Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Adaptive, Hybrid Data Structure for Sparse Volume Data on the GPU

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Visual Computing

eingereicht von

Matthias Labschütz
Matrikelnummer 8971103

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Eduard Gröller
Mitwirkung: Dr. Peter Rautek

Wien, 17.01.2016

(Unterschrift Verfasser)

(Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

An Adaptive, Hybrid Data Structure for Sparse Volume Data on the GPU

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Matthias Labschütz
Registration Number 8971103

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Eduard Gröller
Assistance: Dr. Peter Rautek

Vienna, 17.01.2016

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Matthias Labschütz
Sigmundsgasse 14/10, 1070 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

Foremost, I wish to thank Peter Rautek, my de facto supervisor, for the general direction of the thesis and many specific ideas. The major part of the implementation of this thesis was done at the King Abdullah University of Science and Technology, where I have spent eight months. I wish to thank everybody who has made this possible, in specific the Visual Computing Center, Markus Hadwiger and Peter Rautek. The work atmosphere was excellent and it was one of the major reasons for steady progress.

The implementation of the project was part of the ViSlang framework, which is part of Volume Shop. I wish to thank Peter Rautek, Stefan Bruckner and any preceding developers of the framework. Being able to rely on already implemented parts of a framework was helpful. I have to thank my supervisor for assisting me greatly in installing and getting to know the framework, as well as helping me out *anytime* with questions. Frequent progress reports and discussions about the further advancement were essential for the successful outcome of the project. We were being able to present the major contribution of this work at IEEE Vis 2015. Once again, I have to positively mention the excellent collaboration especially during writing the paper, but also in the later phases of the submission.

I also wish to thank the Institute of Computergraphics at TU Wien and especially Meister Eduard Gröller, for supporting me with hardware and a place to work on short notice and giving me the opportunity to present my work at the Institute for additional feedback.

Finally, I also wish to thank everybody who provided valuable feedback before the conference. Especially the visualization group at KAUST, the Institute of Computergraphics and my co authors, Peter Rautek, Stefan Bruckner, M. Eduard Gröller and Markus Hadwiger.

I also thank the International Office of TU Wien for the stipend for short-term scientific work abroad. Additionally, I want to thank my parents for supporting me over the years.

Abstract

Dealing with large, sparse, volume data on the GPU is a necessity in many applications such as volume rendering, processing or simulation. The limited memory budget of modern GPUs restricts users from uploading large volume data-sets entirely. Fortunately, sparse data, i.e., data containing large empty regions, can be represented more efficiently compared to a common dense array. Our approach makes it possible to upload a full data set even if the original volume does not fit on the GPU.

In previous work, a variety of sparse data structures have been utilized on the GPU, each with different properties. 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.

Our observation is, that data sets often contain regions of different sparsity. Depending on the sparsity of a region, a specific data structure (e.g., an octree, a voxel list) requires the least memory to store the data. We formulate an algorithm that is able to automatically find this memory-optimal representation. By using such a combination of different data structures, we achieve an even better representation than any single data structure for real world data sets. We call such a data structure a *hybrid data structure*.

Any sparse data structure introduces an access overhead. For example, the access to an octree requires one additional indirection per height level of the tree. A voxel list has to be searched to retrieve a specific element. By using a hybrid data structure, we also introduce an access overhead on top of the overhead that comes from using a sparse data structure. In our work we introduce *JiTTree*, which utilizes a data aware just-in-time compilation step to improve the access performance of our hybrid data structure.

We show that the implementation of our hybrid data structure effectively reduces the memory requirement of sparse data sets. JiTTree can improve the performance of hybrid bricking for certain access patterns such as stencil accesses.

Kurzfassung

Der Umgang mit großen, spärlich besetzten Volumen-Daten auf der Graphikkarte ist in vielen Applikationen notwendig, wie beispielsweise im Volumen-Rendering, in der Datenverarbeitung oder bei Simulationen. Der beschränkte Speicherplatz heutiger Graphikkarten hindert Nutzer daran, größere Datensätze vollständig auf die Graphikkarte zu laden. Glücklicherweise ist es aber möglich, spärlich besetzte Daten, also solche, die größere leere Bereiche enthalten, effizienter zu speichern als als gewöhnliches Array. Unser Verfahren ermöglicht es Datensätze zur Gänze zu speichern, selbst wenn die originale Datenmenge nicht auf die Graphikkarte passen würde.

In vorhergehenden Publikationen wurde eine Vielzahl von Datenstrukturen mit unterschiedlichen Eigenschaften verwendet. Baum Strukturen wie der Octree, kd Baum oder N^3 Baum sind hierarchische Lösungen für dichter besetzte Daten. Für weniger dicht besetzte Datensätze ist räumliches Hashing speichereffizienter und ermöglicht eine höhere Zugriffsgeschwindigkeit. Sehr spärlich besetzte Daten können als sortierte Liste effizient gespeichert werden, wobei der Zugriff mit Binärsuche beschleunigt werden kann.

Unserer Beobachtung nach besitzen Datensätze oft Regionen mit unterschiedlicher Dichte. Entsprechend der Dichte einer Region hat eine unterschiedliche Datenstruktur, beispielsweise ein Octree oder eine Liste von Voxeln, den geringsten Speicherverbrauch. Wir präsentieren einen Algorithmus, der diese speicheroptimale Repräsentation automatisch finden kann. Für realitätsnahe Datensätze besitzt so eine Kombination von unterschiedlichen Datenstrukturen einen geringeren Speicher-Verbrauch als jede einzelne Datenstruktur für sich. Wir nennen diese Datenstruktur *hybride Datenstruktur*.

