# JiTTree: A Just-in-Time Compiled Sparse GPU Volume Data Structure

Matthias Labschütz, Stefan Bruckner, M. Eduard Gröller, Markus Hadwiger, and Peter Rautek

**Abstract**—Sparse volume data structures enable the efficient representation of large but sparse volumes in GPU memory for computation and visualization. However, the choice of a specific data structure for a given data set depends on several factors, such as the memory budget, the sparsity of the data, and data access patterns. In general, there is no single optimal sparse data structure, but a set of several candidates with individual strengths and drawbacks. One solution to this problem are hybrid data structures which locally adapt themselves to the sparsity. However, they typically suffer from increased traversal overhead which limits their utility in many applications. This paper presents JiTTree, a novel sparse hybrid volume data structure that uses just-in-time compilation to overcome these problems. By combining multiple sparse data structures and reducing traversal overhead we leverage their individual advantages. We demonstrate that hybrid data structures adapt well to a large range of data sets. They are especially superior to other sparse data structures for data sets that locally vary in sparsity. Possible optimization criteria are memory, performance and a combination thereof. Through just-in-time (JIT) compilation, JiTTree reduces the traversal overhead of the resulting optimal data structure. As a result, our hybrid volume data structure enables efficient computations on the GPU, while being superior in terms of memory usage when compared to non-hybrid data structures.

**Index Terms**—Data Transformation and Representation, GPUs and Multi-core Architectures, Volume Rendering

## 1 Introduction

The display and processing of large volume data poses a challenge due to high computational complexity and memory demands. Memory limitations have become the predominant bottleneck for volumetric data processing and visualization on recent GPU hardware. In practice, however, many types of volume data sets are sparse. Considering their data characteristics such as the regularity and distribution of values within the volume, they can be represented in a more efficient manner than by common regular grids.

Many different approaches have been proposed for handling sparse volumetric data. For example, bricking techniques and tree representations, such as the octree, kd-tree or $N^3$ tree, provide a hierarchical solution for data sets of relatively low sparsity. For data sets of medium sparsity, spatial hashing makes more efficient access and storage possible. Extremely sparse data can be efficiently represented and accessed via binary search in sorted voxel lists.

In general, sparse volume data structures result in a tradeoff between memory efficiency and access performance as more complex memory layouts typically require more costly look-up code, resulting in reduced runtime performance.

In this paper we present JiTTree, a new type of sparse volume data representation that enables memory-efficient storage and high-performance data access. In contrast to other data structures, JiTTree enables the generation of just-in-time compiled GPU code. It automatically adapts to local data properties and it avoids the performance drawbacks of conventional hybrid data structures. Our approach optimizes the memory layout and the corresponding GPU code using a novel on-the-fly optimization scheme. We flexibly support the combination of several existing sparse data structures to minimize memory consumption while enabling high runtime performance. To the best of our knowledge, this is the first volumetric data structure that simultaneously adapts memory layout *and* traversal code in a just-in-time manner. We demonstrate that our approach is superior to other data structures in terms of memory consumption for most practical data sets while enabling high-performance look-ups for many common volume processing tasks.

- *Matthias Labschütz is with KAUST, E-mail: mlabschuetz@cg.tuwien.ac.at.*
- *Stefan Bruckner is with University of Bergen, E-mail: stefan.bruckner@uib.no.*
- *M. Eduard Gröller is with TU Wien and VrVis Research Center, E-mail: mailto:meister@cg.tuwien.ac.at.*
- *Markus Hadwiger is with KAUST, E-mail: markus.hadwiger@kaust.edu.sa.*
- *Peter Rautek is with KAUST, E-mail: peter.rautek@kaust.edu.sa.*

*Manuscript received 31 Mar. 2015; accepted 1 Aug. 2015; date of publication xx Aug. 2015; date of current version 25 Oct. 2015. For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.*

## 2 Related Work

An overview of spatial data structures is given in the book by Samet [23]. The survey by Gaede and Günther [7] includes the concept of spatial hashing, which is a one or two level indirection in the most basic case. Both of these surveys only consider CPU based data structures.

Hierarchical trees were introduced for representing two dimensional and three dimensional data. Quad-trees [5] lead to octrees and multi-dimensional binary search trees (kd-trees) [3]. The latter were used as a compact volume representation for ray tracing, first on the CPU and later on the GPU.

Variants of octrees are discussed in the survey by Knoll [14]. Linear octrees (as introduced for quad-trees by Gargantini [8]) store all values in a specific order. No internal nodes are kept. Therefore these octrees require a fixed size per entry, independently of the sparsity of the data. Pointer octrees allow sparse data to be represented in a more compact way: a list of pointers and a list of values are stored in two separate arrays. First GPU implementations of sparse pointer octrees are found in the work of Beosnon and Davis [2], which uses octrees for surface texture encoding. More recently Crassin et al. [4] uses a pointer octree for volume rendering. These approaches provide an adaptive representation for a whole volume. Nevertheless, for a densely populated volume (or region), a pointer octree introduces an overhead due to its internal nodes. Additionally, the look-up into an octree causes a performance overhead for the indirections when traversing the octree to the leaves.

Recently, algorithms were formulated to generate octrees on the GPU. For instance, Lauterbach et al. [15] presented an efficient bottom-up generation scheme. Starting from a list of entries they sort the nodes depending on their Morton code. An algorithm for parallel octree construction that uses a full volume to construct a sparse octree was described by Ziegler [29]. The basis of the octree construction algorithm are *histogram pyramids* [30] also used in the voxel list generation. We have based our voxel list and octree generation on this parallel algorithm.

Voxel list access was improved by using *spatial hashing* or *binary search*. *Perfect spatial hashing* [16] provides faster access for sparse data than an octree. Nevertheless, the generation of a suitable hash function and an offset table can be computationally demanding.

Binary search provides an alternative to spatial hashing for a small number of elements. The only requirement for binary search is a sorted list. To efficiently sort a list, *voxel radix sort* by Satish et al. [24] can be used. Histogram pyramids [30] provide an alternative to sort the non-zero points in a volume in Morton order.

Bricking on the CPU was used to improve cache coherency by Grimm et al. [9]. Kähler et al. [13] use a CPU based kd-tree on top of a bricking approach to render adaptive mesh refinement data. They render the bricks sequentially in back-to-front order. Ruijters et al. [22] use bricking on the GPU for volume rendering. To reduce the workload on the GPU, they generate a CPU side octree for each brick. The octree is traversed with a threshold for sampling the brick data which is used in slice based rasterization. Virtual memory management [10] can be used to complement our brick-based data structure as well, but it is an orthogonal concept to compact sparse representations.

