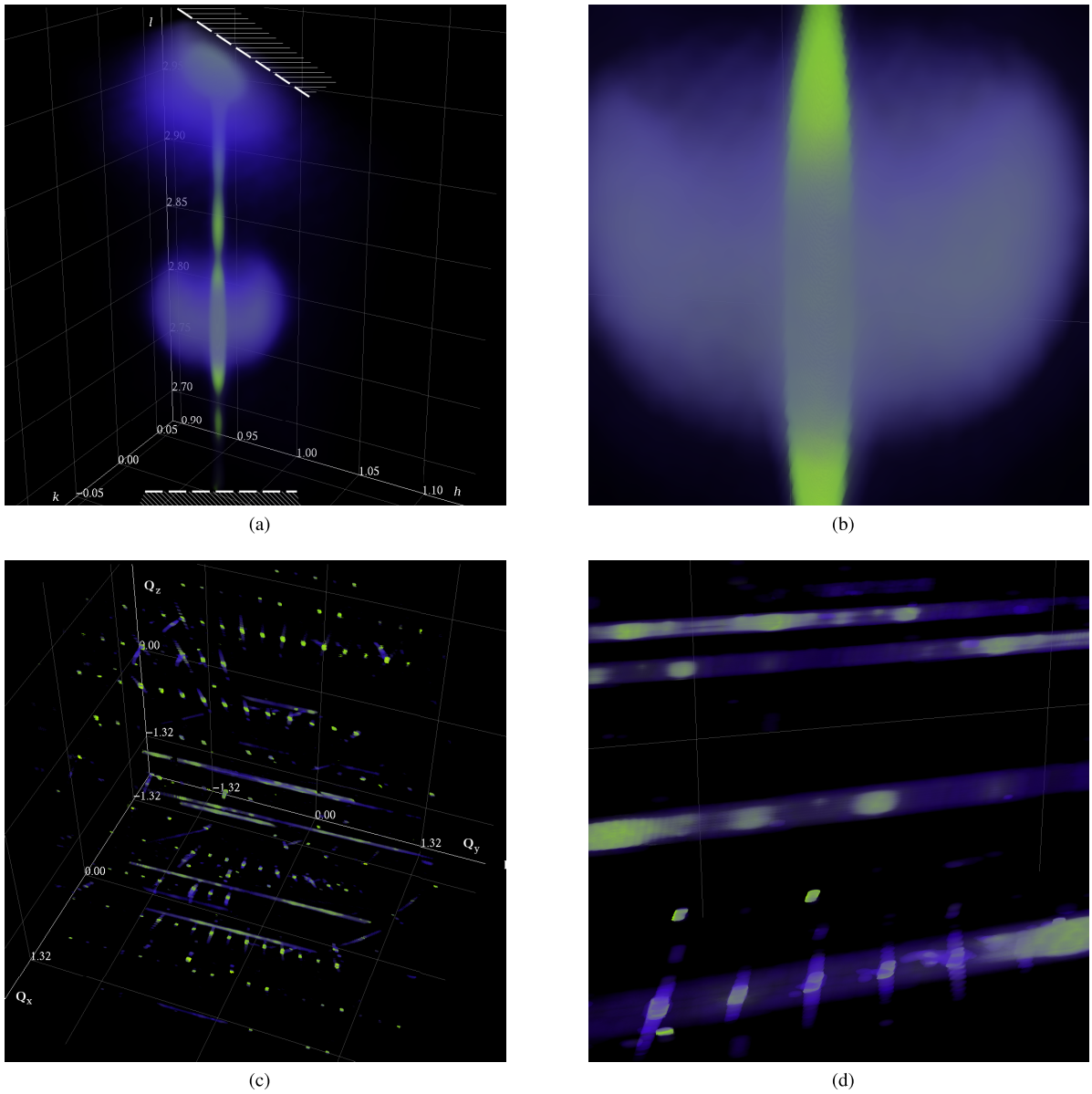


FIGURE 5. **Visualizations of the two datasets generated with our method. (a, b) Renderings of dataset A showing in normalized dimensionless units the 10*l* crystal truncation rod of oscillating intensity and the weaker torus caused by the ferroelectric domain structure of the $PbTiO_3$ film. The crossed out patches in (a) indicate regions without measurements. (c, d) Renderings of dataset B, in units of $Å^{-1}$, showing both sharp Bragg peaks and diffuse line features, oriented along high-symmetry directions in the material.**

sureable diffracted intensity was detected. This removed 36.45% and 99.98% of the samples for dataset A and B, respectively.