Jede effiziente Datenstruktur für spärlich besetzte Daten erzeugt einen Zugriffs-Mehraufwand. Beispielsweise braucht ein Octree eine weitere Dereferenzierung pro Höhenstufe des Baumes. Eine Liste von Voxeln muss erst durchsucht werden, um ein bestimmtes Element zurückzuliefern. Beim Verwenden einer hybriden Datenstruktur wird die Zugriffszeit ebenfalls verringert. Das entsteht einerseits dadurch, dass man nun statt Arrays beispielsweise Octrees oder Listen von Voxeln verwendet, und andererseits verringert auch die Realisierung einer hybrid Datenstruktur die Zugriffszeit. Um die Zugriffszeit zu erhöhen, stellen wir erstmals *JiTTree* vor. JiTTree erzeugt zur Laufzeit des Programmes datenspezifischen Programmcode, der verwendet wird um auf unsere Datenstruktur mit höherer Performanz zuzugreifen.

Es zeigt sich, dass unsere hybride Datenstruktur spärlich besetzte Datensätze effizient speichern kann. Außerdem kann JiTTree die Zugriffszeit für bestimmte Zugriffsmuster, beispielsweise das Stencil-Zugriffsmuster, verbessern.

Contents

1 Introduction **1**
1.1 Problem Statement 1
1.2 Objectives and Approach 4
1.3 Contribution 5
1.4 Overview of the Thesis 5

2 Related Work **7**
2.1 Related Concepts 7
2.2 Spatial Data Structures 10
2.3 GPU Program Optimization and Data Structures 20
2.4 Partial Evaluation and Deferred Compilation 21

3 Hybrid Volume Bricking **25**
3.1 Brick Types 27
3.2 Hybrid Bricking 31
3.3 Modular Data Structure 35

4 JiTTree: Just-in-Time Compiled Tree **37**
4.1 Tree Unrolling 37
4.2 Type Routing 40
4.3 JiTTree Approach 42

5 Implementation **49**
5.1 Hybrid Bricking Construction 49
5.2 JiTTree Construction 59
5.3 OpenCL Programming 62

6 Results **69**
6.1 Memory Consumption 69
6.2 Performance 71
6.3 Varying Brick Size 74

7 Summary and Discussion **81**
7.1 Benefits 82

7.2 Limitations 83

8 Conclusion and Future Work **85**
8.1 Hybrid Bricking and JiTTree 85
8.2 Iteration Delaying 87

A Morton code **91**

Bibliography **93**

CHAPTER 1

Introduction

Volume data is a type of three dimensional data, typically represented as a Cartesian- or regular grid of scalar or vector values. Each sample point of this grid represents a value in space, for example a density at the corresponding spatial position. In the most simple case, such data is implemented as an array of values.

Many applications in visualization and other disciplines deal with volume data. Although we approach the topic through the lense of scientific visualization, it can be applied to other areas. Our test applications include processing (i.e., filtering) and volume rendering, but volume data is also used for example in simulation or real-time global illumination.

Today's volumetric data sets are memory demanding and data sizes grow rapidly as new imaging modalities become available. For instance a data set of type float with the extent of $1024 \times 1024 \times 1024$ voxels requires 4.29 GB of memory. Doubling the side-length to 2048 results in an eigth times higher memory consumption of 34.35 GB. Processing such data on the CPU is common on current desktop systems, however requires a considerable amount of RAM. Even though, CPU systems scale better to large data sets in terms of memory, GPU systems provide more processing power for data parallel algorithms.

The drawback of the GPU is that its memory size is more restricted. Even the highest end consumer graphics card in 2015, the GeForce GTX TITAN X, is limited to 12 GB of memory. Additionally, transferring data to the GPU has to be done explicitly by the programmer. Memory transfer is limited by the PCI-Express 2.0 transfer rate which is in practice up to 6.0 GB/s as stated in the Cuda FAQ [6]. As a result, large volume data-sets can easily exceed the available memory on graphic cards. This leads to the motivation to find a representation that handles such data in a memory efficient way.

1.1 Problem Statement

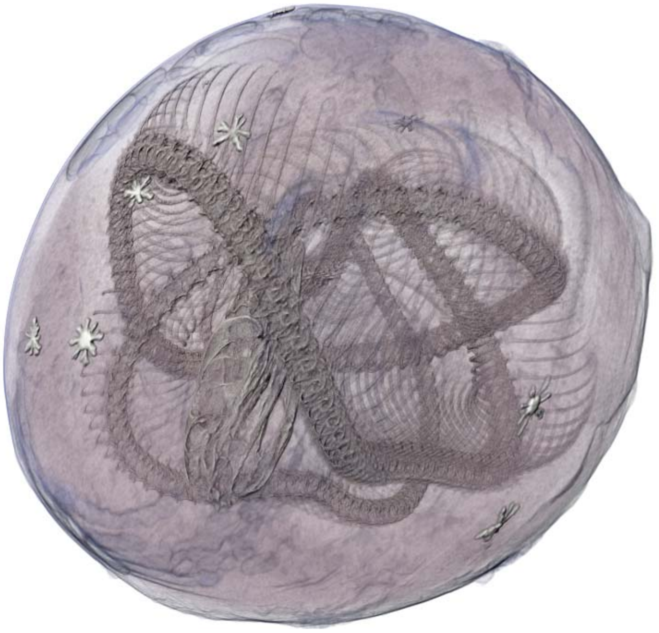
Many real-world volume data-sets contain empty regions where all values are zero. Since the non-zero values are assumed to carry the information, data sets with large empty regions are called *sparse*. The sparsity is defined as the fraction of the number of zero elements over the total size of the data set. A volume containing few non-zero values has a high sparsity, one with many non-zero values a low sparsity. The density is the fraction of the non-zero values in the data set.

