Binned *k*-d Tree Construction for Sparse Volume Data on Multi-Core and GPU Systems

Stefan Zellmann, *Member, IEEE*, Jürgen P. Schulze, and Ulrich Lang

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. In a recent publication we showed how to port a clever state of the art *k*-d tree construction algorithm to a multi-core CPU architecture and by means of thorough optimization we were able to obtain interactive reconstruction rates for moderately sized to large data sets. The construction scheme is based on maintaining *partial* summed-volume tables that fit in the L1 cache of the multi-core CPU and that allow for fast occupancy queries. In this work we propose a GPU implementation of the parallel *k*-d tree construction algorithm and compare it with the original multi-core CPU implementation. We conduct a thorough comparative study that outlines performance and scalability of our implementation.

Index Terms—Scientific Visualization, Sparse Data, Direct Volume Rendering, *k*-d Tree, Parallel and GPGPU Computing

1 Introduction

Spatial indexing data structures for sparse 3-d uniform volumes span non-empty volume regions as tightly and with as few bounding objects as possible. They are usually either *brick-based*, *hierarchical*, or use an *index volume* to augment the volume data with additional per-voxel information. Hierarchical data structures like *k*-d trees subdivide the volume using a divide and conquer strategy. Compared to brick-based indexing, hierarchical data structures are better suited to achieve moderate traversal costs by reducing the number of bounding objects, but construction time and memory consumption are usually higher. With volume data, spatial partitioning is generally preferred over object partitioning, because for integration during rendering it is desirable that bounding objects do not overlap. 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 *k*-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 whole 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, which is usually constructed using an inherently sequential process 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. [1] 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 [1] follows a top down approach that can accept local optima as a solution when finding node splitting positions.

It is worth mentioning that, due to that fact, the construction method performs particularly 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

- S. Zellmann and U. Lang are with the Chair of Computer Science, University of Cologne, 50923, Köln, Germany. E-mail: {zellmann, lang}@uni-koeln.de.
- J.P. Schulze is with the Department of Computer Science, University of California San Diego, La Jolla, CA 92093 USA. E-mail: jschulze@ucsd.edu.

Manuscript received 16 Nov. 2018; revised 15 July 2019; accepted 28 Aug. 2019. Date of publication 3 Sept. 2019; date of current version 28 Jan. 2021. (Corresponding author: Stefan Zellmann.)
Recommended for acceptance by P. Lindstrom.
Digital Object Identifier no. 10.1109/TVCG.2019.2938957

helpful and can be easily adapted to other space partitioning schemes.

This paper extends our prior work originally published in [2]. While the aforementioned paper was centered around an implementation for multi-core CPU architectures only, in this paper we present alternative implementations for both CPUs and GPUs. While the basic scheme of constructing shallow *k*-d trees using SVTs is the same for both implementations, the parallelization strategies differ substantially. For the GPU implementation we had to make a number of compromises so that the output of the two construction algorithms is not exactly the same. We therefore perform a comparative performance and scalability study to assess the effectiveness of the algorithm on either platforms, and also a qualitative study to compare how well the respective *k*-d trees cull empty space and how that affects rendering performance.

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 targeted at multi-core CPUs as well as an adapted version that runs on GPUs. In Section 5 we present results for the two implementations both in terms of tree construction and rendering performance. We briefly discuss the findings from the paper in Section 6. In Section 7 we conclude this publication.

2 Related Work

Spatial indexing for DVR is e.g. necessary in the context of out-of-core rendering [3], for level-of-detail rendering [4], or for skipping over empty or homogeneous space [5]. While earlier work concentrated on interactive rendering of hierarchical volume data [6], with more powerful GPUs the more interesting task today is to interactively construct the spatial index [2]. 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. The brick resolution for out-of-core techniques e.g. may be different from a typical brick resolution for empty-space skipping [7]. It is also not uncommon that brick-based and hierarchical indexing are combined for a coarser volume representation than on a per-voxel level [8], [9]. Multi-resolution techniques [10] combine bricks of different sizes. Special care must be taken to accommodate texture interpolation at different levels of detail. Hierarchical domain decomposition for volume rendering is often based on *min-max trees* [11], [12], which allow for efficient culling by storing minimum and maximum density values at their nodes. Hierarchical indexing data structures traditionally incurred high video memory consumption because indices needed to be stored in linear texture memory [13] 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 [12] or by requiring only bounding boxes to be traversed during volume integration [1], [14]. Sparse volume data sets originate from all kinds of applications [15]. Recent advances in the field have tried to improve removal of empty space in inner structures [14] or have focused on intelligent ways to combine different space leaping strategies [16]. Schneider et al. [17] use Fenwick trees to accelerate construction of indexing data structures. The use of Fenwick trees strongly relates their work to ours, because Fenwick trees are an extension to the summed-volume tables we use. A good general overview of spatial indexing techniques for DVR can be found in the article by Beyer et al. [18].

Summed-volume tables conceptually extend summed-area tables (SAT) that were originally used as an alternative to texture mip-mapping [19] to the third dimension. Research on *parallel scan* for 1-d prefix sums [20], [21] is in general applicable to SAT and SVT construction. Davidson et al. [22] and Harris [23] devised parallel scan algorithms specifically for the GPU. Parallel construction algorithms usually assign blocks of the input table to individual threads or groups of threads and then calculate 1-d prefix sums along each spatial dimension [23], or they directly build up the SAT or SVT inside the block using the respective 2-d or 3-d construction scheme [24], [25]. Bilgic et al. [26] use the parallel scan algorithm from [23] to construct 4 megapixel SATs on the GPU in 5 to 10 milliseconds. Nehab et al. [27] use SATs for recursive image filtering and apply blocking strategies to overlap memory accesses and computation. Other applications of SATs or SVTs to DVR include empty-space skipping [1] and ambient occlusion [28].

Spatial index construction for surface ray tracing has gained significant research interest over the last years. For fully dynamic and deformable scenes, the tasks of constructing the spatial index *and* of using the spatial index to accelerate ray/object intersection are considered per frame operations. While early work in this area was focused on spatial index construction and ray tracing on the CPU [29], [30], later work has concentrated on constructing and ray tracing spatial indices on the GPU [31], [32], [33], [34]. Construction schemes can be generally classified into top down and bottom up [35], where bottom up often implies execution of a sequence of $O(n)$ algorithms over all input primitives, like e.g. initially sorting primitives on a space-filling curve using radix sort. While top down construction using the *surface area heuristic* (SAH) and a plane sweeping strategy is generally considered superior to most other strategies regarding the quality of the resulting tree, this method is slow at constructing spatial indices for scenes with a large number of triangles. Binning [30] is a method to discretize the otherwise continuous sweeping algorithm and can significantly reduce the number of candidate planes to test and thus the overall construction time. To the best of our knowledge, spatial index construction for direct volume rendering, where the spatial index is built on the GPU, has so far not been the topic of any research paper.