Glift [17] provides an abstraction layer to build data structures on top of OpenGL. The library was used to formulate a dynamic, sparse, adaptive structure for shadow maps and 3D painting. Glift uses an adaptive bricking approach on the GPU. The introduction of OpenCL and CUDA allows developers to build such data structures more easily. Just-in-time compilation makes it possible to dynamically allocate the required memory on the GPU and to remove unnecessary code.

Our work is based on the partial evaluation of programs that is described by Jones et al. [12]. We partially evaluate the GPU programs for performance reasons but also to make our data structure more flexible. More recently Rompf et al. [20, 21] and Sujeeth et al. [25] have described a lightweight modular staging approach that enables the deferral of parts of the program compilation to a later stage. The reason is that at a late stage the compiler is aware of the data and can thereby generate optimized programs. We show how such a data aware compilation can be used to generate an efficient sparse volume data structure.

## 3 JiTTree Overview

A JiTTree is a high performance memory efficient volumetric representation with access functions that are tailored to one specific data set. The main concept of the JiTTree is the just-in-time compilation stage that transforms data into efficient access functions. Thereby it enables the use of multiple kinds of data structures to represent one volume, without introducing a significant traversal overhead. For instance a JiTTree can store some of the data using an octree, while other parts of the data are represented as dense grids. We call these data structures (octree, dense grid, etc.) the *elemental data structures* to distinguish them from the hybrid data structure that combines them. In general a hybrid data structure consists of elemental data structures, to locally better represent a data set. The more elemental data structures are used the higher is the potential that the memory efficiency increases. However, with conventional approaches more elemental data structures mean a higher overhead during data traversal. Instead of depending on the number of elemental data structures, the traversal overhead of a JiTTree depends on the homogeneity of the data set. We achieve this by just-in-time compiling traversal code that is tailored to a specific data set and its optimal hybrid representation. Rompf et al. [20] show that such a data aware compilation stage can be used to optimize performance.

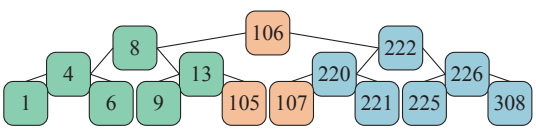



Fig. 1: Binary search tree example. Color encodes the data type.

To illustrate the key concept of our data structure we consider the example of a binary search tree. Figure 1 shows the binary tree of a sorted list. During look-up the tree is traversed from top to bottom. When the key is found the index of the element in the compact array is returned.

Similarly to our sparse volume approach, a binary search tree enables efficient access to a sparse array. Let $a$ be an array of 15 elements at the positions 1, 4, 6, 8, 9, 13, 105, 106, 107, 220, 221, 222, 225, 226, 308. A dense representation of $a$ would allocate memory that is large enough to fit 309 elements. A sparse representation of $a$ stores the 15 elements in a compact array $a_{comp}$ thereby allocating just enough memory for the 15 elements. Additionally, a search tree is recursively built for indexing the elements of the compact array. At each iteration of the search tree construction, the median element becomes the root node of the two new sub-lists.

```
// recursive find function with look-up key k
// k is assumed to exist in the tree
uint find(key, node)
{ // compare key to node key
  if(key < node.key)
    // traverse left subtree
    return find(key, node.left);
  else if(key > node.key)
    // traverse right subtree
    return find(key, node.right);
  else return node.index;
}
```

Listing 1: Recursive look-up function pseudo code

Listing 1 shows the pseudo code of the recursive tree traversal. The search key is compared to the root node. If the search key is smaller than the current node's key, the traversal continues in the left sub-tree. If it is greater the search continues in the right sub-tree. If the key is equal the search is successful and returns the index [0..14] of the compact array. The search tree enables the use of a sparse array which makes it possible to remove the empty elements in the sorted list. Thereby, it greatly saves storage space. Traversal is efficient but nevertheless incurs an overhead in comparison to a dense array. The *node.key* instructions in Listing 1 are the memory fetches that are introduced for tree traversal, and the *node.index* instruction is the memory fetch that is introduced to retrieve the index into the compact array. The just-in-time transformation of such a tree directly incorporates the data of the tree into the code. It thereby eliminates the need for memory fetches and greatly reduces traversal overhead. Listing 2 shows the pseudo code of the just-in-time compiled search tree.

```
// jit find function with look-up key k
// k is assumed to exist in the tree
uint find(key)
{ // compare key to root key
  if(key < 106) // traverse left subtree
    if(key < 8) // traverse left-left subtree
      if(key < 4) return 0;
      else if(key > 4) return 2;
      else return 1;
    if(key > 8) // traverse left-right subtree
      if(key < 13) return 4;
      else if(key > 13) return 6;
      else return 5;
    else return 3;
  else if(key > 106) // traverse right subtree
    // code omitted for brevity
    ...
  else return 7;
}
```

Listing 2: JiTTree look-up function pseudo code

By explicitly compiling the data into the code the access becomes less memory-bound during tree traversal. This is beneficial for many data processing algorithms.

For our case of multiple elemental data structures the compact array $a_{comp}$ needs to store types and pointers. The types are encoded as colors in the example of Figure 1. If we want to execute a function $foo$ on the elements we first have to look-up their type and then route the function call to the specialized function of that type. In case of the recursive look-up the execution of a function is shown in Listing 3.

```
// function foo uses recursive traversal
// type specialization via switch statement
value foo(key, root)
{ // tree traversal to get index
  uint index = find(key, root);
  // compact array lookup
  element e = a_comp[index];
  // specialize function call by element type
  switch(e.type) {
    case green: return foo_green(e.address);
    case orange: return foo_orange(e.address);
    case blue: return foo_blue(e.address);
  }
}
```

Listing 3: foo member function evaluation pseudo code

Through JIT compilation we inline the traversal code, the type specialization and the address look-up into the foo function. Listing 4 shows the result of JIT compiling the member function foo of the elements in the sparse array.

```
// function foo inlined traversal
// type specialization done at jit compilation
value foo(key, root)
{ // compare key to root key
  if(key < 106) // traverse left subtree
    if(key < 8) // traverse left-left subtree
      // call specialized function
      if(key < 4) return foo_green(addr_0);
      else if(key > 4) return foo_green(addr_2);
      else return foo_green(addr_1);
    if(key > 8) // traverse left-right subtree
      if(key < 13) return foo_green(addr_4);
      else if(key > 13) return foo_orange(addr_6);
      else return foo_green(addr_5);
    else return foo_green(addr_3);
  else if(key > 106) // traverse right subtree
    // code omitted for brevity
    ...
  else return foo_orange(addr_7);
}
```