Figure 1.1 shows a volume rendering of two example data sets with different levels and kinds of sparsity. The Kingsnake data set contains about 56 % of empty values on the outside, surrounding the snake-egg. Even though the global sparsity of the data set is 0.56, the data set has varying local sparsity. The central region of this data set, the snake egg, has a high density, the outer regions are of a high sparsity. The Vessels data set has a percentage of 98 % empty values which are more evenly spread out throughout the volume. Still, on a fine level the data set contains dense regions (the vessels) and sparse regions (the regions between the vessels).

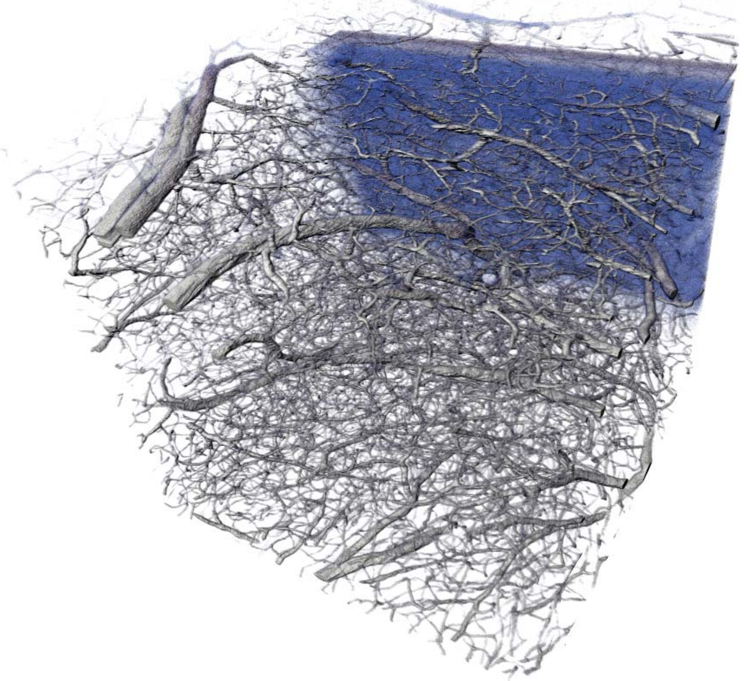
The goal of a *memory efficient* data structure is to avoid storing zero values. Such a data structure has to be adaptive to deal with regions of varying sparsity. This makes it possible to efficiently encode sub-regions of different sparsity in a single data set, but also makes it possible to encode a variety of data sets. Many sparse data structures have been proposed and we give an overview in Chapter 2. Variants of octrees and kd trees provide a hierarchical solution to the problem, but they introduce a traversal overhead when accessing data. Spatial hashing and volume bricking, focus on keeping the access performance high by minimizing the number of indirections to retrieve a data value. The former can be difficult to initialize, the latter can not adapt well to regions of high sparsity.

Typically, the designer of an algorithm has to pick an optimal data structure to solve a specific problem. Determining which representation is optimal can be difficult. Firstly, there are many different candidates to chose from. Secondly, optimality can depend on multiple factors, some of which may not be known to the programmer at the time of designing a program. In specific, optimality not only depends on the algorithm to be executed on the data, but also on the data itself. A programmer has to find an optimal balance between the memory requirement and the performance of an algorithm executed on the data set. We consider for example the *matrix octree* and the *linear octree* which are explained in section 2.2.2. Although both are variants of octrees, they behave differently for different data sets. The matrix octree has an access performance that depends on the side-length of the data set, while the access performance of the linear octree depends on the number of non-zero values in the data set. As a result, the linear octree is preferred if the input data to an algorithm is suspected to be of a high sparsity. The matrix octree on the other hand performs better than the linear octree if the data is of medium sparsity.

A strategy to handle a large variety of different data sets efficiently is to split algorithm design from data-structure design. As a result, the underlying data structure can be exchanged without having to rewrite the algorithm. This makes it possible to pick an optimal data structure if executing an algorithm on a certain type of data. The implementation of our data structure primarily deals with retrieval of data values through a *get* method. Dynamic writes to our data structure are not possible yet. When designing a parallel algorithm, our data structure is accessed through its *get* method. The actual data-structure layout is logically separated from the algorithm implementation.



(a) Kingsnake data set ($1024 \times 1024 \times 795$ voxels) with 44 % of non-zero values



(b) Vessels data set ($1024 \times 1024 \times 1024$ voxels) with 2 % of non-zero values

Figure 1.1: Volume rendering of two data sets with different levels and kinds of sparsity.

1.2 Objectives and Approach

The objective of our approach is to introduce an adaptive data structure. Its goal is to reduce the memory requirement of sparse data sets. This allows us to efficiently store large data sets on the GPU. We are also concerned with the performance of our approach. Our data structure can produce different representations, ranging from low memory requirement to fast access and higher memory requirement. We present an automatic routine to generate the representation with the lowest memory requirement. With a partial evaluation approach we further increase the run-time of this memory-optimal approach.

Our data structure combines multiple different data structures, such as an octree or a voxel list, to benefit from their advantages. Even though we use a meaningful set of four different data structures that complement each other, our approach is extendible to different data structures. This could further improve the results and allows it to incorporate data structures which may become popular in the future.

Any application or algorithm that has to deal with large sparse volume data-sets on the GPU can potentially benefit from our data structure. The drawback is that we introduce a performance overhead and that we did not address trilinear interpolation yet. As a result, our approach is better suited for algorithms with a low computational complexity that have to deal with large data sets. Even for more computationally demanding algorithms our approach can be beneficial, since our approach allows to store larger data sets on the GPU. Executing an algorithm on the entire data set on the GPU can be faster than uploading parts of the data set and batch processing them sequentially.

Our approach is based on the central observation that most data sets contain regions of different sparsity. Depending on the sparsity of a region, a specific data structure requires the least memory. For example in a medium sparse region a pointer octree can be a good representation. For very dense regions, it can be of interest to store the data as a full array of values. Our data structure uses different data structures for different regions of a data set, we therefore call it a *hybrid data structure*. The building blocks of our data structure we call *elemental data structures*. These complement each other in the sense that each of them is better suited for sub-regions of different sparsity. Our data structure adapts to the data set by assigning the optimal representation to the sub-regions of the data.