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. [1]. With their algorithm, construction is performed in a top down fashion starting from the root node to the leaves of the *k*-d tree. The root node is constructed 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 searches the optimal position for an axis

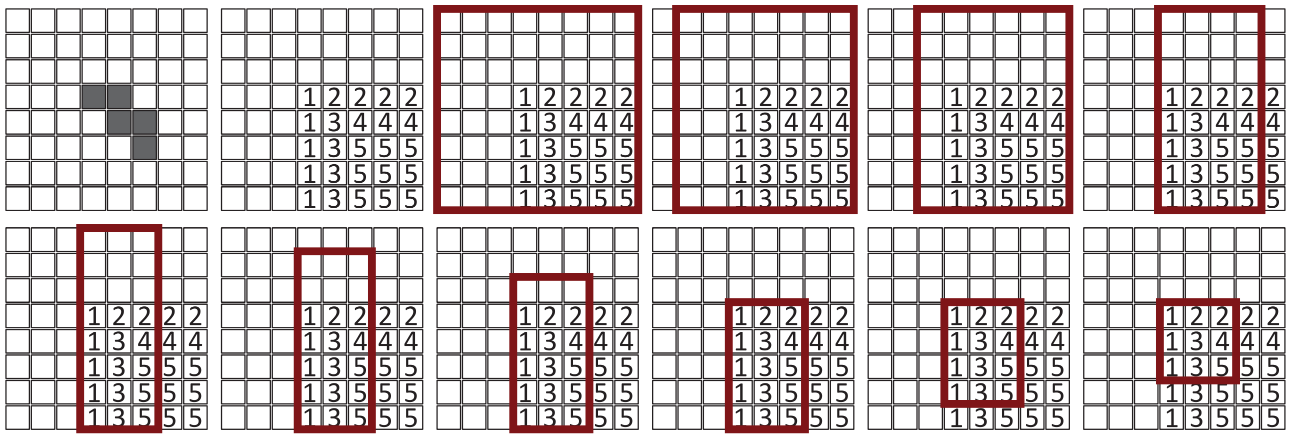


Fig. 1. *Finding tight bounding boxes with regular SVTs.* This illustration should be read from left to right and top to bottom. The serial algorithm originally proposed by Vidal et al. [1] is comprised of calculating the SVT for the volume (first two images) in order to later find tight axis-aligned bounding boxes around the occupied regions of space (the remaining ten images). The illustration shows how the volume of a conservative AABB is successively minimized towards the AABB tightly bounding the occupied regions of space. This procedure is first carried out in the positive and negative x directions, and then in the positive and negative y directions. While the illustration shows this process in 2-d with SATs, the procedure is directly applicable to 3-d by using SVTs. When shrinking the AABBs, the occupancy is initially determined in constant time for the conservative estimate using the SVT. During the course of shrinking the initial AABB towards the AABB with minimal volume, the SVT is used to ensure that any new AABB will have the same occupancy as the conservative AABB. The process of finding a tight AABB can be accelerated by incorporating binary search.

aligned splitting plane that divides the parent AABB into two AABBs, and thus the volume into two half-spaces. Note that the resulting data structure stores non-overlapping AABBs so one could argue that it actually resembles a BVH more than it resembles a *k*-d tree. In order to find a favorable splitting position, the algorithm considers the longest side of the AABB as the splitting axis, and then evaluates various positions to find the best splitting position along that axis. For each candidate plane, AABBs are computed that tightly bound the voxels inside the two respective half-spaces and that fall inside the AABB of the parent node. Then the following cost function is minimized over all potential splitting positions:

$$C(p) = V(B_l(p)) + V(B_r(p)), \tag{1}$$

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. The thresholds proposed by the authors are 5 percent for ratio of empty to non-empty space and 10 percent of the whole volume. In Section 5 we amongst others evaluate if those halting criteria are still useful on contemporary hardware.

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. Another option (that possibly better matches contemporary hardware) is to build deeper trees that are traversed per ray in the compute kernel, which would trade code complexity for being able to cull more empty space.

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 (cf. Fig. 1). 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

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

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 extend the prefix sum concept 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

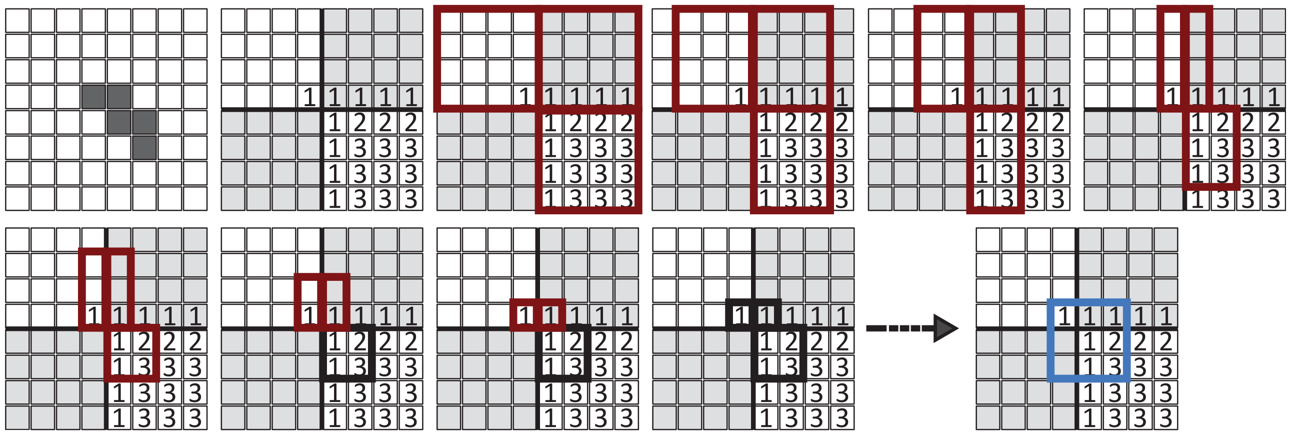


Fig. 2. *Finding tight bounding boxes with partial SVTs.* This illustration should be read from left to right and top to bottom. With our parallel multi-core CPU implementation, partial SVTs are constructed that fit inside the L1 cache of each core (first two images). The following eight images show how tight AABBs can be found in parallel, where each thread is again responsible for processing one block. Finally, the local results are combined using a trivial union operation. Note how the number of local results is proportional to the number of blocks, which with this implementation is proportional, but low compared to the number of voxels. It is thus advantageous to prefer trivial combine over parallel reduction, because the former can be performed right after computing the local AABB and without having to wait until all other local results are known.

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

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 respective algorithms are 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 real 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 (at least with the original implementation proposed by Vidal et al.), 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 two data parallel variants of the algorithm outlined above. We first present the parallel implementation that we proposed in our recent publication [2]. We then informally analyze the scalability of this implementation with regard to later porting the algorithm to the GPU. This informal discussion is based on observations we made during the process of porting the algorithm to the GPU. We then describe a parallel GPU implementation that is loosely based on the CPU parallelization scheme but employs a binning strategy to reduce the costs of the parallel node splitting phase. Due to the binning strategy, the GPU algorithm will not produce the exact same spatial index as the CPU algorithm does, but the overall structure of the algorithm remains the same.

4.1 Multi-Core CPU Implementation

Building up full SVTs in a naive way is an inherently sequential process. For better scalability, we propose to not build a full SVT at all, but rather only partial SVTs storing partial sums inside blocks whose size is advantageous regarding the L1 cache size of the target platform. Decomposing the data set in this way allows for an *embarrassingly parallel* SVT construction phase. Occupancy queries on partial SVTs are then performed in the L1 cache during the parallel node splitting phase, and local results are later combined using a trivial union operation. The procedure is outlined in Fig. 2.

4.1.1 *Partial Summed-Volume Table Construction*

Since we are particularly interested in the binary information if a certain voxel is visible or not, we do not store fully classified alpha values inside the blocks, but only binary occupancy values. It is thus 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 or only once during a rendering session. In addition to the 16-bit partial SVT array, we store a copy of the not yet classified 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, perform classification for each voxel, and set the 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 partial SVT construction can be expected to completely happen in cache memory. 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 compute 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.

