Eurographics Symposium on Parallel Graphics and Visualization (2018)
H. Childs, F. Cucchietti (Editors)

Rapid *k*-d Tree Construction for Sparse Volume Data

S. Zellmann[1], J. P. Schulze[2] and U. Lang[1]

[1]Chair of Computer Science, University of Cologne, Germany
[2]Department of Computer Science, UCSD, California, USA

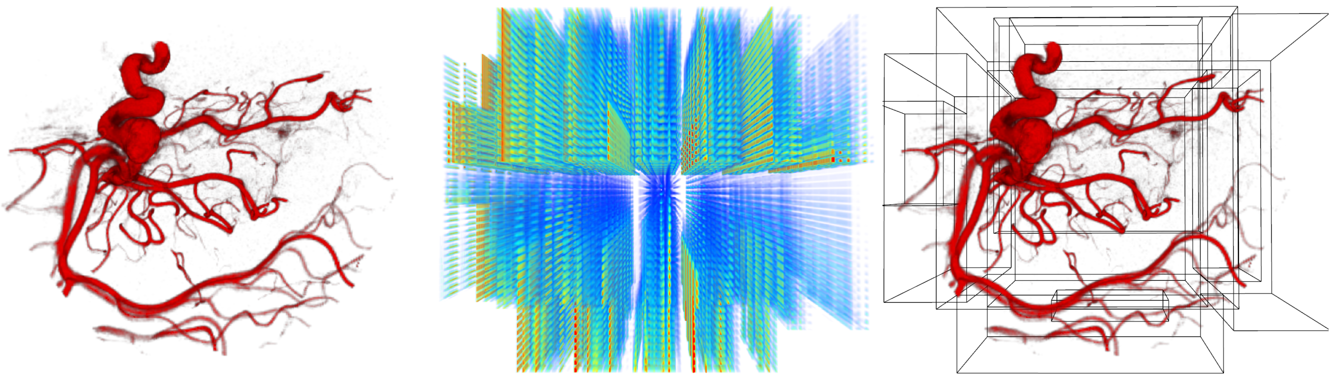

Figure 1: *Stages of the k-d tree construction algorithm. The sparse volume data set depicted on the left hand side of the figure can be efficiently rendered with hierarchical empty-space skipping. Our parallel version of the construction algorithm is based on* partial *summed-volume tables (SVT) shown in the middle that are created for regions of* 32^3 *voxels. Partial SVTs can be efficiently computed in parallel. During the splitting phase of the k-d tree construction algorithm, parallel occupancy queries are performed using the partial SVTs to obtain the k-d tree on the right hand side of the image.*

Abstract

While k-d trees are known to be effective for spatial indexing of sparse 3-D volume data, full reconstruction, e.g. due to changes to the alpha transfer function during rendering, is usually a costly operation with this hierarchical data structure. We pick a serial state of the art implementation that is based on summed-volume tables and propose a parallel version of the construction algorithm for multi-core CPUs. Our parallel k-d tree construction algorithm can be used to rapidly perform full hierarchy rebuilds for moderately sized to large volume data sets. We reformulate the original, highly serial construction algorithm by replacing the summed-volume table (SVT) that is used to perform fast occupancy queries with a list of partial *summed-volume tables. This gives rise to parallelism at several stages of the algorithm. We show how to achieve high scalability with a carefully crafted parallelization scheme. As a side effect, our construction algorithm also relaxes the tremendous memory overhead imposed by full SVTs. For our scalability study, we have integrated the parallel k-d tree implementation into a ray casting volume rendering pipeline. We present comparisons for various sparse 3-D volumetric data sets where k-d trees are first built interactively and then later used to skip over empty space.*

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Shared memory algorithms; Rendering;***

1. Introduction

Spatial indexing of sparse 3-D uniform volume data is based on data structures that span non-empty volume regions as tightly as possible with as few bounding objects as possible. Spatial indexing data structures and algorithms are often *brick-based*, *hierarchical*, or are based on an *index volume* augmenting the volume

DOI: 10.2312/pgv.20181097

data with additional information on a per-voxel basis. Hierarchical data structures like *k*-d trees subdivide the volume using a divide and conquer strategy. Compared to brick-based indexing, hierarchical data structures ensure moderate traversal costs by reducing the number of bounding objects. This usually comes at higher construction costs in terms of time and memory consumption for the index data structure. With volume data, spatial partitioning is generally preferred over object partitioning, because it is desirable that bounding objects do not overlap. Space partitioning data structures with non-overlapping bounding objects fit well in a GPU volume rendering pipeline, because tree traversal can be trivially performed once per view point on the CPU to put the bounding objects in the right order for subsequent rendering on the GPU. Popular hierarchical space partitioning data structures for volume data are *octrees* as well as *binary space partitioning trees* (BSP). *k*-d trees are BSPs that hierarchically subdivide n-dimensional space into half-spaces using axis-aligned hyperplanes. In contrast to *bounding volume hierarchies* (BVH), which are object partitioning data structures that are popularly used for interactive ray tracing of surfaces and dynamic scenes, space partitioning trees are usually fully rebuilt when the visibility conditions change. This is especially true with volume data, where marginal changes to an alpha transfer function can lead to unpredictable changes in visibility, and consequently to spatial indexing data structures that are dissimilar in shape and volume from data structures that were built for only slightly different transfer functions.

Full reconstruction of *k*-d trees is usually time consuming and does not map well to parallel architectures. Since one objective of spatial indexing is to reduce the number of bounding objects, *k*-d trees are usually shallow, and the splitting phase that divides space into half-spaces is poorly parallelizable. Splitting typically involves occupancy queries to determine if regions of the volume are empty or homogeneous. This type of query is usually either prohibitively slow, or is accomplished using accompanying data structures like summed-volume tables (SVT), a data structure detailed upon in Section 3 that is constructed using an inherently sequential process and that has tremendous memory demands. In this work we propose to replace full SVTs with *partial* SVTs to exploit parallelism, to improve overall memory locality, and to reduce memory consumption.