If a data set is uploaded to the GPU, we automatically generate and compile our data structure. OpenCL kernel code that uses our data structure requires to be recompiled when the data is loaded since we inline the traversal code into the algorithm. We specialize the traversal code of our data structure with respect to the data itself to increase the access performance. Since this specialization is done at the run-time of the application, we call our data dependent data structure just-in-time compiled tree (*JiTTree*). OpenCL lends itself well to our approach, since it typically compiles the kernel code during the execution of the application.

Any sparse data structure introduces an access overhead. For example, the access to an octree requires one additional indirection per height level of the tree. A voxel list has to be searched to retrieve a specific element. By using a hybrid data structure, we also introduce an access overhead on top of the overhead that comes from using sparse elemental data structures. We use *JiTTree* to improve the access performance of our hybrid data structure.

1.3 Contribution

The major contribution of this work is the presentation and evaluation of a novel GPU data structure. We extend the concept of traditional volume bricking to support four different brick types with varying properties. We implement dense volume, octree, voxel list and empty bricks. This makes it possible to better fit most data sets, which reduces the memory requirement of sparse volume data on the GPU. Our approach can be extended to support additional brick types. We suspect that this would make it possible to even further decrease the memory consumption of volume data. We call this bricking approach with different brick types *hybrid bricking*.

In addition, we present *JiTTree*, a data dependent code specialization step. We show that it can improve the performance for the stencil access pattern, but does not necessarily work in all cases. Nevertheless, we believe that data dependent compilation on the GPU can be used in other scenarios as well. The concept of the JiTTree was summarized in the journal paper *JiTTree: A Just-in-Time Compiled Sparse GPU Volume Data Structure* [26], and was presented at the IEEE Visualization Conference 2015, Chicago, USA.

Besides the main contribution, we address the problem of automatically determining an optimal representation. We show that, in our case, for a given brick size and a fixed set of elemental brick types the representation with the lowest memory consumption can be easily determined. This is achieved by employing parallel initialization schemes for all of the different brick types to probe their final memory consumption. We also look into representations that trade memory consumption for access performance.

We evaluate our approach by comparing it to traditional bricking approaches. The memory consumption can be easily compared with other representations. The performance is tested for two different access patterns, i.e., stencil access and ray traversal. Our results show, that a combination of different brick types generally has a lower memory consumption than approaches using only two types, such as dense volume and empty bricks.

1.4 Overview of the Thesis

This work is structured in the following chapters. The related work presented in Chapter 2 is split into three main sections. We start with a brief introduction to Morton order, parallel reduction and binary search, since these are basic concepts that other publications widely use. Then we give an extensive summary of spatial data structures for two and three dimensional point data as the basis for our hybrid data structure. Finally we introduce partial evaluation as it is important for data aware just-in-time compilation.

The following three chapters deal with our data structure and its implementation. Chapter 3 introduces *hybrid bricking* with its advantages and drawbacks. We summarize and compare the properties of our four chosen brick types: the dense volume, the octree, the voxel list and the empty brick. Finally, we also introduce the concept of a modular data structure which was the basis of our approach during the implementation phase. In Chapter 4 we introduce *JiTTree* a just-in-time compiled data structure, which aims to increase the run-time performance of the proposed data structure. A detailed explanation on the efficient initialization of the novel data structure is given in Chapter 5. We also briefly cover lower level GPU programming concepts in OpenCL.

The last Chapters deal with the results and evaluation of our approach and covers our outlook for possible future work. In Chapter 6 we present the results. We test two different algorithms on 15 data sets and measure the memory consumption and the run-time performance for different representations of the data. Chapter 7, the summary, briefly recapitulates and evaluates our approach. Finally, we provide a short outlook to possible applications and extensions.

CHAPTER 2

Related Work

The first part of the related work chapter explains basic, but essential related concepts which can be seen as building blocks of our approach and of some work. The second part focuses on related work on spatial data structures to set our hybrid data structure in context to other work. The chapter concludes with a short section about partial evaluation as the basis for *JiTTree*.

2.1 Related Concepts

In the following we explain three basic concepts: Morton order, the parallel reduction and binary search. These are of relevance in our implementation and in related publications.

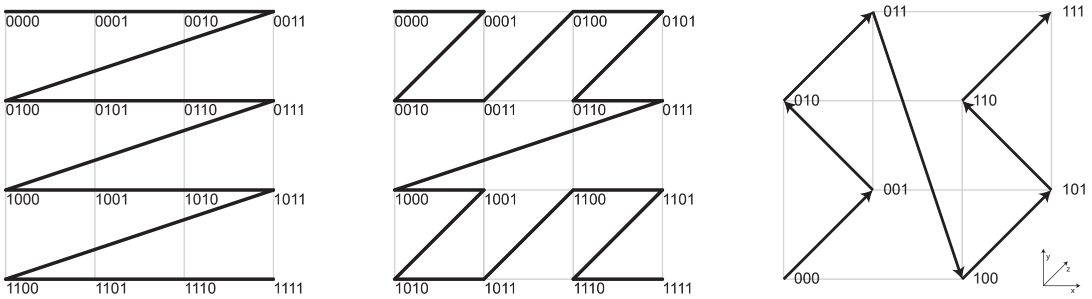
2.1.1 Morton Order

Typically, multidimensional data, has to be stored as a linear array in hardware memory. There are different approaches to transform a multidimensional index to a linear one. For example, Figure 2.1a shows a two dimensional data set, the values are sorted linearly along the black line. Each value is represented by its binary number. The linear array is filled in the following order: $[0000, 0001, 0010, 0011, 0100, ..., 1111]$. This sorting is called row-major order. A two dimensional coordinate $\mathbf{a}$ can be transformed to a one dimensional coordinate $\mathbf{b}$ with the following formula:

$$\mathbf{b} = \mathbf{a}_y width + \mathbf{a}_x \tag{2.1}$$

The problem with the row-major approach is, that values which are vertically adjacent in two dimensional space, are not adjacent anymore in its one dimensional representation. Since many hardware architectures (especially GPUs) are optimized to access data in a consecutive manner, performance of spatially coherent algorithms suffers from this simple coordinate transformation.