4.1.2 Performing Occupancy Queries on Partial Summed-Volume Tables

We reformulate the whole node splitting phase of the *k*-d tree construction algorithm 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. Fig. 2). We therefore initially determine which partial SVTs overlap the AABB of the node we want to subdivide. We then cull partial SVTs that are empty so we no longer have to consider them. Then we compute all the local AABBs inside the remaining partial SVTs. It is in general to be expected that the number of AABBs is rather low, because that number is proportional to the amount of partial SVTs under construction. This amount will also generally decrease with increasing recursion depth. Instead of using parallel reduction, we thus trivially combine the local AABBs to form a single AABB. Other than parallelizing the boundary computation, the node splitting phase of the algorithm outlined in Algorithm 1 remains unchanged compared to the serial algorithm.

Note that this approach results in computing lots of local AABBs that are not necessary for boundary computation. From an algorithmic point of view, it may be beneficial to first determine which partial SVTs overlap, but are not fully contained by the bounding box of the parent node, and later consider only these. Such an approach however turned out to be inferior with our tests. Just calculating all local AABBs and then later combining them resulted in the most intuitive implementation that exposed maximum scalability. The number of partial SVTs that are fully contained inside the parent node's AABB will also diminish with increasing recursion depth, which may further explain this observation.

Algorithm 1. Parallel Recursive Node Splitting Phase

```
procedure NodeSplitting(ParentBounds)
  for Each Candidate Plane $p$ do
    BoundsLeft ← InvalidAABB
    BoundsRight ← InvalidAABB
    for all PartialSVTs $s$ left of $p$ do in parallel
      $s$ ← Cull($s$, ParentBounds)
      LocalBounds ← CalculateLocal($s$)
      Combine(BoundsLeft, LocalBounds)
    end for
    for all PartialSVTs $s$ right of $p$ do in parallel
      $s$ ← Cull($s$, ParentBounds)
      LocalBounds ← CalculateLocal($s$)
      Combine(BoundsRight, LocalBounds)
    end for
  end for
  NodeSplitting(BoundsLeft)
  NodeSplitting(BoundsRight)
end procedure
```

4.2 Scalability Considerations

In order to assess the algorithm described before in terms of scalability on many-core systems like GPUs, we consider the two major phases of the algorithm: parallel SVT construction and parallel node splitting.

Scalability during the first phase is achieved by means of parallelizing over all partial SVTs, i.e. the number of blocks that the volume is subdivided into. By reducing the size (and thus increasing the number) of the blocks, scalability also increases. In addition to that, calculating each individual partial SVT can further be parallelized, e.g. by using parallel reduction to compute prefix sums.

We further consider the parallel node splitting phase from Algorithm 1. While the individual boundary computations are embarrassingly parallel, the top down recursion is not. Regarding a GPU implementation, one would either need to implement the recursion on the CPU and only *offload* the boundary computation to a GPU compute kernel, or to run a single thread on the GPU that generates new work on the fly. Either decision will result in pipeline stalls and possible communication overhead e.g. due to scheduling and executing compute kernels. We further note that prior work to parallelize recursive node splitting (e.g. by Karras [36]) is not easily applicable here because the number of resulting leaf nodes is rather low and also not known a priori. As we describe in the following section, a one to one adaptation of the multi-core CPU implementation for the GPU is thus not advantageous. We thus opt for a construction scheme with a recursive node splitting phase that is based on binning to reduce the number of times the parallel boundary operation is called. Binning will result in similar, but not exactly the same *k*-d trees that the serial and parallel implementations generate.

4.3 GPU Implementation with CUDA

Our GPU implementation uses the NVIDIA CUDA toolkit. The massive amount of threads available on current GPUs necessitates a different parallelization scheme that we

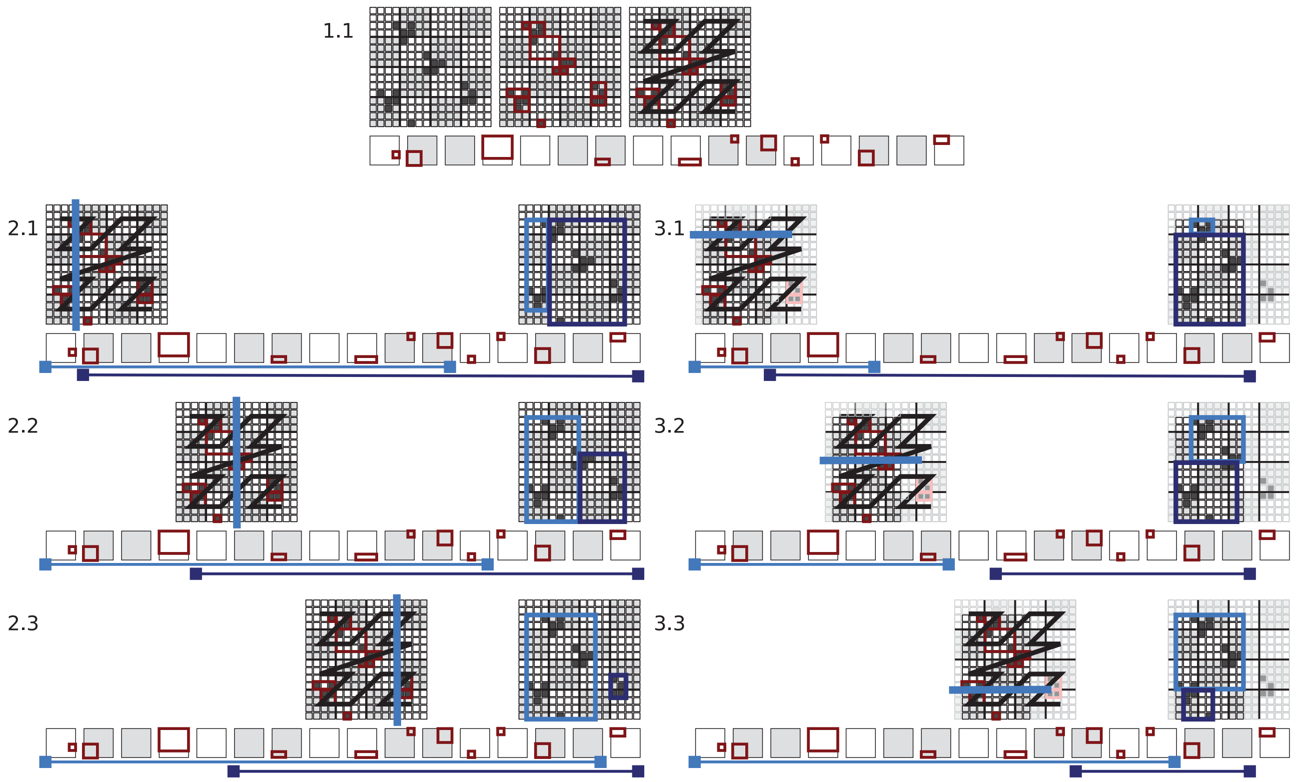


Fig. 3. *Adaptation of our multi-core CPU algorithm for GPUs.* The illustration shows the concepts in 2-d based on SATs, while the actual implementation incorporates the same concepts in 3-d. We first compute partial SVTs (1.1, first row, left) and then compute local AABBs (1.1, first row, middle). We insert invalid AABBs where blocks are not occupied at all, so there is an AABB associated with every block. When we have computed local AABBs, we immediately discard the partial SVTs. We then (1.1, first row, right) sort the AABBs on a z-order Morton curve. This results in the list of AABBs that is depicted in the second row of 1.1. We then perform recursive node splitting with binning. The illustration shows two recursion steps: first we split along the x-direction (2.1 through 2.3) to find that the result from 2.3 is optimal w.r.t. costs. We then show another recursion step (3.1 through 3.3) where the left box from 2.3 is split along the y-direction. Tight AABBs are found by combining the local AABBs that the respective parent nodes overlap using parallel reduce. Since the local AABBs are sorted on a z-order Morton curve, we can determine conservative ranges from the AABB list so that we do not have to reduce all local AABBs all the time. Bars below the respective lists (2.1 through 3.3, bottom rows) indicate which local boxes are to be considered to obtain the light and dark blue AABBs.

outline in the following. The implementation is based on a series of parallel reductions to keep the GPU execution units occupied as best as possible. Fig. 3 presents an overview of our algorithm.