For the performance testing, we used a 4-core, hyperthreaded 3.5 GHz Intel i7 3770K CPU, and three different Nvidia Tesla GPUs. Hardware details for the GPUs are shown in Table 1. GCC version 4.6.3 and the Nvidia CUDA Toolkit version 5.0 with all optimizations enabled were used for compilation. On the CPU, we used twice as many software threads as physical cores, to take advantage of hyperthreading. Our code can be compiled to use either single or double precision floating point numbers; unless otherwise noted, single precision was used. This is sufficient in our case due to the limited dynamic range of the X-ray data, but other applications might require double precision. Performance was measured by rendering four representative 1024 $\times$ 1024 images from the two datasets, these are shown in Fig. 5. On the GPU, the rendering time reported here includes the time required to transfer the final image back to the host (since neither the K20 nor the C2070 has video output), but not the time required to transfer the samples or tree data structure to the GPU, since this only has be done once, and the cost typically will be amortized by rendering a high number of frames. For the same reason, the time

required to load and filter the data and to build the tree is not included.

TABLE 1. Hardware details of Nvidia GPUs used.

GPU	C1060	C2070	K20
Streaming multiprocessors	30	14	13
CUDA cores	240	448	2496
Clock rate (GHz)	1.3	1.15	0.71
Peak GFlop/s (single prec.)	933	1030	3520
Peak GFlop/s (double prec.)	78	515	1170
Memory (GB)	4	6	5
Global memory bandwith (GB/s)	102	144	208

A. BENEFITS OF ANISOTROPIC INTERPOLATION

To demonstrate how anisotropic interpolation can improve image quality, we generated several renderings of dataset B with various settings for interpolation. The settings are given in Table 2 and the resulting figures shown in Figure 6.

To render Fig. 6a, a small search radius r_s and isotropic interpolation were used. The resulting image has quite poor quality. In particular, the rightmost part of the rods appear discontinuous, as the search radius is to small to span the gaps between adjacent samples. This problem can be mitigated by simply increasing the search radius, as was done to render Fig. 6b. While the artifacts of Fig. 6a are removed, the increased radius increases blurring. In Fig. 6c a small radius was combined with anisotropic distance measurement. The anisotropy used compacts distances along the rods. The discontinuity artifacts are almost completely removed, without introducing the same amount of blurring, resulting in improved image quality.

In the case of dataset B, the underlying function $f(x)$ changes much more slowly along the rods than in directions orthogonal to them. It is therefore unsurprising that anisotropic distance can be used to improve the image quality, as explained in Section III-B. Anisotropic interpolation has, however, the disadvantage that it relies upon a priori knowledge about the underlying distribution. Furthermore, in regions where the underlying function changes uniformly in all directions, it will introduce artifacts. This can be seen in Fig. 6c, where the single points in the top and bottom parts of the image appear stretched, relative to the other figures.

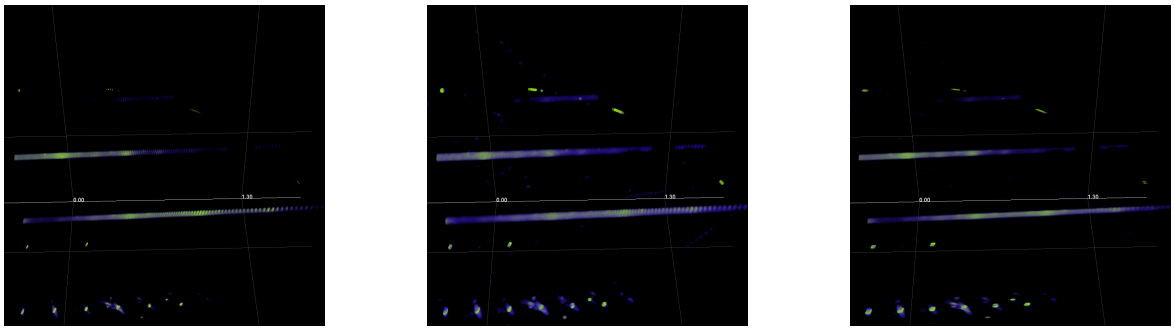
TABLE 2. Settings used in Fig. 6. I_3 is the 3 × 3 identity matrix.

Figure	Search radius r_s	Anisotropy
6a	0.005	$A = I_3$
6b	0.01	$A = I_3$
6c	0.005	$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

B. PERFORMANCE

Rendering times for the different images on the CPU and GPUs varied considerably. While interactive frame rates were achieved for Fig. 5c and 5d (on the fastest GPU), rendering the other images were orders of magnitude slower, as shown in Fig. 7. To explain this variance, recall that in Fig. 5b, and to a lesser extent in Fig. 5a, all rays pass through large regions of high sample density. This is significantly more computationally demanding than if they had been passing through empty space, as in Fig. 5c and 5d. In empty space, the range search will return faster, no interpolation is required, and the empty-space skipping optimization can be employed. It should also be noted that the settings used to achieve satisfactory image quality depends heavily upon the properties of the dataset, as well as the viewing angle. These different parameter values have a great impact on the rendering time.