In order to prove the effectiveness of our approach for *k*-d tree reconstruction, we adapt the work by Vidal et al. [VMD08] and integrate it into an interactive, ray casting-based *direct volume rendering* (DVR) GPU pipeline using post-classification RGBA transfer functions. *k*-d tree construction as proposed in [VMD08] follows a top down approach that can accept local optima as a solution when finding node splitting positions.

It is noteworthy that, due to that fact, the construction method performs especially bad at removing empty space from inner structures of sparse volumes. We emphasize that our contribution is not aimed at improving the *k*-d tree construction algorithm *per se*, but rather at parallelizing a functionality that is integral to and is used by the construction algorithm as an intrinsic function. Fast occupancy queries on arbitrary volume regions in 3-D uniform volumetric grids are essential for rapid spatial index generation regardless of the particular space partitioning algorithm that is used. We believe that our contribution is generally helpful and can be easily adapted to other space partitioning schemes.

The paper is structured as follows. In Section 2 we present related work on 3-D spatial indexing. In Section 3 we briefly recapitulate Vidal et al.'s *k*-d tree construction algorithm that our work is based upon. In Section 4 we propose our parallel *k*-d tree construction algorithm. In Section 5 we present performance results that we compare with measurements based on the original, serial construction algorithm. In Section 6 we briefly conclude this publication.

2. Related Work

Spatial indexing for DVR is e.g. necessary in the context of out-of-core rendering [CKS03], for level-of-detail rendering [WWH*00], or for skipping over empty or homogeneous space [LMK03]. The specific use-case will usually determine the properties of the respective spatial indexing data structure and different use-cases may require opposite parameter settings. Brick resolution for out-of-core techniques e.g. may be different from a typical brick resolution for empty-space skipping [RV06]. It is also not uncommon that brick-based and hierarchical indexing are combined for a coarser volume representation than on a per-voxel level [CNLE09, LCDP13]. Multi-resolution techniques [BHMF08] actually combine bricks of different sizes and have to take special care of how to accommodate texture interpolation at different levels of detail. Hierarchical domain decomposition for volume rendering is often based on *min-max trees* [WFKH07, KTW*11] that allow for efficient culling by storing the minimum and maximum voxel values at the nodes of the hierarchy. Hierarchical indexing data structures traditionally incurred high video memory consumption because indices needed to be stored in linear texture memory [YS93] and because conditional branching e.g. for tree traversal in GPU programs was prohibitive. Nowadays, hierarchical data structures generally aim at low video memory consumption by being defined implicitly [KTW*11] or by requiring only bounding boxes to be traversed during volume integration [VMD08, HAAB*18]. Sparse volume data sets originate from all kinds of applications [WDC12]. Recent advances in the field have tried to improve removal of empty space in inner structures [HAAB*18] or have focused on intelligent ways to combine different space leaping strategies [LBG*16]. Schneider et al. [SR17] use Fenwick trees to accelerate construction of indexing data structures. The use of Fenwick trees relates their work strongly to ours, because Fenwick trees are an extension to the summed-volume tables that we use. We however opted for a solution based on *k*-d trees so we could subdivide our algorithm into a fast CPU tree construction phase, and into a simple GPU ray integration phase without tree traversal on the GPU. A good general overview of spatial indexing techniques for DVR can be found in the article by Beyer et al. [BHP14].

Summed-volume tables conceptually extend summed-area tables (SAT) that were originally used as an alternative to texture mip-mapping [Cro84] to the third dimension. While research on *parallel scan* for 1-D prefix sums [SHZO07, DGS*08] is in general applicable to SVT or SAT construction, parallelization algorithms to efficiently build up SATs have been proposed that scale better on modern processor architectures than their 1-D counterparts [KNI14, PS16]. Nehab et al. [NMLH11] use SATs for recursive image filtering and apply blocking strategies to overlap memory accesses and computation. Applications of SATs or SVTs to DVR include empty-space skipping [VMD08] and ambient occlusion [DVND10].

3. Serial *k*-d Tree Construction

As a starting point for our *k*-d tree construction algorithm, we use the serial construction algorithm originally proposed by Vidal et al. [VMD08]. With their algorithm, construction is performed in a top down fashion from the root node of the *k*-d tree. The root node is found by finding a tight *axis aligned bounding box* (AABB) around all the *alpha classified*, non-empty voxels in the volume data set. The algorithm then aims at finding a good position for an axis aligned splitting plane that divides the parent AABB into two AABBs, and thus the volume into two half-spaces. In order to find a good splitting position, the algorithm considers the longest side of the AABB as the splitting axis, and then, in a manner similar to the binning procedure proposed by Wald [Wal07] for surfaces, evaluates various positions to find the best splitting position along that axis. In order to assess the fitness of the splitting position candidate, AABBs that tightly contain the non-empty voxels in each half-space, and that are fully contained by the AABB of the parent node, are computed. Then the following cost function is minimized over all potential splitting positions:

$$C(p) = V(B_l(p)) + V(B_r(p)), \qquad (1)$$

where p refers to the candidate plane, $B_l(p)$ and $B_r(p)$ are the respective AABBs to the left or the right of the splitting plane, and $V(B_x)$ is the volume of AABB B_x. When an adequate splitting position is found, the algorithm recursively descends to subdivide the two resulting child nodes. The recursion stops when certain halting criteria apply, such as the ratio of empty to non-empty space inside a candidate node dropping below a certain threshold, or the volume of the AABB surrounding the non-empty voxels dropping below a certain percentage of the volume of the whole data set. We generally stick with the thresholds proposed by the authors (5% for the ratio of empty to non-empty space, 10% of the whole volume), as those proved to be effective for the data sets we evaluated the method with. After construction, the *k*-d tree can be traversed from the root to assemble a list of leaf nodes that can then be integrated with a simple volume rendering pipeline in back to front or front to back order. Since leaf nodes in a *k*-d tree will not overlap, the resulting partitioning fits well in a typical GPU volume rendering pipeline. With 3-D texture-slicing, the pipeline is simply restarted for each leaf node's bounding box. In a ray casting pipeline, the list of bounding boxes can be passed to the compute kernel performing integration, adding an extra box traversal loop on top of the integration loop, which is however trivial because it only needs to iterate over the leaf nodes. In both cases, additional memory transfer overhead from the CPU to the GPU is negligible.

In order to find tight bounding boxes around non-empty voxels in a node, Vidal et al. use an iterative algorithm that starts with an initial AABB that is successively shrunk. They choose the parent AABB, subject to the plane split, as an initial bound, and record the number of non-empty voxels contained inside the AABB. It is then safe to shrink the AABB as long as the tighter box contains the same number of non-empty voxels. The authors iteratively move the minimum and maximum corners towards each other along each cartesian coordinate axis, until the next smaller box would contain less voxels than the current one. In that case, the current AABB tightly contains the non-empty voxels. Obviously, the performance of this operation, and thus the performance of the *k*-d tree construction algorithm, depends on being able to quickly find the number of non-empty voxels inside an AABB.

To quickly determine the occupancy inside AABBs, Vidal et al. use an SVT. SVTs are especially well suited for this type of query because they allow for evaluating the volume in a cubic region in constant time. SVTs are 3-D matrices B that store the sums

$$B(l,m,n) = \sum_{i=1}^{i\leq l} \sum_{j=1}^{j\leq m} \sum_{k=1}^{k\leq n} A(i,j,k), \qquad (2)$$

where $l \leq L, m \leq M, n \leq N$ are indices into the (generally real-valued) 3-D source matrix A with dimensions L, M, and N. SVTs are an extension of prefix sums into three dimensions. An intuitive way to construct SVT B from source matrix A is to first copy the whole content of A into B and then to perform the recursive operation

$$B(i,j,k) = \begin{cases} 0 & i,j \text{ or } k \leq 1 \\ B(i,j,k) + B(i-1,j-1,k-1) & \\ +B(i-1,j,k) - B(i,j-1,k-1) & \\ +B(i,j-1,k) - B(i-1,j,k-1) & \\ +B(i,j,k-1) - B(i-1,j-1,k) & \end{cases} \qquad (3)$$

on all elements of B. SVTs have however several downsides: building up an SVT is an inherently serial process. While research on parallel scan operations is generally applicable to prefix sum computation in higher dimensions, the scalability of the respective parallel algorithms is not embarrassingly parallel. SVTs further have a high memory demand, because data items in general need to be stored with higher precision than the source type to avoid overflows. Even if memory consumption is not an issue, storing data items with unnecessarily high precision will have an influence on cache utilization because cache lines and caches will generally tend to contain less actual information. For those reasons, full SVTs are potentially ill suited in regard to modern CPU architectures that have large cache hierarchies and multiple parallel cores. Furthermore, in the specific case where SVTs are used as auxiliary data structures for *k*-d tree construction, building up the SVT is by far the most time consuming task.

The serial *k*-d tree construction algorithm can be logically divided into two parts: SVT construction and recursive node splitting. Since the resulting *k*-d trees are shallow by construction, splitting will typically be performed for only a few nodes. For typical data sets, SVT construction will therefore potentially take about two orders of magnitude longer than the recursive node splitting phase. We therefore consider optimizations to the original algorithm that trade node splitting execution time for construction time of auxiliary data structures to be generally worthwhile.

4. Parallel k-d Tree Construction

We propose a data parallel variant of the algorithm outlined above that runs on the CPU. Our decision to perform k-d tree construction on the CPU is based on the assumption that video memory is usually a more scarce resource than CPU memory, so we want to avoid storing auxiliary data structures in there. With a CPU implementation, it is possible to keep auxiliary data structures in memory so that we do not have to reallocate memory all the time, and so we can store individual data items in an order that is beneficial for cache efficient traversal. Our whole choice of k-d tree construction algorithm is actually based on the fact that it lends itself well to construction on the CPU, and that it anyway fits well in a GPU-based volume rendering pipeline because memory transfer overhead on a per-frame and even on a per-update basis is negligible.

4.1. Partial summed-volume table generation

Building up full SVTs in a naive way is an inherently sequential process, and we expect parallel scan operations that are typically used for calculating prefix sums to not scale well with the number of threads dedicated to the task. We therefore propose to not build up full SVTs at all, but base our parallel algorithm on partial SVTs storing partial sums for fixed-size three-dimensional blocks. Decomposing the data set in that way allows for an *embarrassingly parallel* SVT construction phase. Since we are particularly interested in the binary information if a certain voxel is visible or not, we further propose to not store fully classified alpha values inside the blocks, but only binary occupancy values. Based on those restrictions, it is possible to represent fully occupied partial SVTs where each voxel is visible with 16-bit unsigned integer values for a block size of 32^3. In comparison, with a full SVT storing not only binary but general occupancy information, 64-bit precision data items would be necessary even for moderately sized volumes.