While on the CPU the size of the partial SVTs are dictated by L1 cache size, on the GPU we decided to construct partial SVTs in the on-chip *shared memory* [37] of the *streaming multi-processors* (SM). On current GPUs, the shared memory size amounts to $48\ KB$. Another limit is imposed by the maximum number of threads that can be scheduled on an SM in a *thread block*, with the maximum on current GPUs being 1024. With these limits in mind, we opted for a partial SVT construction scheme that is inspired by the parallel scan operation described by Harris [23]. Prefix summations are performed using a blockwise parallel reduction algorithm in shared memory. The CUDA architecture is particularly prone to bank conflicts when accessing elements in shared memory, which the algorithm avoids with a carefully devised indexing scheme. Further care has to be taken regarding the memory access pattern of the scan operation to ensure cache locality. Bilgic et al. [26] used the algorithm described by Harris to construct *full* SATs. They first performed horizontal scan on the lines of the input image in one CUDA kernel. Then they transposed the image using a second CUDA kernel and performed horizontal scan on the transposed image, thus effectively calculating prefix sums for the (now reduced) columns of the input image. We employ a similar scheme, but with the difference that we only construct partial SVTs. It is thus desirable to not perform the transpose operation in *global memory* (i.e. off-chip GPU DDR3 memory), but rather in the shared memory of the SM. We thus scan the SVTs row-wise, then swap x and y-elements of each slice along the z-direction, perform another scan over all rows, then swap the x and z-elements along the y-direction, perform a last scan over all rows and finally swap all elements back to their original position (cf. Fig. 4).

In order to reach high utilization, we do not write the partial SVTs out to global memory at all, but immediately compute and store the local AABBs that we later combine during the recursive node splitting phase. This significantly simplifies the way that node splitting is implemented, but also implies that the AABB for an individual SVT is the same no matter if the candidate plane we test for is tangential or actually intersects the SVT's surrounding block. It is thus not meaningful to consider candidate planes that intersect partial SVTs, but only those planes that fall inbetween them. This obviously implies that the partial SVTs should be small in size. This assertion goes hand in hand with our observation from above that small partial SVTs are beneficial in order to maximize scalability. We therefore decided for a partial SVT size of 8^3 voxels. When computing local

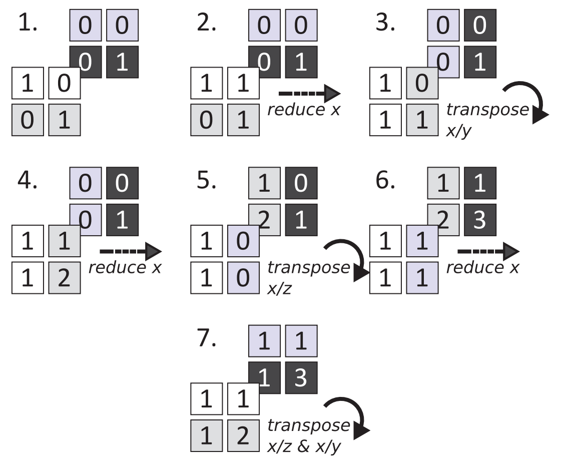


Fig. 4. *Construction of partial SVTs with our GPU implementation.* We in parallel construct partial SVTs of size 8^3. Inside each block we further parallelize by employing a sequence of parallel reductions and transpose operations in CUDA shared memory. The figure illustrates the procedure for partial SVTs of size 2^3.

boundaries, we first trivially cull empty SVTs, but assign an invalid AABB so that we later still have one bounding box per block. For non-empty blocks, we dedicate a single thread out of the thread group to determine the local AABB and employ binary search to fit the bounding box from the negative and positive x, y, and z directions. For 8^3 blocks, binary search implies three comparisons per direction, so that this operation is negligible performance-wise. The compute kernel then writes the local AABB out to global memory and returns execution to the CPU.

After computing local AABBs, we immediately sort them on a z-order Morton curve in global memory using a stable sorting algorithm. During recursive node splitting, we then sweep candidate planes on an eight voxel raster. Because the local AABBs are precalculated, we can simply combine them in global memory using parallel reduction. Since we previously sorted the local AABBs on a Morton curve, we do not have to reduce all AABBs all the time, but can conservatively determine a linear sequence of AABBs that fall inside the bounding box of the parent node. We therefore calculate the Morton codes of the parent node's AABB's minimum and maximum corner. The two Morton codes determine conservative lower and upper limits for indices into the list of local AABBs in global memory. See Fig. 3 for an illustration of this principle.

Sweeping candidate planes on an eight voxel raster rather than e.g. on a single voxel raster generally reduces the number of boundary computations (the process of finding tight AABBs around occupied regions of space). The fact that local AABBs are precalculated further simplifies the boundary computation, so that the overhead for calculating two single AABBs on either side of a plane subdividing a node is significantly reduced compared to the multicore CPU formulation of our algorithm. We however aim at further reducing the number of boundary computations by employing a binning approach: per node that we try to split, we consider only a fixed amount of candidate planes, no matter what the size of the side of the AABB is along which we perform the split. We still need to be careful that planes fall on an eight voxel raster. Sensible numbers of bins are e.g. 4, 8, or 16. With this approach, we can significantly reduce the number of overall boundary computations and thus the number of compute kernel executions.

4.3.1 Implementation Details

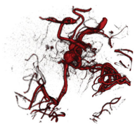
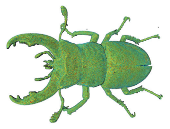
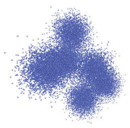
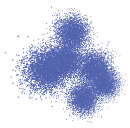
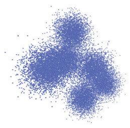
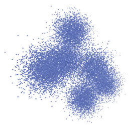
In order to compute partial SVTs, we employ a custom CUDA kernel that extends the method by Bilgic et al. [26] into 3-d but only calculates blockwise results. As outlined before, all operations in this kernel are performed in shared memory and the output of the kernel is a single, possibly invalid, AABB per block. For the ensuing combine operation we use `thrust::reduce()` from the GPU C++ template library Thrust [38] that ships with the CUDA toolkit. Special care has to be taken to reduce the memory allocation overhead incurred by this routine. By default, `thrust::reduce()` will allocate arrays in GPU global memory to hold temporary results using `cudaMalloc()` each time it is called and will immediately release the memory using `cudaFree()` after execution. It is however possible to implement a custom memory allocator and pass it to `thrust::reduce()`, so that it reuses previous memory allocations for temporary buffers. The arrays we reduce are generally small and the overhead for memory allocations as well as the impact of pipeline stalls can be significant. Reuse of memory allocations from previous `thrust::reduce()` calls turned out to have a significant impact on the performance of the parallel node splitting phase compared to an unoptimized version. The ability for the Thrust algorithms to consider custom allocation routines seems to have been added with CUDA version 8.0. With prior versions, our code would compile just fine but Thrust would silently ignore the custom allocator.

5 Results

For the evaluation we use the data sets listed in Table 1. While the first four data sets are publicly available, the remaining four data sets present the results of an *N*-body particle simulation that were sampled on uniform grids of size 256^3, 512^3, 1024^3, and 2048^3, respectively. We integrated our algorithms into the DVR library Virvo [39]. We compare both the construction speed as well as the rendering performance obtained with the parallel CPU and GPU algorithms. For a performance study that compares the serial and parallel versions of the CPU algorithm, we refer the interested reader to our previous publication [2].