While real-time or interactive frame rates always are desired, the primary goal of our method was high quality images, not speed. It should be noted that adjusting the settings easily allows for a trade-off between quality and performance. This makes it possible to identify promising viewing angles at interactive or real-time rates, and then render high quality images offline.



(a) $r_s = 0.005$; isotropic. (b) $r_s = 0.01$; isotropic. (c) $r_s = 0.005$; anisotropic.

FIGURE 6. Examples of the effects of varying search radius r_s and anisotropy matrix, with settings given in table 2. (a) Small r_s and isotropic interpolation, resulting in the diffuse lines appearing to be broken where there is a lack of samples . (b) Increased r_s compared to (a), making the diffuse lines appear continuous while slightly decreasing the resolution. (c) r_s as (a), but with anisotropic interpolation favoring interpolation along the horizontal direction in the figure, effectively hiding the effects of insufficient sampling while retaining good resolution.

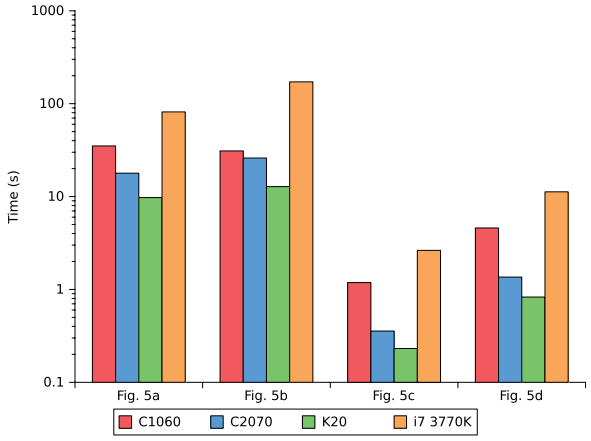


FIGURE 7. Rendering time for the images in Fig. 5, for the parallel CPU version on a 4-core, hyperthreaded Intel i7 3770K and the GPU versions on three different Nvidia Tesla GPUs. Note the logarithmic scale.

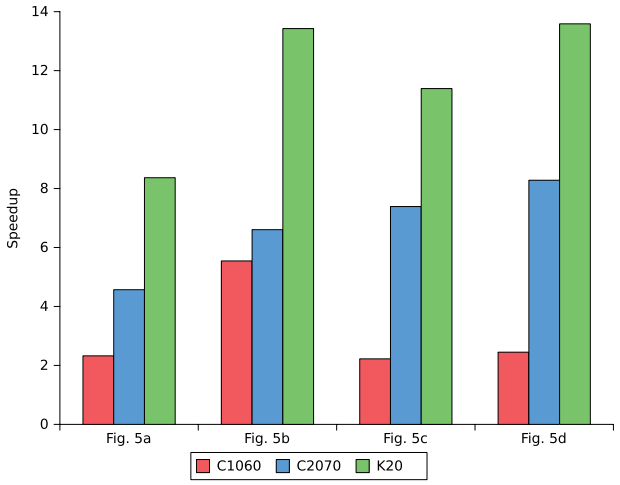


FIGURE 8. Speedups of the GPU version on three different Nvidia Tesla GPUs over the parallel CPU version on a 4-core, hyperthreaded Intel i3770K for the different images in Fig. 5.

The speedups on the GPUs compared to the CPU are shown in Fig. 8. For the K20 we saw speedups between 8 and 14, while the older GPUs achieved speedups between 2 and 9. The significant performance increase on the GPUs is expected, due to the highly parallel and computationally demanding nature of our ray casting algorithm. We saw different speedups for the different images depending upon how well suited their processing requirements were for the different GPU architectures.

1) SINGLE AND DOUBLE PRECISION

The results of comparing single and double precision performance for the different images and processing units can be found in Fig. 9. Here, the texture optimization described in Section IV-C.2 was disabled, due to lack of support for texels consisting of four doubles. GPUs typically perform significantly fewer double precision operations per second compared to single precision. This difference was more pronounced on older GPUs, as evidenced by the poor results for the older C1060 on Fig. 5a and 5b, where single precision was 8.3 and 4.6 times faster than double, respectively. However, as we can see, the situation has improved, for the newer C2070 and K20, single precision was between 1.7 and 3 times faster than double.

To explain differences in rendering performance between the images, we must again consider their different processing requirements. In Fig. 5a and 5b, more rays pass through high density, medium intensity regions compared to Fig. 5c and 5d. Hence, more interpolation must be done to render these images, while, for Fig. 5c and 5d, most of the time is spent searching. Interpolation is more computationally intensive, involving expensive floating point operations. Searching is comparatively simple; the only floating point operations performed are comparisons. Switching from single to double precision will therefore lead to a larger performance hit for the images from dataset A.