To tackle this common problem, Morton [33] proposed the Morton order or often called the z-order curve, shown in Figure 2.1b. A Morton ordered data set has a better probability that spatially close values are close in its one dimensional representation. This is advantageous in software and hardware caching scenarios, in parallel coalescing of memory access and in virtual memory and out-of-core approaches. In such scenarios, memories of different sizes with different access latencies exist. When accessing a part of a slow and large memory, only a subsection of the memory is moved to a faster and smaller memory. The spatial locality of values in Morton order decreases the probability of having to read memory that is far apart when accessing the neighbors of a data entry.



(a) Row-major order of a two dimensional data set

(b) Morton order of a two dimensional data set

(c) Morton order of a three dimensional data set

Figure 2.1: Orderings of two and three dimensional data

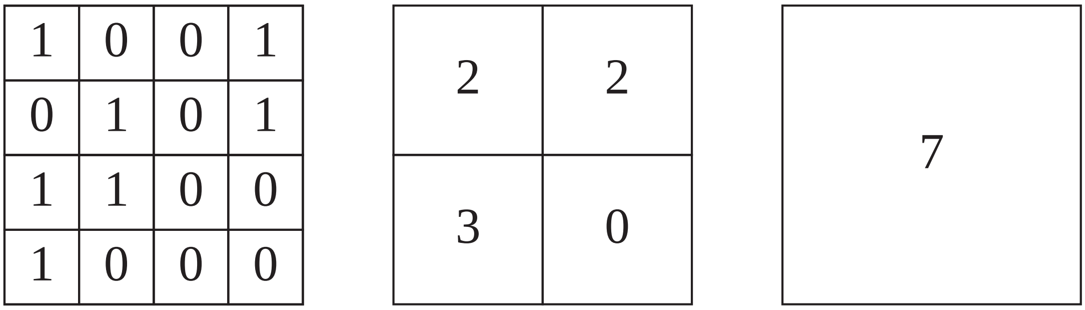
Finding the Morton code of a multidimensional coordinate can be achieved with bit shift operators as done by Baert et al. [1]. The code we use in our implementation to transform a coordinate to Morton order can be found in Appendix A.

Improving memory locality is not the only application of Morton ordering. A Morton order is also a depth-first order of a quad- or octree [1]. This is applied in related work such as in linear octrees by Gargantini [13]. In our case, we make use of the observation that the elements of a voxel list generated by histogram pyramids [50], are sorted in ascending Morton order. This allows us to perform binary search on such a sorted voxel list.

2.1.2 Parallel Reduction

The parallel reduction is a basic parallel algorithm. It can be used if it is required to reduce a large number of values to a smaller number of values by using an associative operation. A mapping of the parallel reduction to the GPU is introduced by Harris [18]. Summing up all intensities in an image, or finding the number of non-zero values in a volume, are examples for parallel reductions.

Figure 2.2 shows a two dimensional parallel reduction for the summation operation over all values in an image. In each iteration four values are summed up and stored in a reduced pyramid representation. The final iteration returns the sum of all values contained in the image. In this work, we use parallel reductions for finding the number of non-zero values inside volumes and for the generation of point lists as well as for the construction of octrees as introduced in the thesis of Ziegler [49].



(a) Original data set (b) First reduction pass (c) Second pass and final result

Figure 2.2: Parallel summation of a two dimensional data set using a pyramid approach.

2.1.3 Binary Search

Consider a sorted list of elements in the form $(key, value)$. key is a unique entry, which can be for example a one dimensional coordinate. $value$ is an arbitrary value, for example an RGB value in an image. There are multiple ways to retrieve the $value$ at a specific key (e.g., at a specific coordinate) from the list. The naïve approach is to iterate over all elements, until the key of interest is either found or the end of the list is reached. Accessing a key with the naïve approach requires $O(n)$ look-ups in the worst case.

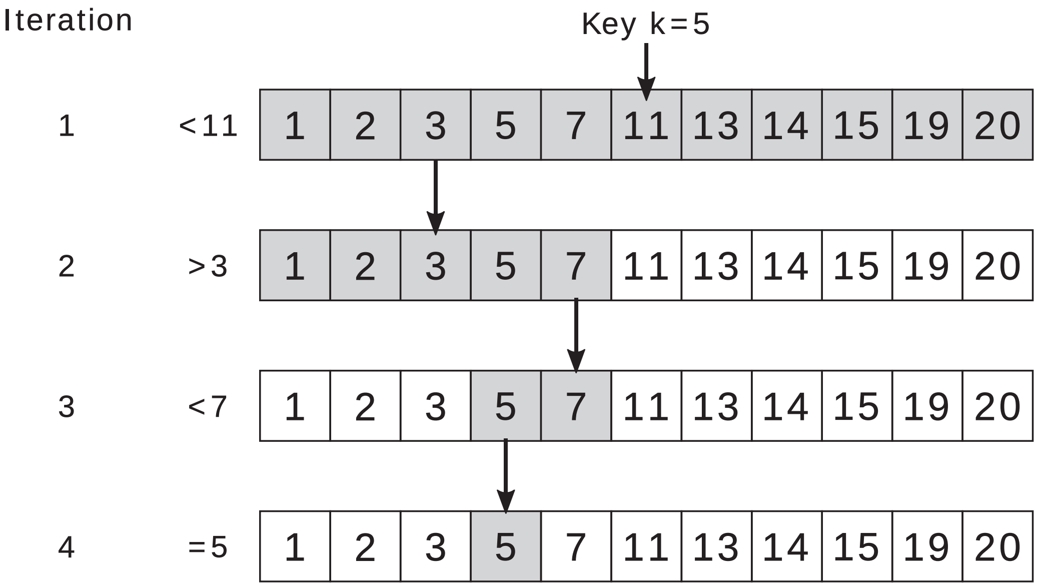


Figure 2.3: A sorted list of keys is searched for the key $k = 5$ with binary search. The example shows the worst case access. The values to each key were omitted for simplicity.