We allocate a contiguous region of CPU memory to contain all the partial SVTs whenever the actual volume is reloaded, which we assume to happen infrequently to only once during a rendering session. In addition to the 16-bit partial SVT array, we store a copy of the unclassified volume data, that we store in a 32^3 block layout according to the memory layout of the SVT. When the transfer function changes, we iterate over the swizzled copy of the volume, apply classification to each voxel, and set the respective entry in the respective partial SVT to either 0 or 1 according to the voxel's visibility.

We then construct the partial SVTs in parallel using multithreading. We expect this operation to scale nearly linear with the number of parallel threads. The 64 KB sized blocks will fit into the L1 cache of each individual CPU core on a typical modern CPU, so that constructing the partial SVTs can be expected to happen in a fully cached fashion. In addition to that, the memory access pattern and branching behavior of SVT construction is highly predictable and uniform for each partial SVT. We carefully tested different SVT construction patterns to calculate the recursion from Equation 3 to construct the partial SVTs from the initially 0 or 1 filled temporary SVT arrays for performance. The pattern that proved best performance wise does not implement the branch, but rather involves first calculating three prefix sums for the $(i, j = 1, k = 1)$, $(i = 1, j, k = 1)$, and $(i = 1, j = 1, k)$ scanlines, followed by calculating the summed-*area* tables for the the three "sides" where i, j, and k are 1, respectively. We then start the recursion at $(i, j, k) = 2$. Patterns that involved conditional branching or additional zero borders in memory proved to be slightly inferior with our tests.

4.2. Performing occupancy queries on partial summed-volume tables

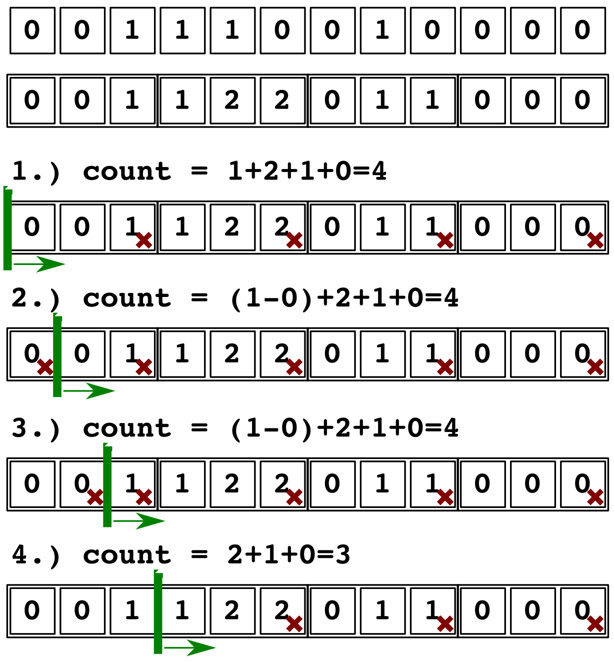

Figure 2: *The image shows how occupancy queries on partial SVTs (the graphic shows this in 1-D, concepts are applicable to 3-D) are performed sequentially. The top row represents a sparse input sequence. The second row represents the partial prefix sums of size 3 for the input sequence. The rows labeled 1.) through 4.) illustrate how a 1-D hyperplane (indicated by the green line and arrow) is swept from left to right to find a close boundary from the left hand side. The row labeled 4.) ends the iteration because the number of voxels inside the bounds changes from four to three.*
The image illustrates the memory accesses required to evaluate the boundary of interest. The memory locations being accessed are indicated by red crosses. It can be seen that internal SVTs only contribute with their overall sum which is stored in the last memory position of the SVT. SVTs at the border however need to be accessed at general positions in order to determine the voxel count. It may generally be useful to build a flat hierarchy over the partial SVTs, so that in the example the result of the sum $2+1$ could be accessed from a higher-level SVT, while only the border SVTs needed to be loaded from memory completely. In this paper we however propose an alternative approach to determine the boundaries in a data parallel fashion that maps better to modern CPU architectures.

In the extreme case, prefix summation can be performed in one of two ways – either by pretabulating all prefix sums that are of potential interest, or by performing the whole summation in place at the time the prefix sums are needed. The first case involves high memory consumption and possibly a time consuming preprocessing step that is hard to parallelize. The second case involves computation with $O(n)$ complexity in the number of input items n. Our algorithm based on partial summation can be seen as a middle ground between the two extreme cases. By computing only partial sums, parallelization potential increases, but at the cost of no longer being able to perform occupancy queries using the partial SVTs with $O(1)$ constant time complexity. This can be easily seen from the 1-D example in Figure 2. In order to calculate the occupancy of a volumetric region, it is necessary to perform a summation over all bricks that are fully or partially covered by that region, making this an $O(n)$ time complexity operation, but in the number of partial SVTs, and not in the number of input voxels. As can be seen from the figure, it is generally possible to make this operation hierarchical and thus reduce the complexity to $O(logn)$ in the number of partial SVTs. In that case, summation would be performed over a higher-level SVT, and over the set of partial SVTs overlapping the borders of the region of interest. Doing so involves storing additional levels of SVTs. Assuming that the partial SVT size along each level is constantly 32^3, such a hierarchy would obviously be shallow; with one additional level – i.e. one "root" SVT placed upon up to 32^3 partial SVTs, it is possible to accommodate volumes of sizes up to 1024^3, so that the hierarchy traversal could in practice simply be hardcoded to contain two or three levels. As can also be seen from Figure 2, even with a hierarchy of partial SVTs, occupancy queries still have linear complexity with respect to the number of SVTs that overlap with the border of the region of interest.