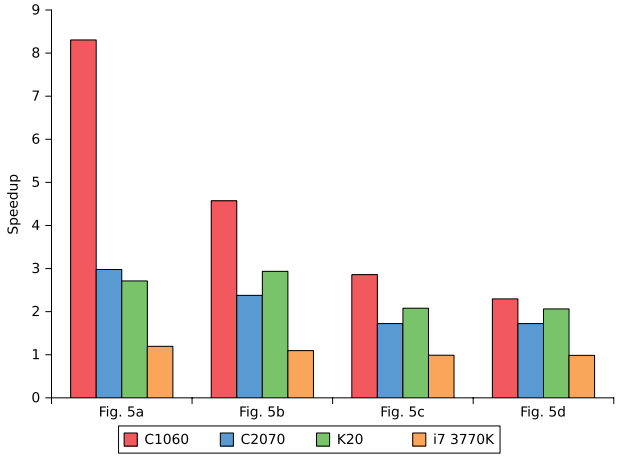


FIGURE 9. Speedup of single over double precision for the images in Fig. 5, for the parallel CPU version on a 4-core, hyperthreaded Intel i7 3770K and the GPU versions on three different Nvidia Tesla GPUs. Lower speedup values implies better double precision performance.

2) TEXTURE MEMORY

Fig. 10 shows the speedup achieved when the texture optimization described in Section IV-C.2 was applied. The effect varied, for Figs. 5a and 5b, the K20 and C1060 got speedups of between 1.2 and 1.6, while the performance for the C2070 degraded slightly. For Figs. 5c and 5d, we saw no significant speedups.

The difference between the images can again be explained by the fact that for Figs. 5a and 5b, the rays pass through extended regions of high sample density and medium diffracted intensity. Hence, significantly more samples are read, compared to Figs. 5c and 5d. Since the texture optimization improves the performance of reading samples, higher speedups were obtained for these images.

To explain the lack of speedup on the C2070, we must take a closer look at the architecture of the different GPUs. The C1060 does not have caches, so manual caching, either

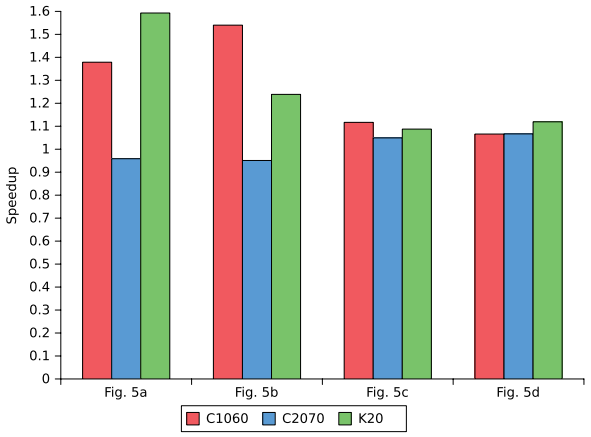


FIGURE 10. Speedups of texture optimization for the images in Fig. 5 for the GPU version on three different Nvidia Tesla GPUs.

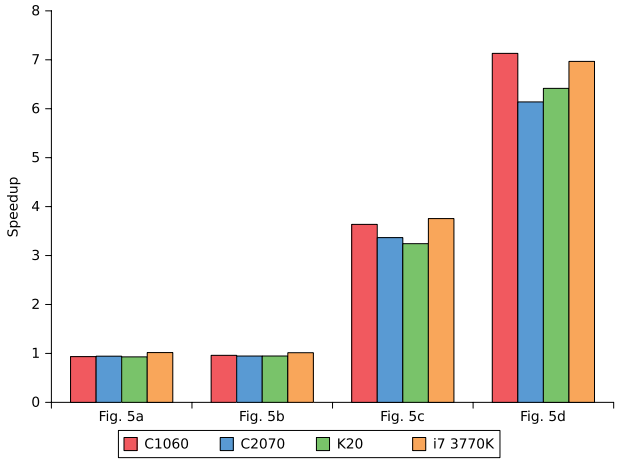


FIGURE 11. Speedups obtained by empty-space skipping for the images in Fig. 5, for the parallel CPU version on a 4-core, hyperthreaded Intel i7 3770K and the GPU versions on three different Nvidia Tesla GPUs.

by using shared memory, or, as we do, texture memory, can therefore increase performance. The newer C2070 has both L2 and L1 caches. Using these yields slightly better performance than using the texture cache manually, and is the cause of the lack of speedup in this case. This can be verified by disabling the L1 cache at compile time. If this is done, using texture memory will improve performance. The newest of the GPUs, the K20, also has L1 and L2 caches, and hence it might seem surprising that we did not see the same results as for the C2070. However, on the K20, loads from global memory are not cached in L1 cache, which is used for local memory accesses only [48]. Hence, using texture memory pays off. The GK110 architecture, on which the K20 is based, also makes it possible to use the cache of the texture pipeline without having to bind the memory to a texture beforehand.