Vidal et al. in their original publication asserted that it is generally beneficial to construct spatial indices with only few leaves that are traversed or sorted up front when the camera changes, and whose leaf nodes are inserted into a short linear list that is then iteratively traversed by a 3-d texture slicing-based volume renderer or by a simple ray marching volume renderer. Since control flow is generally allowed to be more complex on modern GPUs, and previous publications have shown that full ray / tree traversal is an efficient operation, it is interesting to evaluate if the assertion Vidal et al. made still holds today on contemporary hardware. Deeper trees will also cull more empty space than shallow ones. We therefore evaluate two ways to construct and render the *k*-d trees (cf. Fig. 5). We build *shallow* trees with the original criteria (i.e. the minimum size of each node is one-tenth of the size of the root node) that will only consist of a

TABLE 1
Performance Measurements for Various Data Sets

Dataset	Description	Construction (sec)		Rendering (FPS)			
		CPU	GPU	None	Grid	CPU	GPU
		Partial SVTs: 0.027	Partial SVTs: 0.001				
	Aneurism	**Shallow** (54.7 % culled)	**Shallow** (52.6 % culled)			**Shallow**	**Shallow**
	256 × 256 × 256	Splitting: 0.041, $\sum$ **0.068**	Splitting: 0.002, $\sum$ **0.003**			51.4	53.0
	Occupancy: 1.01 %	**Deep** (81.9 % culled)	**Deep** (82.6 % culled)			**Deep**	**Deep**
		Splitting: 0.110, $\sum$ **0.137**	Splitting: 0.198, $\sum$ **0.199**	41.3	41.5	58.0	59.8
			w/o empty space:	41.3	23.9	39.1	39.1
		Partial SVTs: 0.027	Partial SVTs: 0.001				
	Bonsai	**Shallow** (79.9 % culled)	**Shallow** (76.7 % culled)			**Shallow**	**Shallow**
	256 × 256 × 256	Splitting: 0.042, $\sum$ **0.069**	Splitting: 0.002, $\sum$ **0.003**			60.1	62.8
	Occupancy: 6.87 %	**Deep** (96.9 % culled)	**Deep** (97.5 % culled)			**Deep**	**Deep**
		Splitting: 0.056, $\sum$ **0.083**	Splitting: 0.098, $\sum$ **0.099**	39.2	51.8	71.5	69.6
			w/o empty space:	39.2	23.5	39.0	39.0
		Partial SVTs: 0.104	Partial SVTs: 0.006				
	Xmas Tree	**Shallow** (42.5 % culled)	**Shallow** (34.2 % culled)			**Shallow**	**Shallow**
	512 × 499 × 512	Splitting: 0.083, $\sum$ **0.187**	Splitting: 0.002, $\sum$ **0.008**			29.8	28.0
	Occupancy: 2.90 %	**Deep** (72.3 % culled)	**Deep** (70.5 % culled)			**Deep**	**Deep**
		Splitting: 0.298, $\sum$ **0.402**	Splitting: 0.551, $\sum$ **0.557**	24.7	28.7	38.2	37.1
			w/o empty space:	24.7	13.8	24.1	24.1
		Partial SVTs: 0.284	Partial SVTs: 0.015				
	Stag Beetle	**Shallow** (67.8 % culled)	**Shallow** (61.5 % culled)			**Shallow**	**Shallow**
	832 × 832 × 494	Splitting: 0.100, $\sum$ **0.384**	Splitting: 0.004, $\sum$ **0.019**			28.6	23.2
	Occupancy: 4.04 %	**Deep** (99.1 % culled)	**Deep** (98.4 % culled)			**Deep**	**Deep**
		Splitting: 0.291, $\sum$ **0.575**	Splitting: 0.766, $\sum$ **0.781**	21.4	25.4	38.5	35.3
			w/o empty space:	21.4	11.0	21.1	21.1
		Partial SVTs: 0.026	Partial SVTs: 0.001				
	N-Body (256^3)	**Shallow** (71.6 % culled)	**Shallow** (67.6 % culled)			**Shallow**	**Shallow**
	256 × 256 × 256	Splitting: 0.031, $\sum$ **0.057**	Splitting: 0.001, $\sum$ **0.002**			62.7	61.0
	Occupancy: 0.14 %	**Deep** (88.7 % culled)	**Deep** (87.1 % culled)			**Deep**	**Deep**
		Splitting: 0.066, $\sum$ **0.092**	Splitting: 0.118, $\sum$ **0.119**	41.1	50.2	66.7	65.8
			w/o empty space:	41.1	24.9	40.4	40.4
		Partial SVTs: 0.111	Partial SVTs: 0.006				
	N-Body (512^3)	**Shallow** (72.0 % culled)	**Shallow** (69.2 % culled)			**Shallow**	**Shallow**
	512 × 512 × 512	Splitting: 0.052, $\sum$ **0.163**	Splitting: 0.002, $\sum$ **0.010**			49.8	46.2
	Occupancy: 0.14 %	**Deep** (94.2 % culled)	**Deep** (93.4 % culled)			**Deep**	**Deep**
		Splitting: 0.270, $\sum$ **0.381**	Splitting: 0.510, $\sum$ **0.516**	25.8	40.3	63.0	59.9
			w/o empty space:	25.8	14.4	24.8	24.8
		Partial SVTs: 0.826	Partial SVTs: 0.048				
	N-Body (1024^3)	**Shallow** (54.0 % culled)	**Shallow** (69.0 % culled)			**Shallow**	**Shallow**
	1024 × 1024 × 1024	Splitting: 0.514, $\sum$ **1.340**	Splitting: 0.005, $\sum$ **0.053**			30.0	30.8
	Occupancy: 0.15 %	**Deep** (96.5 % culled)	**Deep** (91.4 % culled)			**Deep**	**Deep**
		Splitting: 1.331, $\sum$ **2.157**	Splitting: 1.002, $\sum$ **1.050**	14.3	30.9	54.4	47.0
			w/o empty space:	14.3	7.25	13.7	13.7
		Partial SVTs: 6.582	Partial SVTs: 0.398				
	N-Body (2048^3)	**Shallow** (72.2 % culled)	**Shallow** (68.9 % culled)			**Shallow**	**Shallow**
	2048 × 2048 × 2048	Splitting: 7.553, $\sum$ **14.14**	Splitting: 0.033, $\sum$ **0.431**			18.6	17.2
	Occupancy: 0.15 %	**Deep** (97.6 % culled)	**Deep** (93.5 % culled)			**Deep**	**Deep**
		Splitting: 11.89, $\sum$ **18.47**	Splitting: 3.324, $\sum$ **3.722**	7.04	24.4	44.2	35.6
			w/o empty space:	7.04	2.87	6.24	6.24

We build shallow trees based on the halting criteria proposed by Vidal et al., as well as deep trees with a minimum leaf node volume of 8^3 *voxels. We report the fraction of empty space culled by the leaf nodes, the time it takes to construct partial SVTs, and the time for the node splitting phase. We use four bins for the node splitting phase of the GPU implementation. We test the rendering performance of the respective trees by rendering orthographic, whole viewport filling images from several viewing angles at a resolution of* 2160 × 2160 *pixels. For our tests, we deactivate shading and early-ray termination. For comparison, we also report performance results for rendering without empty space skipping ("None") and for rendering with the grid data structure from OSPRay ("Grid"). The additional row labeled "w/o empty space" refers to the four tests performed with a transfer function that assigns no empty voxels.*

few nodes. We also test with *deep* trees where we allow the volume of a leaf to be at least 8^3 but impose no further restrictions. During rendering, we employ full tree traversal using a stack-based single ray algorithm in the spirit of the `while-while` traversal algorithm proposed by Aila and Laine [40]. While we evaluate construction on both the CPU and the GPU, rendering is implemented on the GPU with CUDA. For comparison, we report the rendering