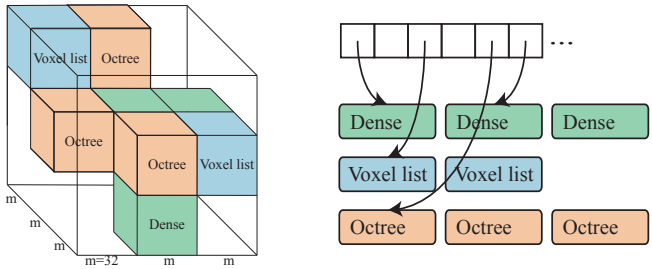
Listing 4: foo JIT function evaluation pseudo code

The overhead to resolve types is eliminated and specialized functions can directly be called. These same concepts apply to the JiTTree. It eliminates memory fetches in the top level traversal code and inlines type specialization. This eliminates type resolving as well as the switch statements that route execution between the different elemental data structures. Therefore, the access time to one node is independent of the number of different elemental node types. As a consequence the JiTTree can have an arbitrary number of different kinds of sparse data structures that improve the overall quality of the JiTTree without introducing new memory fetches. This is an important property of a hybrid data structure that was to the best of our knowledge never achieved before.

In essence the JiTTree decreases memory requirements by increasing program length. GPU algorithms are known to be either memory-bound (i.e., the performance is limited by the number of memory fetches), compute-bound (i.e., the performance is limited by the number of arithmetic computations, also called high computational density), or instruction-throughput-bound (i.e., the performance is limited by the non-arithmetic instructions) [28]. The just-in-time compilation is an attempt to shift some of the burden from the GPU's memory cache to the instruction cache making the overall algorithm less memory-bound.

## 4 Data Structure Layout

The basic principle of our data structure is to adapt to the local sparsity of a specific data set. Other volume representations often make the distinction between dense and empty (or homogenous) regions and treat these regions differently. We aim to locally better adapt to a certain level of sparsity of the data than these representations by introducing more fine grained distinctions. A data set typically contains many subregions with different sparsity characteristics. Each of the regions can be optimally encoded by one data representation. Very sparse regions are for instance better represented by a voxel list, medium sparse regions lend themselves best to hash tables [16] and octrees, while dense regions are most efficiently stored in a dense representation. For our implementation of the JiTTree we chose a combination of four different representations: empty, voxel list, octree, dense. Each type performs best for a different level of sparsity. We describe these *elemental node types* from highest to lowest sparsity in Section 4.1.




(a) Volume (b) Pointer structure

Fig. 2: (a) shows a volume that is split into $3^3$ bricks each of size $m^3$. Each brick holds one of the elemental data structures: dense, voxel list, octree or is an empty region. In (b) the basic data layout is shown. The volume consists of a list of bricks which point to elemental leaf nodes.

To support local adaptation our data structure is designed as a pointer based tree. Figure 2 shows an example for our data layout. Similar to volume bricking approaches [26] we divide the data into bricks. In contrast to other bricking schemes our data structure allows every brick to be represented by a different data structure. The goal is to pick the optimal representation for each brick while globally optimizing brick size, as well as the layout of the data structure. In Section 6 we describe the implementation of the root level in the bricking hierarchy.

### 4.1 Elemental node types

The memory optimal data structure for one brick of a volume depends on the position and the number of non-zero entries. In this work we consider three different data structures which form, together with the empty brick type, the four elemental node types:

- Empty volume: Empty bricks are simply omitted when the data is uploaded to the GPU.
- Voxel list: A voxel list is represented as a set of quadruples $(x, y, z, value)$. Each of the $x, y, z$ coordinates theoretically requires $log_2(m)$ bits where $m$ is the side length of the volume. Voxel lists are memory efficient for extremely sparse volumes. A major drawback of voxel lists is that access to the data requires $O(n)$ steps,

Table 1: Theoretical comparison of the properties of different elemental node types. $m$ is the side length of a data block. $n$ is the number of non-zero elements inside the data block. $d$ is the size of one data entry in bits. $p$ is the pointer size which needs to be sufficiently large.

| | Access time | Bit overhead / non-empty Voxel | | Shift sensitive | Allows empty space skipping |
|---|---|---|---|---|---|
| | | upper bound | lower bound | | |
| Dense volume | $O(1)$ | $\sim dm^3$ | 0 | no | no |
| Pointer octree | $O(\log(m))$ | $p\log_2(m)$ | $\sim p/7$ | yes | yes |
| Voxel List | $O(\log(n))$ | $3\log_2(m)$ | $3\log_2(m)$ | no | no |

where $n$ is the number of elements in the list. It is possible to decrease the look-up time in a sorted voxel list with binary search to $O(log(n))$.

- Octree: The octree itself is an adaptive data structure performing best for medium sparse data sets. In addition to the leaf nodes a full *pointer octree* has $(m^3-1)/7$ internal nodes, where $m$ must be a power of two. The access performance depends on the side length of the volume and not the number of elements with a worst case access of $O(log(m))$ traversal steps. As a result, in very sparse cases, voxel list access can outperform octree access. We have implemented a *pointer octree* [27]. Each node of the tree stores exactly 8 pointers to its children. Pointers are indices into a one dimensional array and can also point to an empty element. In our case the leaf nodes store bricks with $2^3$ values. In our implementation the maximum overhead of pointers for a full octree is approximately $\frac{1}{6}$ of the number of values in the dense volume. However, octrees are shift sensitive [23] and the number of internal nodes in a sparse octree depends on the exact location of the octree.
- Dense volume: A Cartesian grid has the lowest overhead per data voxel and is therefore memory efficient for very dense data. However, the major advantage of a dense grid is fast access and in many cases it makes sense to trade some memory for better performance.

Table 1 gives a theoretical comparison of the implemented elemental node type data structures. The upper bound is reached if the volume is practically empty, the lower bound is reached if the volume contains no zero entries.

Among the many octree implementations the best one, with regard to bit overhead per voxel, is the pointer-less octree (as used by Balmelli [1]) with 0 bit overhead per entry. An octree containing one leaf node has the theoretical minimal size of $3\log_2(m)$, which is the number of bits required to store the Morton code [19] to reach the brick containing the entry. The Morton code encodes each octree traversal step with 3 bits to reach one of the 8 child nodes. In total there are $\log_2(m)$ levels in an octree of side-length $m$. The octree properties in the table are representative for pointer octrees. If the octree contains only one value, the upper bound is reached. In addition to the data value, the octree requires $log_2(m)$ pointers (one for each octree level) to address the single non-zero entry. The number of internal nodes of a full octree is approximately $1/7$ of the number of leaf nodes. Therefore, a full octree requires $\sim 1/7$ pointer per data entry.