3) EMPTY-SPACE SKIPPING

In Fig. 11, the effect of the empty-space skipping optimization is shown. We saw a significant speedup of 3.2–3.8 for Fig. 5c and 6.1–7.1 for Fig. 5d, but little or no effect on the images from dataset A. This is as expected, since dataset B is more sparse, and has more empty space than dataset A. The difference between Fig. 5c and Fig. 5d is caused by the different settings that are used. In order to capture finer details, the original step size used to render Fig. 5d is smaller than that used to render Fig. 5c, hence the potential for speedup is greater in this case.

4) MULTI-GPU SYSTEMS AND LOAD BALANCING

Our multi-GPU load balancer assumes that multiple images will be rendered, since it uses the performance of one image to divide work for the next, as explained in Section IV-C.3. Therefore, to tests its performance, we rendered 4 pairs of images. Each pair consisted of one of the images used thus far (as shown in Fig. 5) as well as a slightly zoomed out version of the same image. For each set, we first rendered the zoomed out image, and then used the results to divide work for the next image. The reported results are for the last rendering only.

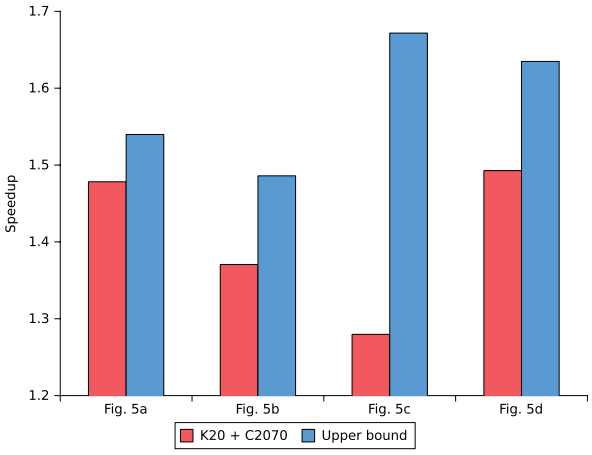


FIGURE 12. Speedup of the multi-GPU version on the combination of Nvidia Tesla C2070 and K20 versus the single-GPU version on the K20 alone. Theoretical upper bounds are also indicated.

Fig. 12 shows the speedups obtained when a C2070 was used together with a K20, compared to just a single K20. The figure also shows a theoretical upper bound on the speedup. This upper bound was computed by assuming that if a fraction β of the work is assigned to the K20, and $1-\beta$ to the C2070, the rendering time will be $\max(\beta T_{\mathrm{K20}}, (1-\beta)T_{\mathrm{C2070}})$, where T_{K20} and T_{C2070} are the rendering times measured individually on the K20 and C2070, respectively. The theoretical lower bound on joint rendering time can then be found solving the equation $\beta T_{K20} = (1-\beta)T_{C2070}$ for β. The theoretical upper bound on speedup can then trivially be computed.

The combination of C2070 and K20 achieved an average speedup of 1.4, while the average theoretical upper bound is 1.58. The cause of this discrepancy is that the amount of work for a thread is not known in advance. Hence, using thread count as a proxy for amount of work, even when ray lengths are adjusted for, leads to inaccurate estimates. The poor speedup results for Fig. 5c was caused by the fact that the rendering time was so short that the time to transfer the image back to the host became significant. The theoretical

upper bound does not take this overhead into account, and was therefore too high in this case.

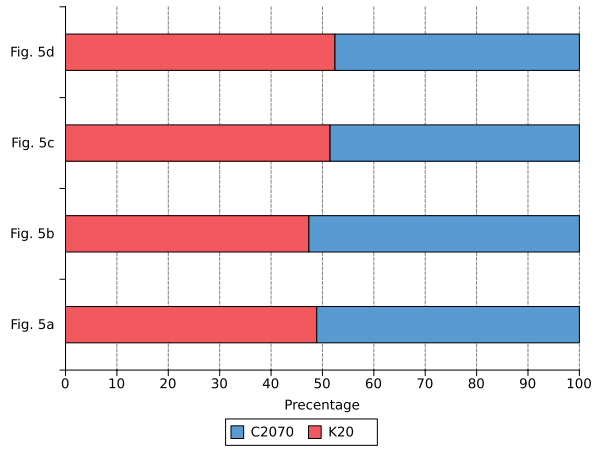


FIGURE 13. Fraction of total compute time on each GPU for the multi-GPU version with two different Nvidia Tesla GPUs.

Fig. 13 shows how well the load balancing scheme was able to divide work evenly between the GPUs. The figure shows the fraction of compute time spent on each GPU. Ideally, the GPUs should spend the same amount of time, resulting in a 50/50 division. As we can see, for all cases, the division was fairly close to the ideal.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have developed a method based on volume ray casting for visualizing volumetric scattered point data. We have applied the method to two qualitatively different experimental X-ray diffraction datasets of highly crystalline materials containing disorder, showing that high-quality visualization of intensity distributions in **Q**-space are highly useful for extracting information on complex nanostructures and disorder from X-ray diffraction experiments. This is one example of the growing importance of visualization.