If dealing with sorted lists where random access is possible, the retrieval performance can be increased to $O(log(n))$ with binary search [25]. Binary search requires a linearly sorted list with respect to the key. In Figure 2.3 binary search is illustrated to find the key 5 in a sorted list of $keys$. The algorithm compares the key to the central element and decides in which half of the list to continue. If the list is reduced to one element, the algorithm checks if the $keys$ are equal. If so, the $value$ is returned. If not, the list does not contain the key.

We use binary search to increase the look-up performance of our voxel list implementation. In related work, the linear octree also uses binary search for memory access.

2.2 Spatial Data Structures

We now give an introduction to spatial data structures. An overview is given in the book written by Samet [41]. He distinguishes between data structures for point data, collections of rectangles, curvilinear data and a number of specific representations for volume data. In point data every entry has a fixed number of keys, which can be its spatial coordinate. Data entries can additionally contain values of arbitrary type. Rectangle data deals with bounding boxes or other rectangular objects. Curvilinear data represents the boundary representation of objects, for example a polygonal mesh. Finally, the chapter about volume data deals with very specific volumetric representations, such as constructive solid geometry (CSG), or binary space partitioning trees.

For our case, point data is the most relevant representation. While any sparse volume representation can be treated as spatial point data, typical volume data-sets, as represented in Figure 1.1, can not be easily represented by any of the other representations, such as for example as a polygonal mesh or CSG representation.

For point data, Samet summarizes the following relevant representations:

1. Non-hierarchical data structures
2. Quadtrees
    a) Point quadtree
    b) Region-based quadtrees: matrix quadtree and point-region quadtree
3. Kd trees
4. Bucket methods: hashing and the grid file method

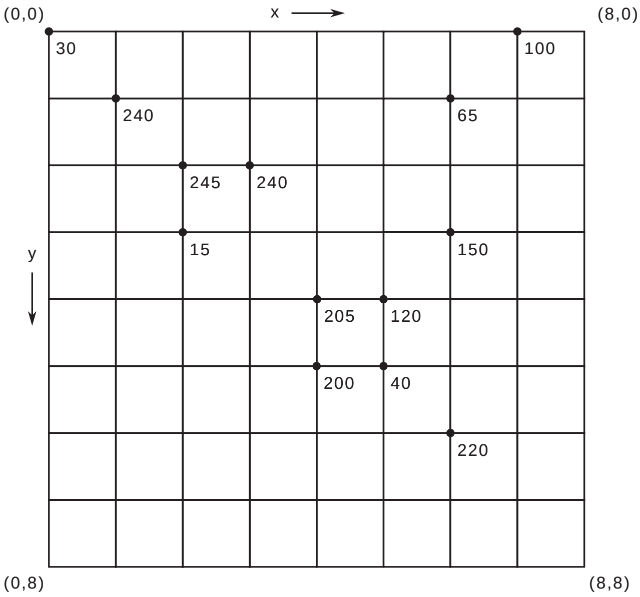
A more recent survey by Gaede and Günther [12] includes a number of more specialized trees and hashing schemes for spatial data. Both of these surveys only consider CPU based data structures. However, many of these representations were also implemented on GPUs. For example, the Octree was used on the GPU by Lefebvre et al. [29] or Crassin et al. [7]. Foley et al. [9] or Horn et al. [19] have implemented a kd tree on the GPU. Lefebvre and Hoppe [28] introduced perfect spatial hashing on the GPU.

2.2.1 Non-Hierarchical Data Structures

A list and a grid are non-hierarchical data structures. Retrieving a coordinate from a list of entries requires $O(n)$ look-ups, where n is the number of elements in the list. Figure 2.4a shows such a sequential list of two dimensional values. The fixed-grid method splits the spatial domain into equally sized cells (or bricks for a volumetric case). Each cell can either contain a single value or a linked list of entries. In our case, we are only interested in a single value per cell. This reduces the time to access a value at a specific coordinate to $O(1)$. Figure 2.10b shows a grid representation of the same data. The fixed-grid method only works well if the data is uniformly distributed. Our definition of a volume data-set is basically represented by the grid method where all values (i.e., also zero values) are stored in a single array.

x	y	Value
0	0	30
1	1	240
2	2	245
2	3	15
3	2	240
4	4	205
4	5	200
5	4	120
5	5	40
6	1	65
6	3	150
6	6	220
7	0	100

(a) Sequential list representation



(b) Grid representation (fixed grid)

Figure 2.4: Non-hierarchical representations

Binary search can be used to improve the random access performance of sorted lists to $O(log(n))$. When dealing with entries with multiple keys, e.g., entries located in two or three dimensional space, it is necessary to define a linear sorting over all keys. This can be done by sorting the elements linearly along a space filling curve. To efficiently sort a list on the GPU, *voxel radix sort* by Satish et al. [43] can be used. In our case, *histogram pyramids* [50] provide an alternative to sort the non-zero points in a volume in Morton order. The voxel list generated from a volume data-set via *histogram pyramids* is already sorted in Morton order. The details are explained in section 5.1.4.

2.2.2 Quadtrees and Octrees

The common property of all different types of quadtrees is that they recursively decompose two dimensional space into four quadrants. Octrees are the extension of quadtrees to three dimensional space.