Keeping in mind that the occupancy query is used to *iteratively* shrink the AABB around non-empty voxels, it is obvious that linear summation can in this case become a performance bottleneck. The number of partial SVTs to iterate over can be problematic when building up the top levels of the k-d tree where the volume of the candidate AABBs is relatively high. In addition, linear summation does not scale to multiple cores, and we want our algorithm to be able to be robust against future hardware development that we expect to be comprised of even higher parallelism and more independent compute cores on a single chip.

We therefore propose to reformulate the whole node splitting phase of the k-d tree construction to no longer perform an occupancy query on the whole set of partial SVTs at all, but to rather perform boundary computations locally for each individual partial SVT, and then combine the locally computed boundaries using the *union* operation (cf. Figure 3). We further propose to combine such a strategy with a simple culling approach that performs the boundary computation only for partial SVTs that are non-empty (identifying empty SVTs can be done in constant time).

On the downside, this potentially involves a number of unnecessary boundary computations. These are however performed in thread-local memory which we presume to persist in L1 cache and thus to be cheap operations. We also do not expect the number of local boundary computations during the note splitting phase to be severe. At the root level of the recursion, we would expect many partial SVTs that would require testing, but on the other hand, with sparse data sets, we would expect that many partial SVTs can be immediately culled. At the lower levels of the recursion, we would generally expect that only few partial SVTs overlap the region of interest, so that at that level boundary computations usually contribute to the overall boundary.

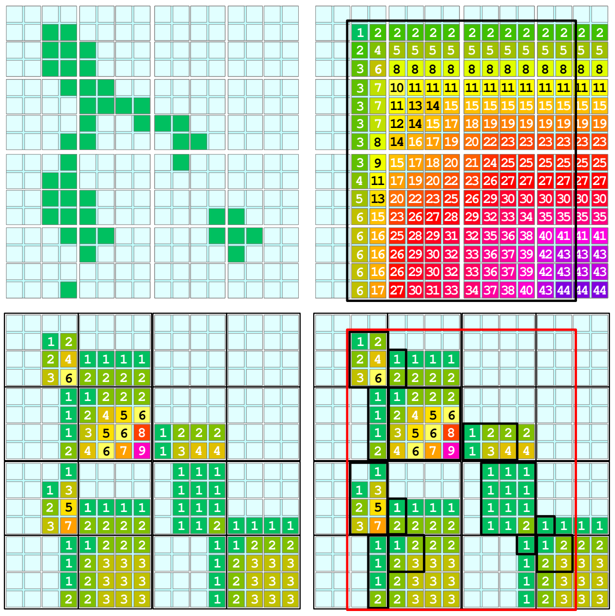
Figure 3: *Finding tight bounding boxes with SVTs. The top left image shows the original data set, with voxels being classified as either visible (green) or invisible (light blue). The top right image shows the process of finding bounding boxes with the serial algorithm. The SVT is used to iteratively shrink the loose initial bounding box until a tight bounding box is found. Our parallel version of the algorithm is based on partial SVTs (bottom left). Local bounding boxes are constructed in parallel. The global bounding box can be calculated in several ways, e.g. in parallel for each partial SVT and later combining them using the* union *operation (right). While the illustration depicts the process in 2-D, concepts are directly applicable to 3-D.*

In the following section we compare our approach with a serial implementation based on the original paper by Vidal et al. and also take the three aforementioned strategies to find tight bounds around non-empty volume regions into account.

5. Results

For our performance measurements we integrated our parallel k-d tree construction algorithm in the DVR scientific visualization library Virvo [SWWL01]. We therefore wrote a custom ray casting plugin using the NVIDIA CUDA toolkit. The simple ray traversal code is based on the emission plus absorption model with optional local shading [Max95]. In contrast to an ordinary ray caster, our plugin code has an extra loop that first tests primary rays for intersection with leaf node bounding boxes, and potentially integrates over the volume of the box using ray marching. We use OpenMP pragmas to parallelize the partial SVT generation and the computation of the local AABBs that are later combined using a serial union operation. In order to evaluate the performance of our parallel construction algorithm, we performed test runs using four volume data sets of different size that are depicted in Figure 4. We carefully chose transfer functions with different properties. One transfer function was designed so that a *k*-d tree with only a single leaf node would result. Two more transfer functions were designed with an effort so that the resulting leaf nodes in total had different volume.

We performed our evaluation on a workstation computer equipped with an Intel Core i7-3960X processor with six cores, twelve threads when enabling simultaneous multithreading (SMT), and a base frequency of 3.30 GHz. The L1 cache of this processor has a size of 384 KB, so that we expect all blockwise operations of our algorithm to be performed in fast on-chip memory.

We compare the performance of our parallel implementation to the original serial *k*-d tree construction algorithm. We also want to evaluate how our parallel node splitting phase with ensuing union operation to combine local bounding boxes compares to a serial node splitting phase as illustrated in Figure 2. We therefore test the three modalities: *serial* (SER), *serial node splitting* (SER-SPLIT), and *all parallel* (PAR). We present overall performance results in Table 1. Figures 5, 6, 7, and 8 illustrate the ratio between the construction times spent for the SVTs and for the node splitting phase. The results of our measurements indicate that the parallel construction scheme generally provides high scalability. In contrast to the serial algorithm, *k*-d tree reconstruction for moderately sized data sets of about 250^3 can generally be performed with very low latency; *k*-d tree reconstruction for data sets of about the size 500^3 is possible with latency that allows for interactive manipulation of alpha transfer functions. For data sets made up of about 1000^3 voxels, latency can, depending on the transfer function, be as low as three seconds. Our scalability tests with the Drosophila data set ($1000 \times 1000 \times 910$ voxels) show however that for larger volumes, a sequential node splitting phase can result in extremely high construction time, and that the appearance of the transfer function has a strong influence and can result in linear summations over partial SVTs becoming the major performance bottleneck of the algorithm. The tests also show that this issue can be mitigated by using a parallel node splitting phase like the one we proposed above. However, from the four graphs a trend can be seen that with increasing volume size, the ratio between SVT construction and node splitting gradually shifts toward node splitting. We therefore consider further improving the node splitting phase interesting future work. The node splitting phase as it is currently designed simply computes local AABBs for every partial SVT inside the region of interest. Especially for larger volumes, it would be interesting to evaluate if calculating and unifying bounding boxes only for the SVTs at the borders of the region of interest (which would reduce the number of computations and memory accesses, but would come at the expense of a more complicated and more serial control flow) can help to improve construction time in this case.