Table 1 also shows that the dense volume representation has the best access performance. The optimal memory representation depends on the data set. The dense representation is optimal for full volumes, but has the highest overhead for a single value. The worst case of the octree performs better than the worst case of the dense data representation. The voxel list has a constant overhead per non-zero voxel independent of the number of values inside a volume. The memory requirement of a specific representation depends on the density inside a volume block, the data type of the data entries and for the octree, the exact location of the non-empty voxels inside the volume. It can be seen that the three data structures complement each other since they are optimal for cases where the others perform poorly.

Other data structures such as hash tables or multilevel bricking could further complement the data structures. However, for the moment we restrict ourselves to the analysis of our data structure using the described elemental node types.

## 5 Optimization of elemental node level

We optimize the hybrid data structure with respect to memory by locally adapting the elemental node types and globally adapting the brick size. The performance is optimized through several low-level optimizations that can be added on demand by a data aware compiler.

**Memory optimization:** To find a representation that requires a minimum of memory it is necessary to pick a combination of the most suitable elemental node and an optimal block size $m$. To determine the layout of a specific volume data set, we use different variations of a parallel reduction. The size of a dense representation is known beforehand, as $dm^3$ where $m$ is the side-length of a brick and $d$ is the number of bytes of one data entry. To estimate the size of a voxel list inside a sub-volume it is necessary to find the number of non-zero values. We efficiently achieve this by applying a parallel reduction. The octree size estimation requires two adapted reductions: one to determine the number of leaves and one to determine the number of internal nodes in the final sparse octree. Finding the number of leaf nodes requires a reduction on a binary occupancy volume (i.e., voxels are one for non-zero values and zero otherwise). Finding the number of internal nodes requires a reduction starting at a binary representation at $level = height - 1$ of the octree. In our case, the octree is slightly modified to support leaves of 8 values instead of single scalar octree leaf values, which basically shifts the reductions up by one level inside the tree. This is done because, pointers to the leaf nodes only make sense if the data values are much larger than the pointers. This is not the case for most scalar data, therefore leaves with 8 values require less memory than the full pointer octree and the access performance is also higher.

To globally optimize the memory requirement we compute the memory requirements of each brick for a given brick size. It is sufficient to choose the minimal elemental data structure for each brick. From the results of all considered brick sizes the global minimum is picked as the memory optimal solution.

**Performance optimization:** During the performance optimization stage different low level data dependent optimizations can be applied. Our data aware compilation stage makes use of high level information and thereby applies better optimizations than the regular (non-data aware) compiler. Code branches that are known not to be traversed because of the specific configuration of the memory optimal data structure are removed during the just-in-time compilation step. For instance, for a configuration that does not contain any voxel lists, all code that is specific to the voxel lists is removed from the program. Note that a regular compiler (e.g., the OpenCL compiler) cannot apply this dead code removal since it depends on the data representation.

A software stack cache for regular access patterns is another low level optimization that is performed by our compiler. This leads to performance benefits for ray traversal for very small work group sizes only. For ray traversal patterns a simplified version of the kd-restart algorithm [6] is used to perform empty space skipping. When hitting an empty brick the algorithm directly advances to the most distant point on the empty brick bounding box. This saves redundant look-ups in empty bricks and results in speed up factors of up to 1.5. All these optimizations can be enabled on demand and are disabled (i.e., removed from the code) without any side effects if they do not improve the performance.

## 6 Optimization of root level

The root node representation stores the type and the pointer to every brick inside an array. This representation can either directly be uploaded to the GPU or it can be transformed to code through a just-in-time compilation step.

### 6.1 Direct volume bricking approach

The direct bricking approach stores internal nodes as a tuple $(type, pointer)$. The size of the bricks is globally defined in our current implementation. The *type* can be one of the elemental node types. The *pointer* is handled in a type specific way.

Figure 3 gives a conceptual overview of the data structure. At the root level, each node needs to store a type and a pointer, which consists of two 4 byte integers in our case. Elemental nodes depend on the size of the data type in bytes $d$. Dense volumes store full bricks of data values. Each list stores one 4 byte pointer to its last element in the list offset array. Each list element stores a data value and three 2 byte values as its coordinates. Internal octree nodes always store eight 4 byte pointers and leaf nodes store eight data values.

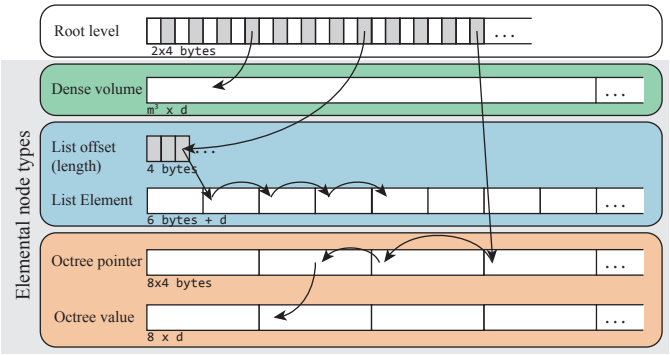



Fig. 3: Conceptual data layout: root level nodes point to the elemental data structures. Traversal is specific to the type of a brick. Different traversals are shown for the different elemental node types.

Every look-up starts by an address translation and then retrieves the $(type, pointer)$ tuple of the brick. Look-ups for empty nodes can be discarded at this point. The specialized look-up procedures are:

- Dense volume bricks store a pointer into the dense volume array at the root level. The size of the dense volume array is $m^3$ times the number of dense volume bricks.
- To access one of multiple voxel lists stored in a buffer it is necessary to define the start and end voxel of each list in addition to its entries. One look-up retrieves the start offset from an offset array. Afterward, the list is traversed to search for a specific coordinate. If the coordinate is not found the look-up returns zero.
- For the octree the brick pointer directly points to the root of the tree inside a global octree pointer array. The brick resolution determines the maximal traversal depth of the octree. The octree is traversed until it reaches a leaf node or an empty node. The actual values are stored in a separate region of the memory.

### 6.2 Just-in-Time compilation approach

To reduce the traversal overhead that is introduced by the root level we use just-in-time compilation that eliminates the data fetch by incorporating it into the code. As described in Section 3, a search tree is used to efficiently index the data. We use a kd-tree to index the elemental bricks of our data structure. The kd-tree splits the set of bricks at brick boundaries into large homogeneous regions containing only one elemental type. This makes it possible to discard empty regions fast. Homogeneous sub regions implement a bricking approach that uses only a single brick type, which locally reduces the number of branches.