The introduced method differs from traditional ray casting algorithms by not using voxels, but rather operating directly on the scattered point data. The method finds the value of the scalar field to be visualized at positions along the rays by interpolation using nearby samples. We use an octree to efficiently find the close-by samples. We implemented standard ray casting optimization techniques such as early ray termination and empty-space skipping. A novel, acceleration data structure agnostic algorithm for performing empty-space skipping, suitable for situations where voxels or similar representations are not used, has been developed. We have shown that in situations where the average absolute value of the directional derivatives depends strongly on the directions, image quality can be improved by using anisotropic interpolation to find values at points along the rays. Versions for multicore CPUs, GPUs and multi-GPU systems have been implemented.

Our implementations were tested using actual X-ray diffraction data, consisting of up to 120 M data points. Our method is capable of producing images of good quality. The rendering time varies significantly, between 0.2 s and 12.7 s, (Nvidia Tesla K20), depending upon dataset, and settings used. The GPU implementation (on Nvidia Tesla K20) achieves a speedup between 8 and 14 for different images, compared to the multithreaded CPU version (Intel i7-3770K).

In future research, one may investigate how performance can be improved with further optimizations. Devising methods to automatically determine optimal parameter settings is also a possible direction of future research. It would be interesting to look at how our method compares to a different algorithm such as splatting. As experimental and simulated datasets become larger, one idea would be to also look at the tradeoffs of data compression Aqrawi *et al.*, [49]. Without doubt, future work will aim at further developing real-time interactive multi-dimensional visualization tools for interactive analysis and visualization of vast data sets.

ACKNOWLEDGMENT

The authors would like to thank Nvidia's CUDA Research Center Program and NTNU for hardware donations, Thomas Tybell for providing the $PbTiO_3$ sample, Frode Mo for facilitating the experiments at the Swiss-Norwegian Beamlines, and Emil J. Samuelsen for providing the diaquabis(salicylato)copper(II) sample.

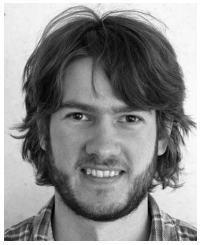
REFERENCES

[1] C. Hansen and C. R. Johnson, *The Visualization Handbook*. Amsterdam, The Netherlands: Elsevier, 2005.
[2] M. Levoy, "Efficient ray tracing of volume data," *ACM Trans. Graph.*, vol. 9, no. 3, pp. 245–261, Jul. 1990.
[3] M. P. Garrity, "Raytracing irregular volume data," *ACM SIGGRAPH Comput. Graph.*, vol. 24, no. 5, pp. 35–40, Nov. 1990.
[4] L. Westover, "SPLATTING: A parallel, feed-forward volume rendering algorithm," Ph.D. dissertation, Dept. Comput. Sci., Univ. North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1991.
[5] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3D texture hardware," *Radiat. Oncol.*, vol. 61, pp. 1–6, Jun. 1994.
[6] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proc. Symp. Vol. Visualizat.*, 1994, pp. 91–98.
[7] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A simple and flexible volume rendering framework for graphics-hardware-based raycasting," in *Proc. 4th Eurograph. IEEE VGTC Conf. Vol. Graph.*, Jun. 2005, pp. 187–241.
[8] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering," in *Proc. Symp. Interact. 3D Graph. Games*, Feb. 2009, pp. 15–22.
[9] L. Marsalek, A. Hauber, and P. Slusallek, "High-speed volume ray casting with CUDA," in *Proc. IEEE Symp. Interact. RT*, Aug. 2008, p. 185.
[10] H. Ludvigsen and A. C. Elster, "Real-time ray tracing using nvidia optix," in *Eurographics Short Papers*, H. P. A. Lensch and S. Seipel, Eds. New York, NY, USA: Wiley, 2010, pp. 65–68.
[11] G. Nielson, "Scattered data modeling," *IEEE Comput. Graph. Appl.*, vol. 13, no. 1, pp. 60–70, Jan. 1993.
[12] I. Amidror, "Scattered data interpolation methods for electronic imaging systems: A survey," *J. Electron. Imaging*, vol. 11, no. 2, pp. 157–176, 2002.
[13] J. Wihelms, J. Challinger, N. Alper, S. Ramamoorthy, and A. Vaziri, "Direct volume rendering of curvilinear volumes," *SIGGRAPH Comput. Graph.*, vol. 24, no. 5, pp. 41–47, Nov. 1990.