Samet [41] distinguishes between the point quadtree and region-based quadtrees. Each node of the point quadtree, introduced by Finkel et al. [8], stores an x and y position, a pointer to its four children and a value. The point quadtree is similar to the kd tree, which only splits in a single dimension for each node. The drawback of the point quadtree is that it has a higher branching factor per node and a larger memory requirement in comparison to the kd tree as stated by Samet [41]. As a result, we do not cover the point quadtree in detail.

In the following we discuss the two predominant region-based quadtrees: the matrix quadtree (MX quadtree) and the point-region quadtree (PR quadtree). In more recent literature the MX quadtree is often called pointer quadtree.

MX Quadtree or Pointer Quadtree

The MX quadtree stores data points at discrete integer locations. Each data point corresponds to a 1×1 square. Possible data sets that can be represented by the MX quadtree are a fixed grid representation or the elements of a matrix, hence the name. Figure 2.5 shows the spatial representation of the MX quadtree for the example data set presented in the previous section. The data value of a point is associated with the upper left corner of a cell. The upper and left boundaries of each cell of this quadtree are closed. The lower and right boundaries are open.

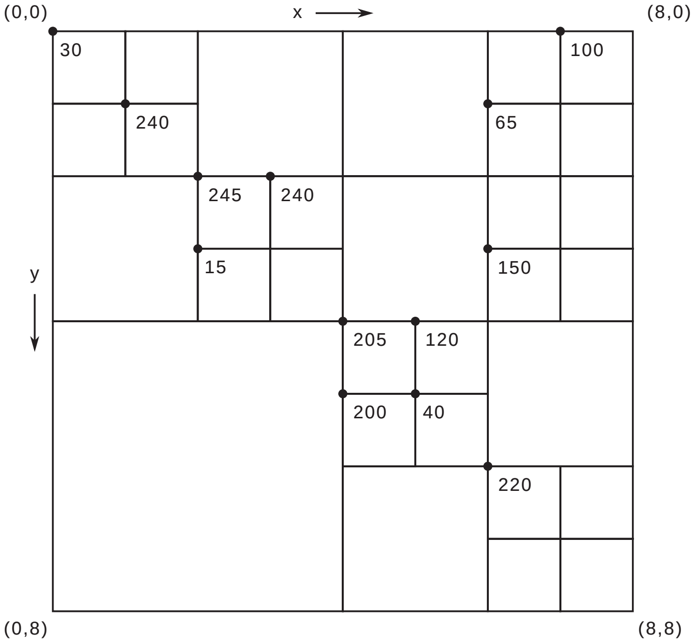


Figure 2.5: MX quadtree (*pointer quadtree*) spatial representation

The corresponding tree representation of the data structure is shown in Figure 2.6. Four pointers are stored at every internal node and four data values at each leaf node. Internal nodes are either gray, in this case they have at least one non-zero child node, or white, if all their children are zero. Leaf nodes that contain a value are indicated with a black dot in Figure 2.6.

In the most basic case, this node data structure is implemented by having four pointers and four data values at every node. For internal nodes the data values are zero, for leaf nodes the pointers are null. An important purpose of using a quadtree is to reduce the memory requirement of a data set. Often two different node types are introduced: internal nodes and leaf nodes. This results in a more compact overall representation.

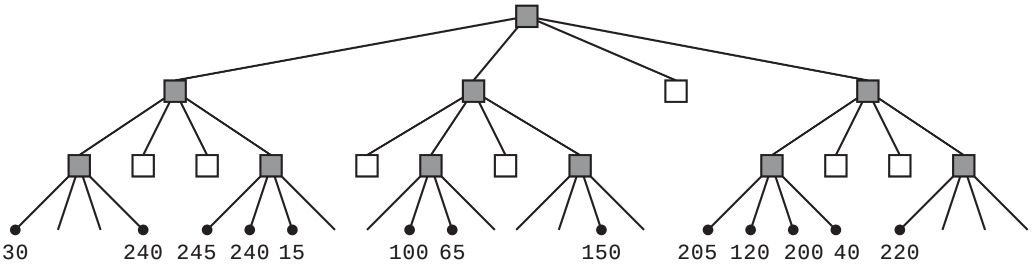


Figure 2.6: MX quadtree tree (*pointer quadtree*) representation

A further suggested memory optimization, as mentioned by Wilhelms [45], is to store only a single pointer and arrange the children of each node in contiguous memory. Nevertheless, this idea does not reduce the memory requirement of such a tree when using a pointer octree. Lets consider the example in Figure 2.6 and calculate the total memory requirement of both cases. We assume that each pointer and each value requires one memory unit of storage.

(a) In the traditional implementation, we store four pointers at each grey node, we get a total of $10 \times 4 = 40$ pointers. The internal nodes one level below the leaves, can directly store the values instead of pointers to the leaf values, therefore we require 40 memory units of storage space.

(b) If we store one pointer for each node and keep the child elements in contiguous memory, we only require one pointer per internal node, that is 17 pointers. The drawback is, that we have to store all leaves in contiguous memory, even the zero ones. These are 24 leaves. In total we have to use $17+24 = 41$ memory units of storage space. Since we can assume that the root node points to the first level of the tree, we can omit the first pointer and thus require 40 units of memory space.

Even for the more general case, i.e., if the memory requirement of the data values and the pointers differ, the memory requirement of both implementations is the same.

In more recent literature, such as the survey by Knoll [23], the MX quadtree is called *pointer quadtree* or *pointer octree* for the three dimensional case.