6. Conclusions and Future Work

We presented a parallel algorithm to fully rebuild *k*-d trees for spatial indexing of sparse volumes. Our algorithm is based on using partial SVTs that can be efficiently constructed and later queried in parallel and in fast CPU cache memory. We believe that partial SVTs can be used to generally parallelize inherently serial spatial index data structure construction algorithms, and that it is not limited to the divide and conquer strategy used in this paper. Scalability studies indicate that the algorithm is suitable for future multi-core architectures.

In the future we would like to evaluate if the node splitting phase of our algorithm can be further optimized by considering only SVTs at node borders for bounding box calculation. This would however result in a slightly more complicated and also more serial control flow. We also believe that the partial SVT construction phase can be further optimized using SIMD parallelization, e.g. by rearranging the data layout so that data is traversed along a diagonal of the SVT's bounding box. With that, multiple partial sums could be calculated in parallel when constructing individual partial SVTs. For simplicity, we used a serial *union* operation to combine AABBs to form a global bounding box. We presume that a parallel *union* operation would help to further improve scalability of our algorithm especially for large data sets.

Acknowledgements

The aneurism data set was published online by Philips Research. The bonsai tree data set was published online by S. Roettger from University of Stuttgart. Both data sets are available online from [TC112]. The christmas tree data set was made available publicly by the Department of Radiology of the University of Vienna and is obtainable from [Erl06]. The Drosophila brain microscopy data set was provided with friendly permission by Kei Ito from the Institute of Zoology, University of Cologne, who is also affiliated with the Institute of Molecular and Cellular Biosciences, University of Tokyo.

References

[BHMF08] Beyer J., Hadwiger M., Möller T., Fritz L.: Smooth mixed-resolution GPU volume rendering. In *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2008), SPBG'08, Eurographics Association, pp. 163–170. doi:10.2312/VG/VG-PBG08/163-170. 2

[BHP14] Beyer J., Hadwiger M., Pfister H.: A Survey of GPU-Based Large-Scale Volume Visualization. In *EuroVis - STARs* (2014), Borgo R., Maciejewski R., Viola I., (Eds.), The Eurographics Association. doi:10.2312/eurovisstar.20141175. 2

[CKS03] Correa W. T., Klosowski J. T., Silva C. T.: Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (Washington, DC, USA, 2003), PVG '03, IEEE Computer Society, pp. 2–. doi:10.1109/PVG.2003.10002. 2

[CNLE09] Crassin C., Neyret F., Lefebvre S., Eisemann E.: Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)* (Boston, MA, Etats-Unis, feb 2009), ACM, ACM Press. to appear. 2

[Cro84] Crow F. C.: Summed-area tables for texture mapping. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 207–212. doi:10.1145/964965.808600. 2

[DGS*08] Dotsenko Y., Govindaraju N. K., Sloan P.-P., Boyd C., Manferdelli J.: Fast scan algorithms on graphics processors. In *Proceedings of the 22Nd Annual International Conference on Supercomputing* (New York, NY, USA, 2008), ICS '08, ACM, pp. 205–213. 2

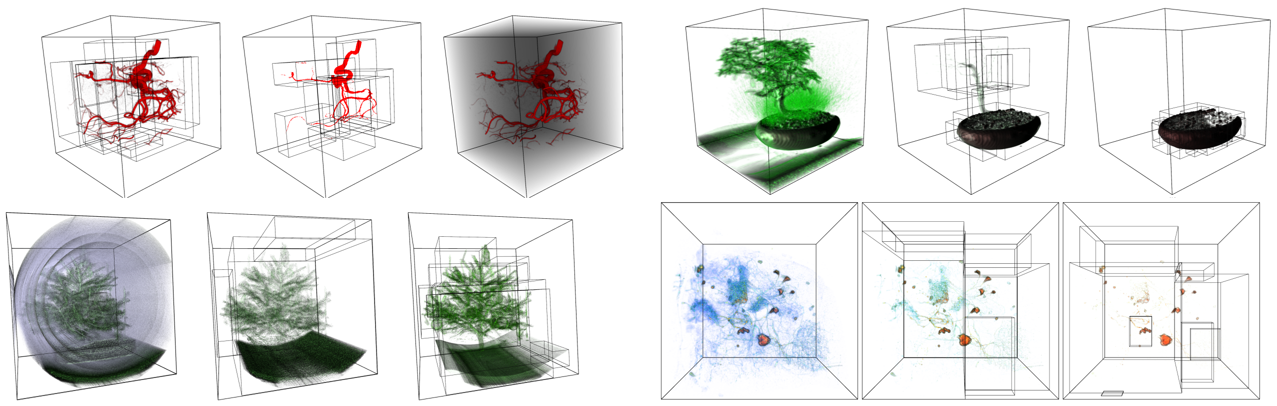

Figure 4: *The test results in Section 5 were obtained by constructing k-d trees for the combinations of data sets and transfer functions depicted in the figure. The aneurism data set depicted on the top left of the figure has a resolution of* $256 \times 256 \times 256$ *voxels. The bonsai tree data set on the top right has the same resolution of* $256 \times 256 \times 256$ *voxels. The CT-scan of the christmas tree depicted in the bottom row on the left side has a resolution of* $512 \times 499 \times 512$ *voxels. The section of the Drosophila brain microscopy data set to the bottom right has a resolution of* $1{,}000 \times 1{,}000 \times 910$ *voxels. Transfer functions were carefully chosen to find different numbers of leaf nodes with different volume.*