[14] P. Navratil, J. Johnson, and V. Bromm, "Visualization of cosmological particle-based datasets," *IEEE Trans. Visualizat. Comput. Graph.*, vol. 13, no. 6, pp. 1712–1718, Nov./Dec. 2007.
[15] R. Fraedrich, S. Auer, and R. Westermann, "Efficient high-quality volume rendering of SPH data," *IEEE Trans. Visualizat. Comput. Graph.*, vol. 16, no. 6, pp. 1533–1540, Nov./Dec. 2010.
[16] S. W. Park, L. Linsen, O. Kreylos, and J. D. Owens, "A framework for real-time volume visualization of streaming scattered data," in *Proc. Workshop Vis., Model. Visualizat.*, 2005, pp. 225–232.
[17] M. Hopf and T. Ertl, "Hierarchical splatting of scattered data," in *Proc. 14th IEEE Visualizat.*, 2003, pp. 443–440.
[18] M. Chen, "Combining point clouds and volume objects in volume scene graphs," in *Proc. 4th Int. Workshop Vol. Graph.*, Jun. 2005, pp. 127–235.
[19] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl, "Interactively visualizing procedurally encoded scalar fields," in *Proc. Eurograph. IEEE Symp. Visualizat.*, May 2004, pp. 35–44.
[20] C. Ledergerber, G. Guennebaud, M. Meyer, M. Bacher, and H. Pfister, "Volume MLS ray casting," *IEEE Trans. Visualizat. Comput. Graph.*, vol. 14, no. 6, pp. 1372–1379, Nov./Dec. 2008.
[21] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[22] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, no. 1, pp. 1–33, 2010.
[23] Ø. E. Krog and A. C. Elster, "Fast GPU-based fluid simulations using SPH," in *Applied Parallel and Scientific Computing* (Lecture Notes in Computer Science), K. Jonasson, Ed. Berlin, Germany: Springer-Verlag, 2012, pp. 98–109.
[24] L. Dematt and D. Prandi, "GPU computing for systems biology," *Briefings Bioinf.*, vol. 11, no. 3, pp. 323–333, 2010.
[25] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Med. Phys.*, vol. 38, no. 5, pp. 2685–2697, 2011.
[26] E. Smistad, A. C. Elster, and F. Lindseth, "Real-time gradient vector flow on GPUs using OpenCL," in *Journal of Real-Time Image Processing*. New York, NY, USA: Springer-Verlag, Jun. 2012, pp. 1–8.
[27] E. Smistad, A. C. Elster, and F. Lindseth, "Fast surface extraction and visualization of medical images using OpenCL and GPUs," in *Proc. Joint Workshop High Perform. Distrib. Comput. Med. Imaging*, 2011, pp. 1–10.
[28] D. Shepard, "A two-dimensional interpolation function for irregularly-spaced data," in *Proc. 23rd ACM Nat. Conf.*, 1968, pp. 517–524.
[29] M. Tomczak, "Spatial interpolation and its uncertainty using automated anisotropic inverse distance weighting (IDW)—Cross-validation/jackknife approach," *J. Geograph. Inf. Decision Anal.*, vol. 2, no. 2, pp. 18–30, 1998.
[30] A. Adamson and M. Alexa, "Anisotropic point set surfaces," *Comput. Graph. Forum*, vol. 25, no. 4, pp. 717–724, 2006.
[31] J. Als-Nielsen and D. McMorrow, *Elements of Modern X-ray Physics*, 2nd ed. New York, NY, USA: Wiley, 2011.
[32] S. O. Mariager, C. B. Sørensen, M. Aagesen, J. Nygård, R. Feidenhans'l, and P. R. Willmott, "Facet structure of GaAs nanowires grown by molecular beam epitaxy," *Appl. Phys. Lett.*, vol. 91, no. 8, pp. 083106-1–083106-3, Aug. 2007.
[33] N. Max, "Optical models for direct volume rendering," *IEEE Trans. Visualizat. Comput. Graph.*, vol. 1, no. 2, pp. 99–108, Jun. 1995.
[34] P. Ljung, C. Lundstrom, A. Ynnerman, and K. Museth, "Transfer function based adaptive decompression for volume rendering of large medical data sets," in *Proc. IEEE Symp. Vol. Visualizat. Graph.*, Oct. 2004, pp. 25–32.
[35] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *ACM Comput. Surv.*, vol. 11, no. 4, pp. 397–409, Dec. 1979.
[36] H. Samet, *Foundations of Multidimensional and Metic Data Structures*. San Mateo, CA, USA: Morgan Kaufmann, 2006.
[37] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Inf.*, vol. 4, no. 1, pp. 1–9, 1974.
[38] T. L. Falch, J. B. Fløystad, D. W. Breiby, and A. C. Elster, "Optimization and parallelization of ptychography reconstruction code," in *Proc. 25th Norwegian Inf. Conf.*, Jan. 2012, pp. 117–128.
[39] NVIDIA. (2012, Apr. 30). *NVIDIA CUDA C Programing Guide*, Santa Clara, CA, USA [Online]. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation
[40] NVIDIA. (2012, May 4). *CUDA C Best Practices Guide* [Online]. Available: http://developer.nvidia.com/nvidia-gpu-computing-documentation
[41] D. G. Spampinato, A. C. Elster, and T. Natvig, "Modelling multi-GPU systems" in *Advances in Parallel Computing*. Amsterdam, The Netherlands: Ios Press, 2009, pp. 562–569.
[42] R. Takahashi, J. K. Grepstad, T. Tybell, and Y. Matsumoto, "Photochemical switching of ultrathin $PbTiO_3$ films," *Appl. Phys. Lett.*, vol. 92, no. 11, pp. 112901-1–112901-3, Mar. 2008.
[43] I. K. Robinson, "Crystal truncation rods and surface roughness," *Phys. Rev. B*, vol. 33, no. 6, pp. 3830–3836, Mar. 1986.
[44] D. D. Fong, G. B. Stephenson, S. K. Streiffer, J. A. Eastman, O. Auciello, P. H. Fuoss, and C. Thompson, "Ferroelectricity in ultrathin perovskite films," *Science*, vol. 304, no. 5677, pp. 1650–1653, Jun. 2004.
[45] S. Jagner, R. G. Hazell, and K. P. Larsen, "The crystal structure of diaquabis(salicylato)copper(II), $Cu[C_6H_4(OH)COO]_2(H_2O)_2$," *Acta Crystallograph. Sec. B*, vol. 32, no. 2, pp. 548–554, Feb. 1976.
[46] S. Tjotta, E. Samuelsen, and S. Jagner, "Short-range stacking order in the layered material diaquabis (salicylato) copper (II), studied by X-ray and Raman scattering," *J. Phys., Condensed Matter*, vol. 3, no. 20, pp. 3411–3419, May 1991.
[47] P. Kraft, A. Bergamaschi, C. Bronnimann, R. Dinapoli, E. Eikenberry, H. Graafsma, B. Henrich, I. Johnson, M. Kobas, A. Mozzanica, C. Schleputz, and B. Schmitt, "Characterization and calibration of PILATUS detectors," *IEEE Trans. Nuclear Sci.*, vol. 56, no. 3, pp. 758–764, Jun. 2009.
[48] NVIDIA. (2013, Jun. 17). *Tuning CUDA Applications for Kepler*, Santa Clara, CA, USA [Online]. Available: http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf
[49] A. A. Aqrawi and A. C. Elster, "Bandwidth reduction through multithreaded compression of seismic images," in *Proc. IEEE IPDPSW*, Jun. 2011, pp. 1730–1739.