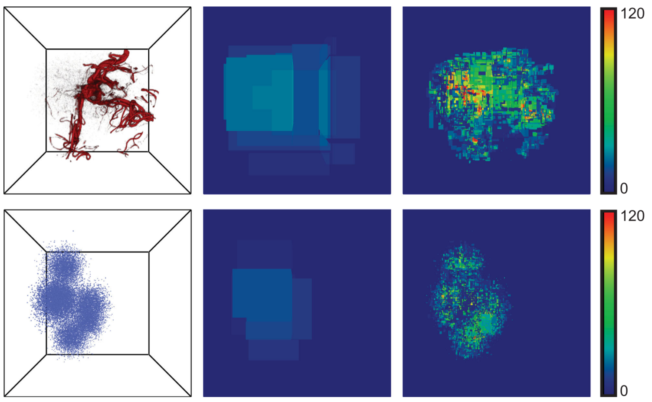


Fig. 5. *Shallow versus deep trees*. We conduct performance measurements with *shallow* trees where leaf nodes' volume is up to 10 percent of that of the root node (middle), and *deep* trees, where leaf nodes' maximum volume is 8^3 (right). The image shows the aneurism data set with 256^3 voxels and the *N*-body data set with a resolution of 512^3 voxels. The color coding indicates the number of tree nodes (both inner nodes and leaves) a primary ray encounters when marching from front to back through the volume.

performance obtained with simple ray marching without empty space skipping. We further reimplemented the open source grid accelerator from OSPRay [41] with CUDA and integrated it into our test framework. The grid accelerator uses a *min-max* range data structure, so that the construction time depends on the size of the transfer function array. OSPRay currently uses a simple 2-d lookup table with $O(n^2)$ construction complexity for that. While this data structure imposes that the alpha transfer function is one-dimensional and piecewise linear, recent work by Wald [42] at least suggests that construction can be sped up significantly. As there is currently no implementation available that incorporates this method, we however stick with the 2-d lookup table and omit construction time as this does not depend on the size of the data set. We run our tests on a graphics workstation equipped with two Intel Xeon Gold 5122 CPUs (four cores, eight threads each) with a clock frequency of 3.60 GHz (i.e. 16 threads per simultaneous multithreading on eight cores total), as well as an NVIDIA Titan V GPU. We deliberately use transfer functions that result in sparse configurations, but also test and report performance results for transfer functions that assign no empty voxels at all so we can assess the overhead of using tree traversal for volumes without empty space. We render images with 2160×2160 pixels (the vertical resolution of a 4K display), disable early-ray termination, and use a camera setup where the volume is completely visible inside the viewport and rendered with orthographic projection. We rotate the view in 2° steps around the three major cartesian axes and render 270 images obtained for each viewing angle. After a warmup phase to avoid undesired cache effects, we repeat this procedure 200 times and average the results. Performance numbers for the construction phase are obtained by repeatedly rebuilding the *k*-d tree. We measure and compute the average for 1,000 repetitions after performing 100 warmup iterations.

Our GPU implementation is based on binning. While it is obvious that using more bins would have an impact on the performance of the construction phase (the number of bins asymptotically relates to the number of GPU kernel calls), it is interesting to evaluate how the number of bins impacts tree quality and thus rendering performance. We thus contrast the performance of the node splitting phase (the SVT construction phase is independent of the number of bins) with the rendering performance for the eight data sets when building either shallow or deep trees. We report the results of this comparison in Fig. 6, where we construct trees with two to 32 bins. Note that since plane sweeping is performed on an eight voxel raster, for the 256^3 data sets large bin counts like 32 in practice only affect the root level of the tree.

Table 1 shows the overall performance of our implementation. Since the results from Fig. 6 indicate that rendering performance is not proportional to the number of bins, we report GPU construction results based on using four bins. We report performance results for both construction phases. We also state the occupancy, i.e. the percentage of voxels that are non-empty, and the ratio of empty space that is culled by the leaves of the respective trees (i.e. $\frac{\#\ culled\ voxels}{\#\ empty\ voxels}$). It is also interesting if using the optimization based on Morton curves has a significant impact on the performance of the node splitting phase. We therefore performed a comparison based on the four *N*-body data sets with different grid resolutions and report timing results in Fig. 7.

Our results indicate that *k*-d trees, no matter if they are shallow or deep, generally outperform simple acceleration data structures like the structured grid from OSPRay. In addition, traversal overhead as compared to grid traversal is negligible if the volume is not sparse. Anyhow, the effectiveness of shallow trees is generally below that of deep trees, where the better culling properties outweigh the higher traversal costs. This is becoming even more apparent the larger the spatial extent of the data set; in the case of the 2048^3 data set, the shallow tree construction setup results in a rendering performance significantly below the performance that can be achieved with the OSPRay grid, while with the deep tree construction setup favorable results can be achieved. It is generally noteworthy that shallow trees build extremely fast. In the future we would like to investigate if a hybrid data structure combining shallow *k*-d tree nodes with structured grids that cull empty space at the leaf node level can outperform the deep tree setup w.r.t. construction and rendering performance.

6 Discussion

The two implementations we presented both use a top down construction approach. While a parallel version of the serial CPU algorithm was straightforward to implement, a reasonably fast GPU version that resembles the original algorithm turned out to be hard to implement; an exact reimplementation of the CPU algorithm for the GPU will perform orders of magnitude worse than a parallel and optimized CPU implementation. In the context of surface ray tracing, many fast GPU *k*-d tree or BVH construction algorithms employ a bottom up approach and perform a sequence of $O(n)$ operations on the primitive or leaf level. Algorithms like LBVH by Lauterbach et al. [31] are known to construct spatial indices in real-time whose quality is however inferior compared to trees constructed with the SAH. Our choice to also use a top down construction scheme on the GPU was led by reasons of comparability with the multi-core CPU implementation. The GPU top down construction algorithm we proposed