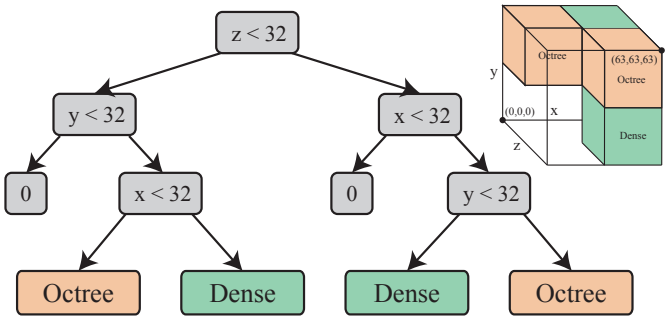



Fig. 4: Example of a kd-tree.

To group bricks with the same type, our splitting criterion maximizes the *information gain*, as introduced by Mitchell [18] in the context of optimizing decision trees. The entropy $E$ of a region $S$ with respect to the brick types is given in Equation 1. $b$ is the number of different brick types, in our case four. $p_i$ is the portion of the bricks with type $i$.

$$E(S) = -\sum_{i=1}^{b} p_i \log_b p_i \tag{1}$$

Minimizing the average entropy is equivalent to maximizing *information gain*. We calculate the average entropy for each possible split as given in Equation 2. The volume is recursively split into two sub-regions $S_j$, $j = 2$. Lets define $v_j$ as the volume of a sub-region (i.e., a kd-tree child) $S_j$, the *information gain* $I(S)$ is then computed as

$$I(S) = \sum_{j} \frac{v_j}{v} E(S_j) \tag{2}$$

where the entropy of each sub-region $E(S_j)$ is weighted by the volume of this sub-region $v_j$ normalized by the total volume $v$.

Splitting is applied recursively until each kd-tree leaf contains bricks of only one elemental node type. Figure 4 shows an example where a volume consists of eight bricks, each of size $32^3$. After the kd-tree is constructed it is just-in-time compiled. Listing 5 shows the code for the kd-tree in Figure 4.

```
1 // jit get function with look-up position (x,y,z)
2 voxel get(x,y,z)
3 { // compare to splitting positions
4   if(z < 32) // traverse left subtree
5     if(y < 32) return 0;
6     else
7       if(x < 32) return get_oct(index_oct1, x,y,z);
8       else return get_dense(index_dense1, x,y,z);
9   else // traverse right subtree
10    if(x < 32) return 0;
11    else
12      if(y < 32) return get_dense(index_dense2, x,y,z);
13      else return get_oct(index_oct2, x,y,z);
14 }
```

Listing 5: JiTTree get function pseudo code

In Listing 5, four optimizations were applied:

(i) **traversal optimization**: no memory fetches are required for the kd-tree traversal. The splitting axis order is directly encoded in the conditionals. Splitting plane positions are inserted as literals.

(ii) **efficient discarding of empty regions**: empty regions are quickly discarded without any memory fetches.

(iii) **implicit specialization:** no routing between specializations of the *get* functions of different elemental data structures is required. The $\text{get}_{dense}$ and $\text{get}_{oct}$ functions are inserted at the leaf nodes. No type checks are required, and no type dependent *switch* statement is needed.

(iv) **index in-baking:** the indices of the octrees (i.e., $\text{index}_{oct1}$ and $\text{index}_{oct2}$) and dense bricks (i.e., $\text{index}_{dense1}$ and $\text{index}_{dense2}$) are literals that are directly baked into the code. This eliminates one memory fetch per leaf node.

Leaves that are empty return zero. Leaves that contain a single non-zero elemental node directly call the specific *get* method. In the case of multiple bricks in one leaf node the kd-tree generation guarantees that they are of the same type. Therefore the same get method is called for all of them. However, the index (which is a parameter of the get method) is different for each brick. By generating local index arrays we can further optimize the code, remove additional branching and benefit from the above mentioned kd-tree generation that reduces the number of leaf nodes.

Listing 6 shows code for the unoptimized case where three dense bricks are next to each other and fall into the same kd-tree leaf node.

```
// jit get function without leaf node branch removal
voxel get(x,y,z)
{
   ... // traversal to a leaf node
   if(z < 32) return get_dense(index_dense1, x,y,z);
   else if (z < 64) return get_dense(index_dense2, x,y,z);
   else return get_dense(index_dense3, x,y,z);
}
```

Listing 6: Leaf node pseudo code without branch removal

Although the three branches call the same get method, the index parameter differs. The branches can only be removed by introducing a local index array that holds the valid index for each brick.

Listing 7 shows code for the fifth optimization that we apply:

(v) **leaf node branch removal**: branches that are introduced by the leaf nodes of the kd-tree are removed by introducing a local index array.

```
// jit get function with leaf node branch removal
voxel get(x,y,z)
{
   ... // traversal to a leaf node
   index[] = {index_dense1, index_dense2, index_dense3};
   i = index[indexTransform(x,y,z)];
   return get_dense(i, x,y,z);
}
```

Listing 7: Leaf node pseudo code with branch removal

The array index[] is initialized with the literals $\text{index}_{dense1}$, $\text{index}_{dense2}$, and $\text{index}_{dense3}$, that are resolved at compile time. The indexTransform function transforms the global coordinates to a local linear index into the kd-tree cell. As a consequence the get method can be executed in parallel by all threads reducing branch divergence. This optimization is only applicable in the case that bricks of the same type cluster in a region and are grouped by the kd-tree into one cell. In Listings 6 and 7 only three bricks of the same type are grouped together. In practice these groups often become larger and more branches are removed.

## 7 Implementation

OpenCL and CUDA (version 7) support run-time compilation and therefore lend themselves to just-in-time compilation approaches. Our JiTTree implementation was done using OpenCL. A source-to-source compiler takes an OpenCL program with specialized instructions for memory access and translates it to pure OpenCL code during runtime. The compiler can better specialize the generated code when the data set is known. Computation on multiple independently generated JiTTrees is done by defining them as separate kernel arguments.

To speed up the memory optimization phase a fast calculation of the memory requirements is essential. Regions containing only zero values are marked with the empty type and no further optimization is needed. The computation of the memory requirements for a dense node is trivial. For voxel list nodes a single parallel reduction operation is needed to compute the number of non-zero values. The size of the octree nodes is calculated by first counting the number of octree leaf nodes and then counting the number of internal nodes. This is efficiently achieved by computing two reduction operations. These computations are all done efficiently in parallel. The memory requirements of the elemental node types are sufficient to compute the memory optimal JiTTree for a given brick size. We repeat this computation for different brick sizes to get the overall most efficient hybrid data structure.