	Aneurism			Bonsai			Xmas Tree			Drosophila		
	TF 1	TF 2	TF 3	TF 1	TF 2	TF 3	TF 1	TF 2	TF 3	TF 1	TF 2	TF 3
SER	0.18	0.18	0.18	0.18	0.18	0.18	1.44	1.44	1.44	9.36	9.38	9.39
SER-SPLIT	0.05	0.03	0.04	0.04	0.05	0.04	0.41	0.75	0.98	4.79	20.4	32.5
PAR	0.04	0.03	0.02	0.03	0.04	0.03	0.22	0.30	0.34	5.22	2.80	3.20

Table 1: *Construction times in seconds for the four test data sets and the three respective transfer functions. We test serial construction (SER), parallel constructions of partial SVTs, but with a serial node splitting phase (SER-SPLIT), and parallel construction with parallel node splitting.*

[DVND10] Díaz J., Vázquez P.-P., Navazo I., Duguet F.: Real-time ambient occlusion and halos with summed area tables. *Computers & Graphics 34*, 4 (2010), 337 – 350. Procedural Methods in Computer Graphics Illustrative Visualization. 3

[Erl06] Erlangen U.: Vollib, 2006. Accessed: 2018-03-06. URL: http://www9.informatik.uni-erlangen.de/External/vollib/. 6

[HAAB*18] Hadwiger M., Al-Awami A. K., Beyer J., Agos M., Pfister H.: SparseLeap: Efficient empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics* (2018). 2

[KNI14] Kasagi A., Nakano K., Ito Y.: Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with gpu implementations. In *2014 43rd International Conference on Parallel Processing* (Sept 2014), pp. 251–260. doi:10.1109/ICPP.2014.34. 2

[KTW*11] Knoll A., Thelen S., Wald I., Hansen C., Hagen H., Papka M.: Full-resolution interactive CPU volume rendering with coherent BVH traversal. In *Proceedings of IEEE Pacific Visualization 2011* (2011), pp. 3–10. 2

[LBG*16] Labschütz M., Bruckner S., Gröller M. E., Hadwiger M., Rautek P.: JiTTree: A just-in-time compiled sparse GPU volume data structure. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (Jan 2016), 1025–1034. doi:10.1109/TVCG.2015.2467331. 2

[LCDP13] Liu B., Clapworthy G. J., Dong F., Prakash E. C.: Octree rasterization: Accelerating high-quality out-of-core gpu volume rendering. *IEEE Transactions on Visualization and Computer Graphics 19*, 10 (Oct 2013), 1732–1745. doi:10.1109/TVCG.2012.151. 2

[LMK03] Li W., Mueller K., Kaufman A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *IEEE Visualization, 2003. VIS 2003.* (Oct 2003), pp. 317–324. doi:10.1109/VISUAL.2003.1250388. 2

[Max95] Max N.: Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (Jun 1995), 99–108. doi:10.1109/2945.468400. 5

[NMLH11] Nehab D., Maximo A., Lima R. S., Hoppe H.: Gpu-efficient recursive filtering and summed-area tables. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 176:1–176:12. 2

[PS16] Papatriantafyllou A., Sacharidis D.: High performance parallel summed-area table kernels for multi-core and many-core systems. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833* (New York, NY, USA, 2016), Springer-Verlag New York, Inc., pp. 306–318. doi:10.1007/978-3-319-43659-3_23. 2

[RV06] Ruijters D., Vilanova A.: Optimizing gpu volume rendering. In *WSCG - Winter School of Computer Graphics* (Feb 2006), vol. 14, pp. 9–16. 2

[SHZO07] Sengupta S., Harris M., Zhang Y., Owens J. D.: Scan primitives for GPU computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, Switzerland, 2007), GH '07, Eurographics Association, pp. 97–106. 2

[SR17] Schneider J., Rautek P.: A versatile and efficient GPU

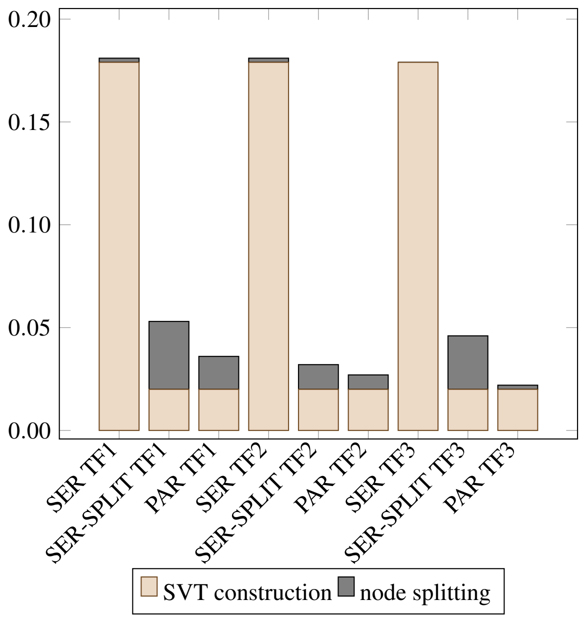

Figure 5: *Aneurism data set: construction times (seconds) to illustrate the ratio between time spent for SVT construction and for node splitting.*