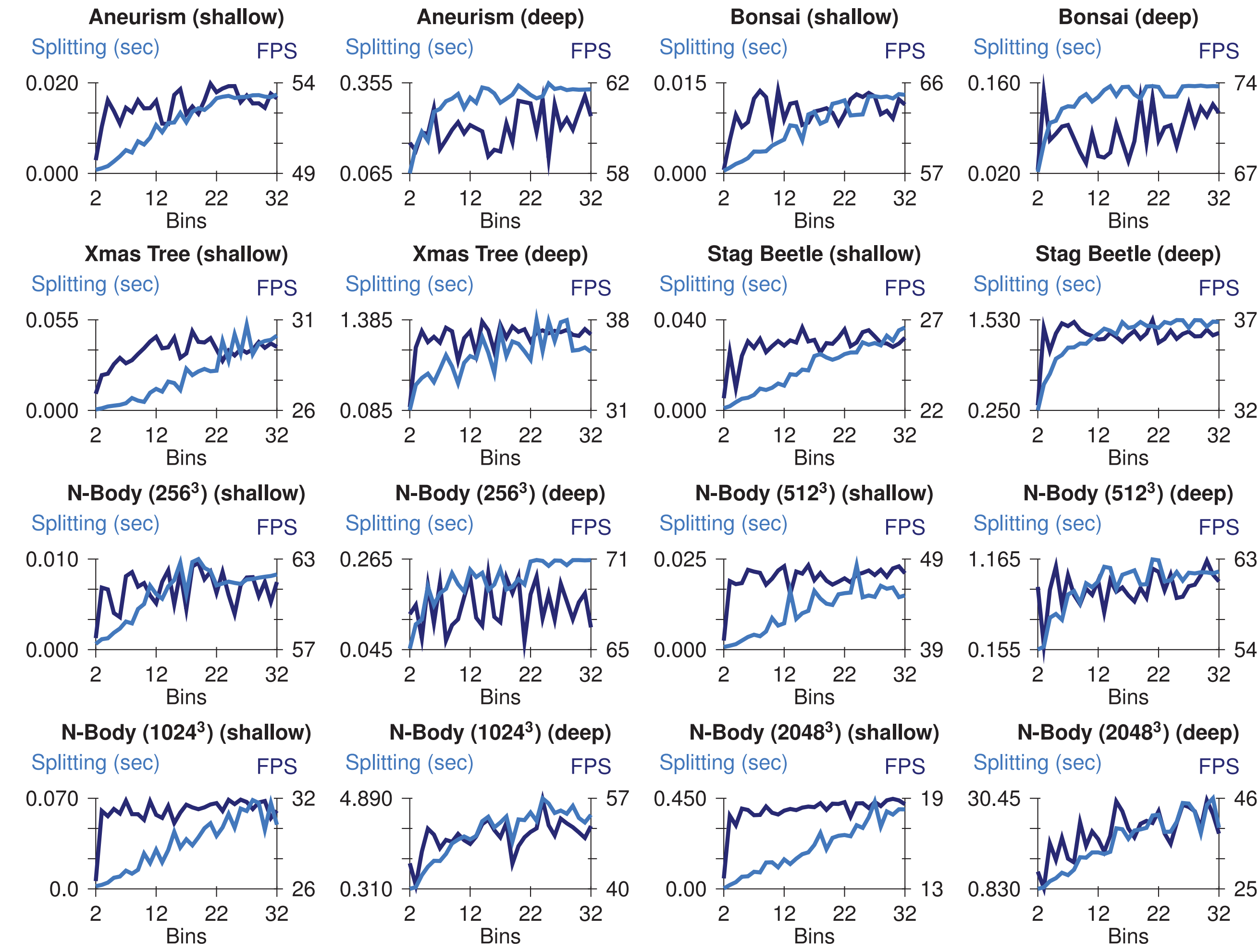


Fig. 6. *Effect of the number of bins on the performance of the node splitting phase on the GPU.* We found that setups using fewer than four bins resulted in inferior trees and thus lower rendering performance. Above that limit, an increased bin count would however generally not result in trees with higher quality.

compares favorably to the CPU construction algorithm but is still highly dependent on the depth of the constructed trees. While the halting criteria originally proposed by Vidal et al. result in very fast tree construction, deeper trees that take longer to construct are able to cull more empty space and thus have better rendering performance.

Our results further indicate that binning is not generally detrimental to rendering performance. On the contrary, on certain occasions the *greedy* top down tree construction algorithm would even find better *global* results when binning was used as opposed to sweeping on an eight voxel raster. As the node splitting performance asymptotically relates to the number of tested splitting planes, we believe that binning would be generally preferable even on the CPU. Our

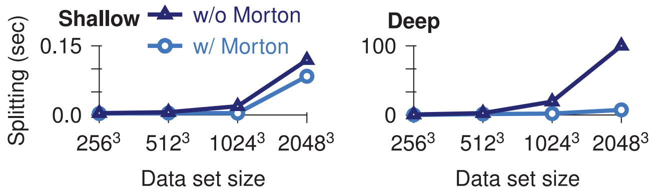


Fig. 7. *Impact of the z-order Morton curve optimization* from Fig. 3 on construction time for the four *N*-body simulation data sets. The performance impact is most noticeable for large data sets and when building deep trees with many small leaves.

results corroborate this assumption as the *relative* node splitting overhead on the GPU decreases the larger the spatial extent of the data set.

Another positive side effect of the binning algorithm is its relatively low temporary memory consumption. While the CPU variant that does not use binning and explicitly stores the partial SVTs in main memory, partial SVTs are only temporarily stored in CUDA shared memory, which is immediately released after local AABBs have been computed. Storing partial SVTs requires storing an extra copy of the volume with 16 bit precision. With the binning algorithm, 8^3 blocks are represented by a single local AABB that can be stored (including alignment) with 32 bytes. As we store a (potentially empty) local AABB for each 8^3 block, the temporary storage overhead for e.g. 2048^3 data sets is thus only 512 MB.

7 Conclusion

We presented two implementations of a parallel algorithm to fully rebuild *k*-d trees for spatial indexing of sparse volumes, one targeting multi-core CPU systems, and the other one targeting GPUs. The adapted GPU implementation employs binning, local AABB precalculation and z-order Morton curves for fast retrieval of neighboring blocks to optimize the recursive node splitting phase of the multi-core CPU algorithm. The GPU algorithm does not produce the exact same *k*-d trees as the CPU algorithm does, but enables far better scalability. Our performance study revealed that tree construction on the GPU compares well to tree construction on the CPU. It however also revealed that the original halting criteria by Vidal et al. favoring shallow trees result in poor quality of the spatial index if the spatial extent of the data set is large. Since the heuristic we use is greedy, binning will generally not result in inferior trees compared to the ones produced with the original algorithm. With binning, building deeper trees is affordable because the number of potential split positions depends on the bin count and is not directly asymptotical to the number of voxels.

Our approach employs a top down construction scheme. We used top down construction so that we can still compare our algorithm with the CPU implementations from our previous paper. In the future we want to compare our algorithms with bottom up *k*-d tree and BVH construction algorithms like they are e.g. used for surface ray tracing on the GPU. We believe that our algorithm will compare favorably to bottom up construction because the heuristic we use is comparable to the SAH for surfaces, and because tree depth can be used as a variable to either provide better tree construction or better rendering performance.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. We would also like to thank the researchers and institutes who provide the publicly available data sets that we used for the evaluation.

References

[1] V. Vidal, X. Mei, and P. Decaudin, "Simple empty-space removal for interactive volume rendering," *J. Graph. Tools*, vol. 13, no. 2, pp. 21–36, 2008.

[2] S. Zellmann, J. P. Schulze, and U. Lang, "Rapid k-d tree construction for sparse volume data," in *Proc. Eurographics Symp. Parallel Graph. Vis.*, 2018, pp. 69–77.

[3] W. T. Correa, J. T. Klosowski, and C. T. Silva, "Visibility-based prefetching for interactive out-of-core rendering," in *Proc. IEEE Symp. Parallel Large-Data Vis. Graph.*, 2003, p. 2.

[4] M. Weiler, R. Westermann, C. Hansen, K. Zimmermann, and T. Ertl, "Level-of-detail volume rendering via 3d textures," in *Proc. IEEE Symp. Vol. Vis.*, 2000, pp. 7–13.

[5] W. Li, K. Mueller, and A. Kaufman, "Empty space skipping and occlusion clipping for texture-based volume rendering," in *Proc. IEEE Vis.*, Oct. 2003, pp. 317–324.

[6] R. Kähler, M. Simon, and H.-C. Hege, "Interactive volume rendering of large data sets using adaptive mesh refinement hierarchies," *IEEE Trans. Vis. Comput. Graph.*, vol. 9, no. 3, pp. 341–351, 2003.

[7] D. Ruijters and A. Vilanova, "Optimizing GPU volume rendering," *Winter School Comput. Graph.*, vol. 14, pp. 9–16, Feb. 2006.

[8] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering," in *Proc. ACM SIGGRAPH Symp. Interactive 3D Graph. Games*, Feb. 2009, pp. 15–22.

[9] B. Liu, G. J. Clapworthy, F. Dong, and E. C. Prakash, "Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 19, no. 10, pp. 1732–1745, Oct. 2013.

[10] J. Beyer, M. Hadwiger, T. Möller, and L. Fritz, "Smooth mixed-resolution GPU volume rendering," in *Proc. 5th Eurographics/IEEE VGTC Conf. Point-Based Graph.*, 2008, pp. 163–170.