After a specific representation of the data structure is defined we either apply a bricking approach or generate a just-in-time compiled kd-tree. Our source-to-source compiler first generates a kd-tree on the application side based on the previously defined brick types. The kd-tree is recursively unrolled into non-recursive OpenCL code. The source-to-source compiler generates code for internal kd-tree nodes as well as for leaf nodes. Internal nodes generate conditionals (i.e., if/else statements). Leaf nodes are either empty (code returns zero), contain a single brick (code calls a *get* method) or contain multiple bricks of the same type (code uses *leaf node branch removal* described in Section 6.2).

The memory for the different elemental data structures is allocated once the optimal data structure is known. The actual initialization of the data structures is done per brick. The initialization of voxel lists and octrees on the GPU is done using histogram pyramids [30]. We make use of the fact that voxel lists initialized via histogram pyramids are sorted in Morton order. Therefore binary search is used when the data is accessed.

The octree nodes are generated using a variant of OcPyramids [29], that combine multiple histogram pyramids. In our implementation we allow a block of size $2^3$ in the leaf nodes resulting in a more compact octree with an overhead of approximately $\frac{1}{6}$ pointer per data entry for a full octree.

## 8 Results

The results are generated on a machine using an NVIDIA GeForce GTX TITAN X, Intel Core i7-4770K @ 3.50 GHz and 16 GB RAM. The screen resolution for rendering is $768 \times 768$ pixels.

### 8.1 Memory consumption

We compare the memory requirement for 15 data sets shown in Table 2. Larger renderings can be seen in Figure 10. Noise was removed from most of the shown data sets since we are interested in the investigation of sparse data sets. Wherever we applied a threshold to a data set we denote this in Table 2 with the specific threshold indicated as subscript. The data after thresholding are shown as icons. It is important to note that we also tested our data structure on all shown data sets without applying a threshold. Although our data structure is not designed for dense data, it outperforms other representations, such as a dense bricking or a dense representation, for most cases. However, for a better analysis of the data structure we used data sets with varying sparsity. We compared the final size in relation to the total size of a dense representation. The percentage of non-zero values gives the

Table 2: Memory consumption results. The brick types are empty (gray), dense (green), octree (orange) and voxel list (blue)

| | Image | Volume size | Optimal brick size | Non-zero values (%) | Final size (% of dense) | Type of bricks (% of volume) |
|---|---|---|---|---|---|---|
| Bonsai $_{th=0.08}$ |  | 512x512x189 | 16x16x16 | 8.17 | 14.85 | |
| Bunny $_{th=0.08}$ |  | 512x512x361 | 8x8x8 | 28.76 | 34.28 | |
| Carp $_{th=0.08}$ |  | 256x256x512 | 16x16x16 | 16.27 | 19.82 | |
| Christmas tree $_{th=0.03}$ |  | 512x499x512 | 16x16x16 | 1.47 | 4.02 | |
| Engine $_{th=0.08}$ |  | 256x256x256 | 8x8x8 | 8.34 | 12.32 | |
| Frog $_{th=0.08}$ |  | 256x256x44 | 8x8x8 | 11.22 | 18.23 | |
| Hydrogen |  | 128x128x128 | 8x8x8 | 32.72 | 38.58 | |
| Monkey-CT |  | 256x256x62 | 16x16x16 | 17.11 | 29.68 | |
| Piggy Bank $_{th=0.03}$ |  | 512x512x134 | 8x8x8 | 23.72 | 30.53 | |
| Porsche $_{th=0.03}$ |  | 559x1023x347 | 16x16x16 | 40.31 | 52.46 | |
| Schaedel $_{th=0.03}$ |  | 512x512x333 | 8x8x8 | 15.64 | 23.14 | |
| Stag beetle |  | 832x832x494 | 16x16x16 | 4.06 | 4.75 | |
| Kingsnake $_{th=0.15}$ |  | 1024x1024x795 | 16x16x16 | 43.68 | 44.85 | |
| Vessels $_{th=0.15}$ |  | 1024x1024x1024 | 16x16x16 | 1.67 | 4.22 | |
| Connectomics |  | 1024x1024x1024 | 16x16x16 | 29.68 | 34.04 | |

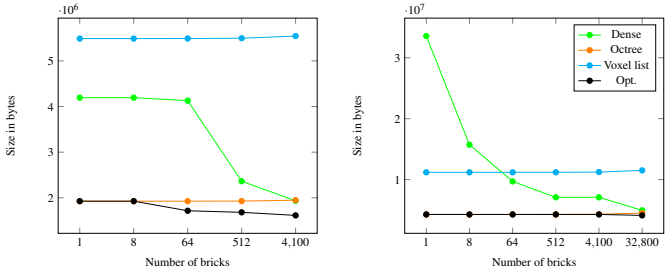
theoretical lower bound for the optimal data structure. The last column shows the distribution of elemental node types for the memory optimal JiTTree. Grey denotes the empty, green the dense, orange the octree and blue the voxel list representation.

The Vessels data set, for example, has a high number of voxel list bricks, which means that many bricks contain only a very small number of non-zero values. The stag beetle data set results in a low overhead because it contains a large number of empty bricks and most bricks containing non-zero values are densely populated. In our experiments our approach never exceeded three times the size of the lower bound solution.

In comparison to perfect spatial hashing, which achieves an overhead of $3-7$ bit per data entry for very sparse data, we have shown that our approach can be applied to a broader range of volumes (also with much higher density). The overhead of our data structure for the tested data sets lies between 1.8 and 27.7 bit per non-zero voxel. Volumes with very dense and very sparse regions, can be represented very efficiently with our approach. Still, the approach is extensible to other elemental representations such as spatial hashing, to further improve the overall performance for special data set characteristics.

Figure 5 shows the memory requirements for two data sets for different brick sizes and different elemental data. In Figure 5a the hydrogen data set is shown. It can be represented optimally with a brick size of 8. A combination of dense, octree and voxel list (*Opt.*) performs better than any bricked solution using the octree, the dense, or the voxel list representation only. Even though the voxel list does not provide good results over the whole data set, there are still very sparse bricks that are best represented by a voxel list as seen in Table 2.

In Figure 5b the results for the Engine data set are shown. The data is represented effectively by an octree with a larger brick size. Nevertheless, the memory optimal (*Opt.*) solution still performs better. If for a specific data set a single elemental data structure would be the best solution, our approach would degenerate and fall back to this solution. However, we have never observed this in practice.

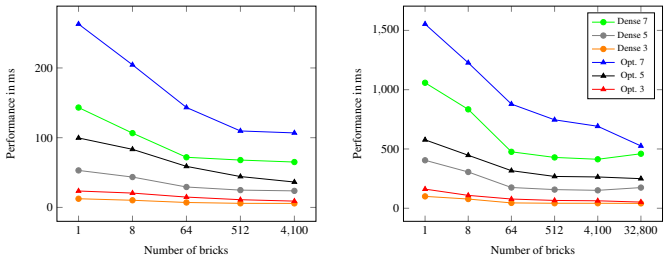



(a) Hydrogen (b) Engine

Fig. 5: Memory requirement of dense, octree, voxel list and memory optimal bricking. The memory optimal bricking outperforms any other bricking for most cases.