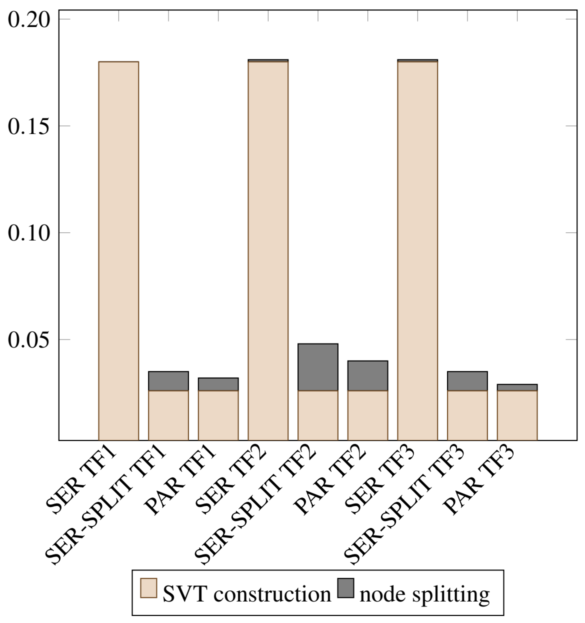

Figure 6: *Bonsai tree data set: construction times (seconds) to illustrate the ratio between time spent for SVT construction and for node splitting.*

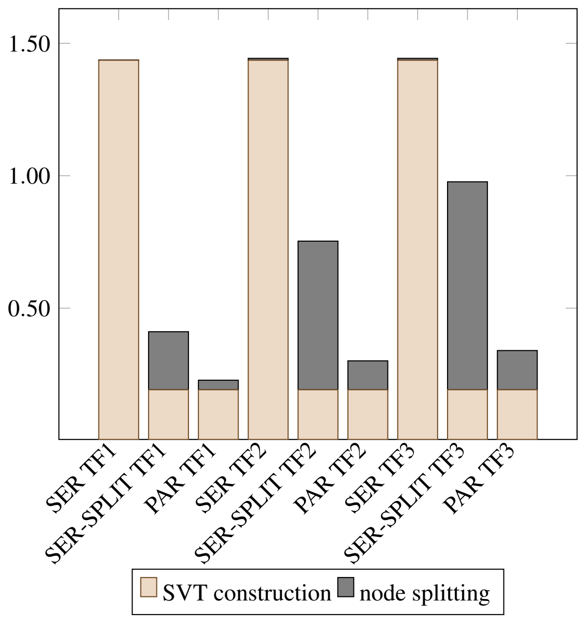

Figure 7: *Christmas tree data set: construction times (seconds) to illustrate the ratio between time spent for SVT construction and for node splitting.*

data structure for spatial indexing. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (Jan 2017), 911–920. doi:10.1109/TVCG.2016.2599043. 2

[SWWL01] SCHULZE J., WOESSNER U., WALZ S., LANG U.: Volume rendering in a virtual environment. *Immersive Projection Technology and Virtual Environments 2001: proceedings of the Eurographics Workshop in Stuttgart, Germany, May 16-18, 2001* (2001), 187. 5

[TC112] TC18: Tc18, 2012. Accessed: 2018-03-06. URL: http://www.tc18.org/code_data_set/3D_images.php. 6

[VMD08] VIDAL V., MEI X., DECAUDIN P.: Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools 13*, 2 (2008), 21–36. 2, 3

[Wal07] WALD I.: On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), RT '07, IEEE Computer Society, pp. 33–40. 3

[WDC12] WANG Y., DOU W., CONSTANS J. M.: Accelerating volume ray casting by empty space skipping used for computer-aided therapy. In *2012 International Conference on Audio, Language and Image Processing* (July 2012), pp. 661–667. 2

[WFKH07] WALD I., FRIEDRICH H., KNOLL A., HANSEN C. D.: Interactive isosurface ray tracing of time-varying tetrahedral volumes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (Nov 2007), 1727–1734. doi:10.1109/TVCG.2007.70566. 2

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMANN K., ERTL T.: Level-of-detail volume rendering via 3d textures. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization* (New York, NY, USA, 2000), VVS '00, ACM, pp. 7–13. 2

[YS93] YAGEL R., SHI Z.: Accelerating volume animation by space-leaping. In *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on* (Oct 1993), pp. 62–69. doi:10.1109/VISUAL.1993.398852. 2

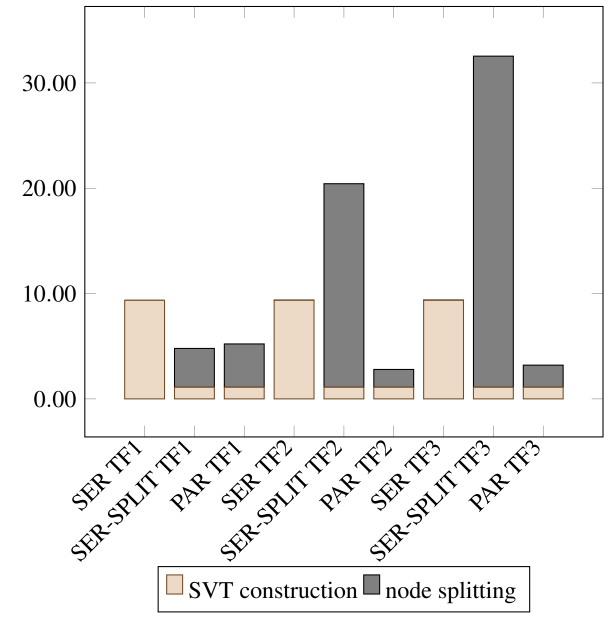

Figure 8: *Drosophila data set: construction times (seconds) to illustrate the ratio between time spent for SVT construction and for node splitting.*