[11] I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen, "Interactive isosurface ray tracing of time-varying tetrahedral volumes," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 6, pp. 1727–1734, Nov. 2007.

[12] A. Knoll, S. Thelen, I. Wald, C. Hansen, H. Hagen, and M. Papka, "Full-resolution interactive CPU volume rendering with coherent BVH traversal," in *Proc. IEEE Pacific Vis.*, 2011, pp. 3–10.

[13] R. Yagel and Z. Shi, "Accelerating volume animation by space-leaping," in *Proc. IEEE Conf. Vis.*, Oct. 1993, pp. 62–69.

[14] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agos, and H. Pfister, "SparseLeap: Efficient empty space skipping for large-scale volume rendering," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 974–983, Jan. 2018.

[15] Y. Wang, W. Dou, and J. M. Constans, "Accelerating volume ray casting by empty space skipping used for computer-aided therapy," in *Proc. Int. Conf. Audio Lang. Image Process.*, Jul. 2012, pp. 661–667.

[16] M. Labschütz, S. Bruckner, M. E. Gröller, M. Hadwiger, and P. Rautek, "JiTTree: A just-in-time compiled sparse GPU volume data structure," *IEEE Trans. Vis. Comput. Graph.*, vol. 22, no. 1, pp. 1025–1034, Jan. 2016.

[17] J. Schneider and P. Rautek, "A versatile and efficient GPU data structure for spatial indexing," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 911–920, Jan. 2017.

[18] J. Beyer, M. Hadwiger, and H. Pfister, "A Survey of GPU-Based Large-Scale Volume Visualization," in *EuroVis - STARs*, R. Borgo, R. Maciejewski, and I. Viola, Eds. Aire-la-Ville, Switzerland: The Eurographics Association, 2014.

[19] F. C. Crow, "Summed-area tables for texture mapping," *SIGGRAPH Comput. Graph.*, vol. 18, no. 3, pp. 207–212, Jan. 1984.

[20] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for GPU computing," in *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS Symp. Graph. Hardware*, 2007, pp. 97–106.

[21] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proc. 22nd Annu. Int. Conf. Supercomputing*, 2008, pp. 205–213.

[22] A. Davidson and J. Owens, "Toward techniques for auto-tuning GPU algorithms," in *Proc. 10th Int. Conf. Appl. Parallel Sci. Comput. - Vol. 2*, 2012, pp. 110–119.

[23] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Reading, MA, USA: Addison-Wesley, Aug. 2007, ch. 39, pp. 851–876.

[24] A. Kasagi, K. Nakano, and Y. Ito, "Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations," in *Proc. 43rd Int. Conf. Parallel Process.*, Sep. 2014, pp. 251–260.

[25] A. Papatriantafyllou and D. Sacharidis, "High performance parallel summed-area table kernels for multi-core and many-core systems," in *Proc. 22nd Int. Conf. Euro-Par: Parallel Process. - Vol. 9833*, 2016, pp. 306–318.

[26] B. Bilgic, B. Horn, and I. Masaki, "Efficient integral image computation on the GPU," in *Proc. Intell. Veh. Symp.*, 2010, p. 528–533.

[27] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM Trans. Graph.*, vol. 30, no. 6, pp. 176:1–176:12, Dec. 2011.

[28] J. Díaz, P.-P. Vázquez, I. Navazo, and F. Duguet, "Real-time ambient occlusion and halos with summed area tables," *Comput. Graph.*, vol. 34, no. 4, pp. 337–350, 2010.

[29] I. Wald, C. Benthin, and P. Slusallek, "Distributed interactive ray tracing of dynamic scenes," in *Proc. IEEE Symp. Parallel Large-Data Vis. Graph*, Oct. 2003, pp. 77–85.

[30] I. Wald, "On fast construction of SAH-based bounding volume hierarchies," in *Proc. IEEE Symp. Interactive Ray Tracing*, 2007, pp. 33–40.

[31] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 375–384, 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1111/j.1467-8659.2009.01377.x

[32] J. Pantaleoni and D. Luebke, "HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry," in *Proc. Conf. High Perform. Graph.*, 2010, pp. 87–95. [Online]. Available: http://dl.acm.org/citation.cfm?id=1921479.1921493

[33] L. R. Domingues and H. Pedrini, "Bounding volume hierarchy optimization through agglomerative treelet restructuring," in *Proc. 7th Conf. High-Perform. Graph.*, 2015, pp. 13–20. [Online]. Available: http://doi.acm.org/10.1145/2790060.2790065

[34] D. Meister and J. Bittner, "Parallel locally-ordered clustering for bounding volume hierarchy construction," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 03, pp. 1345–1353, Mar. 2018.

[35] Y. Gu, Y. He, K. Fatahalian, and G. Blelloch, "Efficient BVH construction via approximate agglomerative clustering," in *Proc. 5th High-Perform. Graph. Conf.*, 2013, pp. 81–88. [Online]. Available: http://doi.acm.org/10.1145/2492045.2492054

[36] T. Karras, “Maximizing parallelism in the construction of BVHs, octrees, and k-d trees,” in *Proc. 4th ACM SIGGRAPH/Eurographics Conf. High-Perform. Graph.*, 2012, pp. 33–37. [Online]. Available: https://doi.org/10.2312/EGGH/HPG12/033–037

[37] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[38] N. Bell and J. Hoberock, “Thrust: A productivity-oriented library for CUDA,” in *GPU Computing Gems Jade Edition*, W.-m. W. Hwu, Ed. San Francisco, CA, USA: Morgan Kaufmann, 2011, ch. 26, pp. 359–373.

[39] J. Schulze, U. Woessner, S. Walz, and U. Lang, “Volume rendering in a virtual environment,” in *Proc. 7th Eurographics Conf. Virtual Environ. 5th Immersive Projection Technol.*, 2001, p. 187–198.

[40] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on GPUs,” in *Proc. Conf. High Perform. Graph.*, 2009, pp. 145–149. [Online]. Available: http://doi.acm.org/10.1145/1572769.1572792

[41] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gnther, and P. Navratil, “OSPRay - a CPU ray tracing framework for scientific visualization,” *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 931–940, Jan. 2017.

[42] I. Wald, “Computing minima and maxima of subarrays,” in *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, E. Haines and T. Akenine-Möller, Eds. Berkeley, CA, USA: Apress, 2019, pp. 61–70.

Stefan Zellmann received the graduate degree in information systems from the University of Cologne, the doctor’s degree in computer science in 2014, with a PhD thesis on high performance computing and direct volume rendering. He is with the Chair of Computer Science, University of Cologne since 2009. His research interests include real-time ray tracing, high performance scientific visualization, as well as GPGPU and coprocessor architecture and programming. He is a member of the IEEE.

Jürgen P. Schulze received the MS degree from the University of Massachusetts and the PhD degree from the University of Stuttgart, Germany. He is an associate research scientist with UCSD’s Qualcomm Institute, and an associate adjunct professor with the Computer Science Department, where he teaches computer graphics and virtual reality. His research interests include applications for virtual and augmented reality systems, 3D human-computer interaction, and medical data visualization. He is the director of the Immersive Visualization Laboratory, UCSD’s Qualcomm Institute.

Ulrich Lang received the graduate degree from the University of Stuttgart as Dr.-Ing. He is a chair of computer science with the University of Cologne since 2004. Before that, he was deputy director of HLRS, a German national supercomputing Center in Stuttgart, where he also headed the visualization department. At HLRS he coordinated the development of COVISE, a COllaborative VIsualization and Simulation Environment, that supports, e.g. the analysis of simulation results in virtual environments.