## 8.2 Performance

The performance was tested for a mean filter with different kernel sizes as well as for volume ray casting. Figure 6 shows the performance for the mean filter. As expected the *dense* node type results in the best performance for all brick sizes and kernel sizes. The memory optimized data structure (i.e., a mixture of dense, octree and voxel list) is still faster than an octree bricking solution for these data sets.




(a) Hydrogen (b) Engine

Fig. 6: Performance of a mean filter with kernel sizes $7^3$, $5^3$ and $3^3$ in ms.

Figure 7 compares the performance of the direct volume bricking approach of the memory optimal solution (Section 6.1) with the JIT compilation approach of the memory optimal solution (Section 6.2). The JIT approach improves the performance in most cases with an average speed up factor of 1.29 over all measured cases. The highest speed up in Figure 7, a factor of 1.79, is given for the stag beetle dataset at a

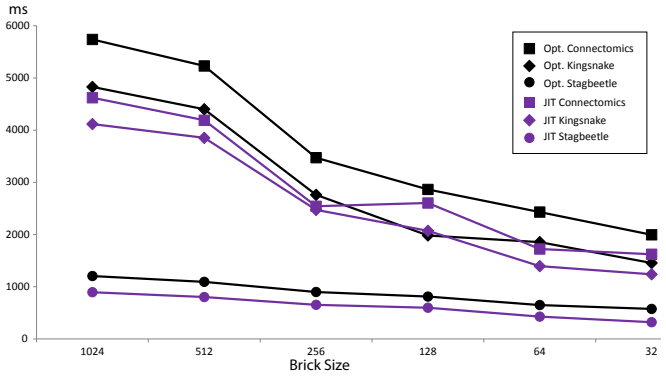



Fig. 7: Performance comparison for a mean filter with $3x3x3$ kernel size. JIT compilation (purple) compared to direct volume bricking bricking (black) for three different data sets.

brick size of $32^3$. Only for one case (kingsnake dataset with brick size $128^3$) we observe a decrease in performance by a factor of 0.96.

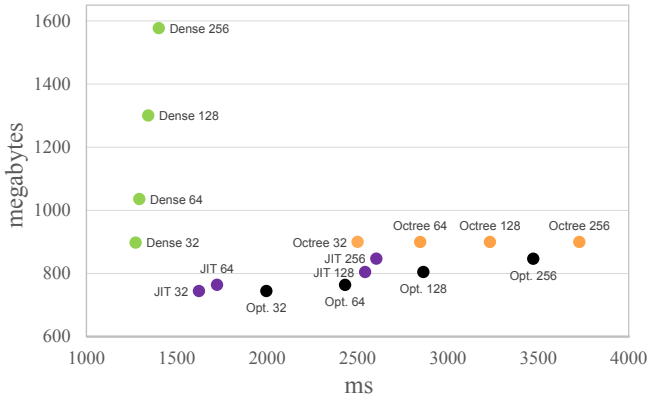



Fig. 8: $3x3x3$ mean filtering of the Connectomics data set. Different brick sizes are tested for dense, octree and memory optimized bricking approaches.

Figure 8 shows the performance of a $3x3x3$ mean filter for different data representations. The dense bricking approach (green) performs better with a smaller brick size. The octree bricking (orange) does not decrease its memory requirement significantly with a smaller brick size. Nevertheless it is faster for a smaller brick size since the traversal depth is reduced. The memory optimized solutions (black) perform best in terms of memory consumption over all different brick sizes. The just-in-time compiled variations (purple) improve the performance of the memory optimal solutions. A comparison with voxel lists was omitted from Figure 8, since their performance is much worse for an entire data set. Nevertheless, voxel lists perform optimal in terms of memory for very sparse nodes, which makes them attractive for a hybrid data structure.

Table 3: JIT generated code properties and build times for the Connectomics data set.

| # bricks | # leaves | max. branch depth | kd-tree gen. (ms) | OpenCL comp. (ms) |
|---|---|---|---|---|
| 8 | 1 | 0 | 0.05 | 151 |
| 64 | 12 | 6 | 0.62 | 275 |
| 512 | 95 | 14 | 6.42 | 3086 |
| 4096 | 363 | 17 | 67.44 | 5365 |
| 32768 | 2148 | 22 | 1085.25 | 78306 |

Table 3 shows the JIT generated code properties and build times for a typical case. The number of leaf nodes increases with the number of bricks. However, the dependency is not linear since the kd-tree groups bricks of the same type together. Therefore, the number of leaves does not grow as fast as the number of bricks. Table 3 also shows the source-to-source kd-tree generation and the OpenCL compile times.

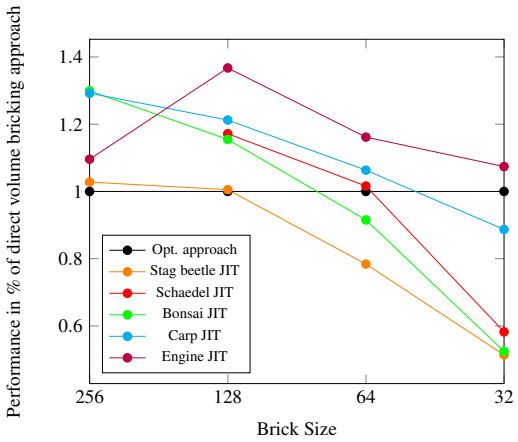



Fig. 9: Performance of ray casting with the JIT approach compared to the memory optimal bricking (black line).

In contrast to filtering, the JIT approach does not perform well for ray casting as shown in Figure 9. In our experiments we found that it only outperforms the direct bricking for small, homogeneous data sets. Larger data sets with a high number of bricks are better handled with a direct bricking approach. We suspect that the ray access pattern leads to much higher branch divergence than the stencil access pattern for the mean filter calculation.

## 9 Discussion and Future Work

Our hybrid data structure performs very well in terms of adaptation resulting in low memory consumption. By adding more elemental data structures we hope to approach the theoretical optimum of memory consumption with our hybrid data structure in the future. One of the most important properties of a modern GPU data structure is to reduce memory bandwidth (which is the bottleneck in memory-bound kernels). With our just-in-time compilation approach we can offload some of the burden reducing the memory bandwidth. However, just-in-time compilation comes at the cost of longer programs and especially nested conditionals. In some cases this transforms memory-bound programs into instruction-bound programs. Also with our approach we are exhausting the capabilities of the current OpenCL compilers. This limits the JiTTree to compile only up to 32768 leaf nodes. For higher node counts we get inconsistent compile times. Our experiments show that the just-in-time compilation of the root level is a viable option to improve performance. In the future we want to investigate other spatial subdivision schemes that can be just-in-time compiled to further increase performance.