THOMAS L. FALCH received the M.Sc. degree in computer science from the Norwegian University of Science and Technology, Trondheim, Norway, where he is currently pursuing the Ph.D. degree with the Department of Computer Science.

He is currently working on a heterogeneous computing framework for medical image processing and visualization. His current research interests include heterogeneous computing and scientific visualization.



JOSTEIN BØ FLØYSTAD received the M.Sc. degree in physics from the Norwegian University of Science and Technology, Trondheim, Norway, where he is currently pursuing a Ph.D. degree with the Department of Physics.

His current research interests include *in situ* X-ray imaging, ptychography, and other X-ray scattering techniques.

DAG W. BREIBY received the master's and Ph.D. degrees in physics from the Norwegian University of Science and Technology (NTNU), Trondheim, Norway, in 1999 and 2003, respectively.

He was a Post-Doctoral Fellow with Risoe National Laboratories, Denmark, from 2003 to 2007, before joining the Department of Physics at NTNU as Associate Professor in 2007. He has authored or co-authored over 45 journal papers and several book chapters. His current research interests include structure/property relations in functional materials, X-ray physics, and experimental nanotechnology.

ANNE C. ELSTER (S'83–M'95–SM'00) received the B.S. degree in computer systems engineering from the University of Massachusetts, Amherst, MA, USA, in 1985, where she also took several courses in computer science and honors mathematics, and the M.S. and Ph.D. degrees in electrical engineering from Cornell University, Ithaca, NY, USA.

She explored various HPC systems in the late 80s and early 90s. She worked for Schlumberger in Austin from 1994 to 1997, then part-time at the University of Texas at Austin from 1997 to 2000 and founded ACENOR Inc. in 2000. Since January 2001, she has been with the Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway, where she is an Associate Professor. She is the Co-Founder and Co-Director of NTNUs Computational Science and Visualization Program, and holds a Visiting Scientist appointment with the University of Texas at Austin. Her current research interests include high-performance computing in general, with a current focus on heterogeneous and parallel computing. She served on the MPI standards committees (MPI and MPI-2), and has served on several panels and many program committees through the years. She has supervised 50+ master students, published 50+ technical papers, and given 40+ invited talks and tutorials. She is a member of ACM, AGU, IEEE, SIAM, Tekna, and several other technical societies.