We want to improve the JIT compilation approach for other memory access patterns like ray traversal. One potential cause of the lower performance in this case is branch divergence which can be addressed with optimization techniques like *branch distribution* and *iteration delaying* [11].

Another direction for research is to investigate how to add dynamic write capabilities to our current implementation. For the moment we have focused on read-only access since most other efficient sparse data structures are also optimized for this situation. Finally, the addition of spatial hashing would further increase the adaptivity of our hybrid data structure. Also it could be a candidate for a just-in-time compiled root level.

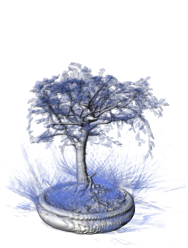
In addition, it could be beneficial to defer the JIT compilation even further. For instance on compute clusters each node could just-in-time compile the optimal data structure for a sub-domain of the data. Further experiments (for instance integration of the JiTTree with OpenMP) are necessary to understand the applicability of our approach in such scenarios.

## 10 Conclusion

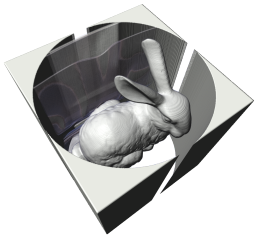
Sparse data structures are an important tool for efficient computations on the GPU. Our hybrid data structure makes it possible to combine multiple elemental data structures resulting in lower memory requirements. The individual elemental data structures can be exchanged and improved separately. JiTTree reaches a noticeable benefit even with a set of simple elemental data structures. We demonstrated that the involved optimization leads to good results for a broad range of data sets.


## Acknowledgments

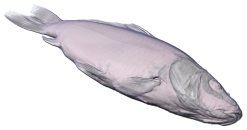
The research presented in this publication was supported by the King Abdullah University of Science and Technology (KAUST) Visual Computing Center, and the ViMaL project (FWF - Austrian Science Fund, no. P21695).

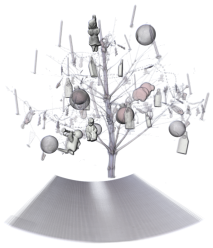
(a) Bonsai



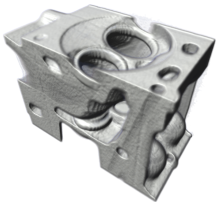
(b) Bunny



(c) Carp

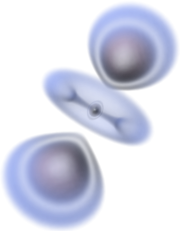


(d) Christmas tree

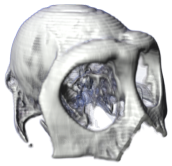


(e) Engine

(f) Frog

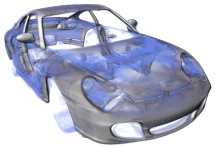


(g) Hydrogen



(h) Monkey-CT

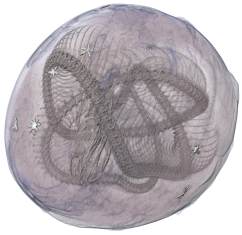
(i) Piggy Bank



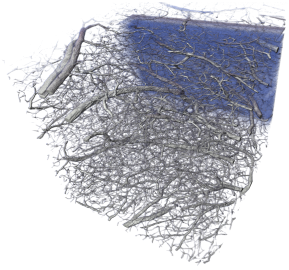
(j) Porsche

(k) Schaedel

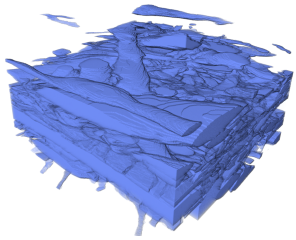
(l) Stag beetle



(m) Kingsnake



(n) Vessels



(o) Connectomics

Fig. 10: Volume rendering of the benchmark data sets.

## REFERENCES


[1] L. Balmelli, J. Kovaevi, and M. Vetterli. Quadtree for embedded surface visualization: Constraints and efficient data structures. In *Proc. Int. Conf. Image Processing (ICIP)*, pages 487–491, 1999.

[2] D. Benson and J. Davis. Octree textures. *ACM Transactions on Graphics (TOG)*, 21(3):785–790, 2002.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[4] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[5] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[6] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 15–22, New York, NY, USA, 2005. ACM.

[7] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.

[8] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, Dec. 1982.

[9] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers & Graphics*, 28(5):719–729, 2004.

[10] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2285–2294, Dec 2012.

[11] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1–3:8. ACM, 2011.

[12] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[13] R. Kähler, M. Simon, and H.-C. Hege. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *Visualization and Computer Graphics, IEEE Transactions on*, 9(3):341–351, July 2003.

[14] A. Knoll. A survey of octree volume rendering methods. *Scientific Computing and Imaging Institute, University of Utah*, 2006.

[15] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[16] S. Lefebvre and H. Hoppe. Perfect spatial hashing. In *ACM SIGGRAPH*, pages 579–588, New York, NY, USA, 2006. ACM.

[17] A. Lefohn, J. M. Kniss, R. Strzodka, S. Sengupta, and J. D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, Jan. 2006.

[18] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[19] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, International Business Machines Company, 1966.

[20] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012.

[21] T. Rompf, A. K. Sujeeth, N. Amin, K. J. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. *ACM SIGPLAN Notices*, 48(1):497–510, 2013.

[22] D. Ruijters and A. Vilanova. Optimizing GPU Volume Rendering. In *Winter School of Computer Graphics (WSCG) 2006*, pages 9–16, 2006.

[23] H. Samet. Spatial data structures. Addison-Wesley, 1995.

[24] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[25] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computer Systems*, 13(4s):134:1–134:25, 2014.

[26] X. Tong, W. Wang, W. Tsang, and Z. Tang. Efficiently rendering large volume data using texture mapping hardware. In *Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization*, pages 121–132. Springer, 1999.

[27] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *SIGGRAPH Comput. Graph.*, 24(5):57–62, Nov. 1990.

[28] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. In *Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17)*, pages 382–393. IEEE, 2011.

[29] G. Ziegler. *GPU data structures for graphics and vision*. PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken, 2010.

[30] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel. GPU point list generation through histogram pyramids. In *Proceedings of VMV (2006)*, pages 137–141, 2006.