PR Quadtree

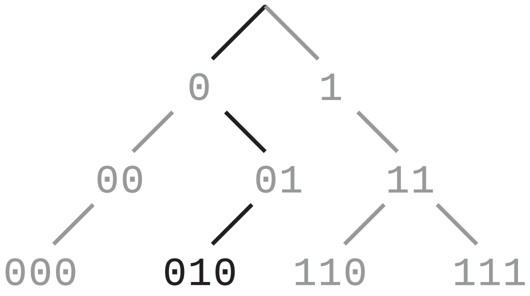
In contrast to the MX quadtree, the PR quadtree can store a single point anywhere inside a leaf cell. This point is not necessarily located at the position corresponding to the integer coordinate of the leaf cell, but at an arbitrary location covered by the leaf cell sub-region. To encode its exact location each leaf node containing a non-zero entry also has to store its x and y position.

Octree Variants

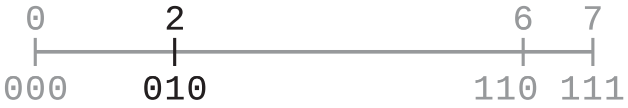
Variants of octrees for volume rendering are discussed in the survey by Knoll [23]. Besides the pointer octree, he introduces octree hashing and mentions the full octree, the linear octree and the branch-on-need octree. The full octree stores all of its leaves and if needed also all the internal nodes, therefore no pointers have to be stored, such an octree is also called *pointerless octree*. In the following we will explain the linear, the branch-on-need octree and octree hashing.

All these methods make use of so called **locational codes** which are closely related to the Morton code. Locational codes encode a specific node in a quadtree, an octree or binary tree. An illustrative example of a locational code for a special binary tree is given in Figure 2.7. Similarly to the quadtree, this binary tree always splits a spatial range in the middle.

To encode a leaf of this binary tree with its locational code, we start at the root of the tree. For each tree traversal step 0 denotes a traversal to the left and 1 a traversal to the right sub-tree. Let us assume that the leaves of this binary tree encode the integer locations between 0 and 7 on a one dimensional number line. The decimal representation of each leaf now corresponds to the position in the interval between 0 and 7. For example the leaf 010 corresponds to the position 2 on this number line.



(a) Locational codes for internal and leaf nodes.



(b) Linear representation of the leaf nodes.

Figure 2.7: Locational codes for a binary tree

Linear Quadtree

The linear quadtree was introduced by Gargantini [13]. It stores all non-zero values in a one dimensional array in the order if its locational code. Figure 2.8 shows an example of such a linear quadtree. A so called quaternary code, the two dimensional extension of the previously explained locational code, is used. It encodes each quadrant with either of the numbers 0, 1, 2 or 3 in base 4. Each leaf node stores the full quaternary code for traversing down the quadtree as its key. Lets consider the element 13 in Figure 2.8. Starting from the root of the quadtree, the element lies in Quadrant 1. Inside Quadrant 1, the element lies in Quadrant 3. The locational code of the element is 13.

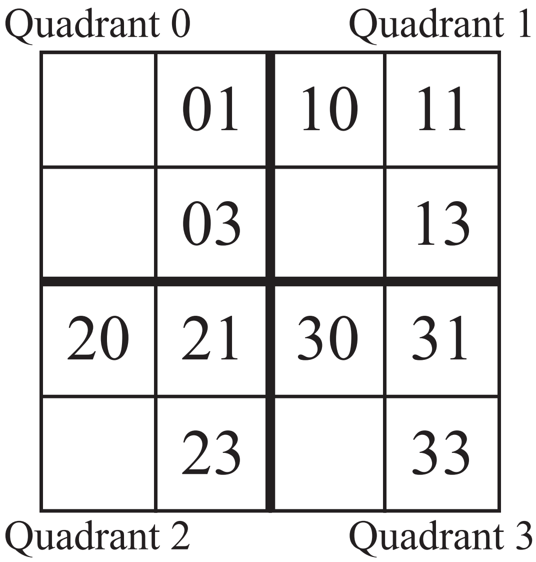


Figure 2.8: Linear quadtree representation using quaternary numbers of base 4.

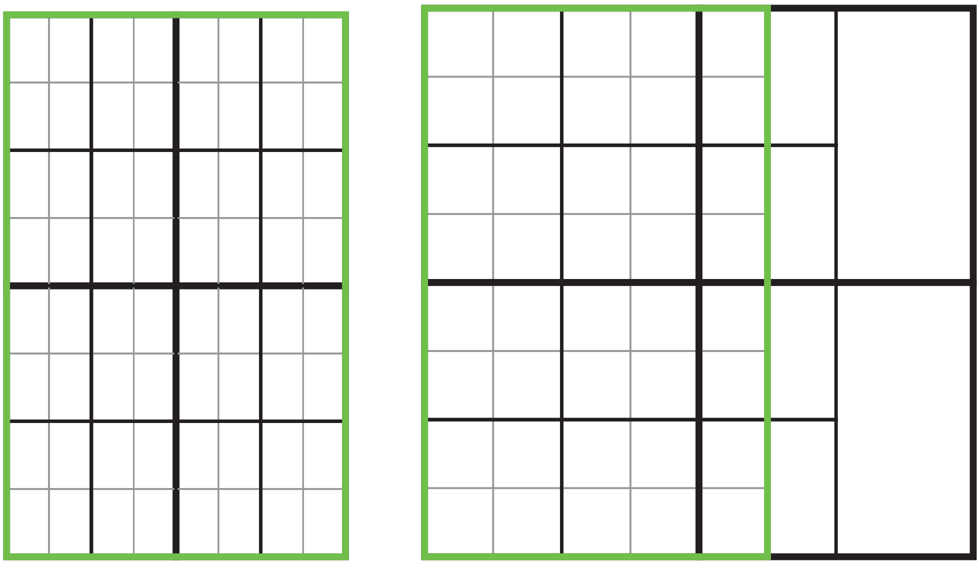
The quaternary codes can be linearly sorted with respect to their decimal value. For example: $01 < 03 < 10$. To retrieve a value from a linear quadtree, the look-up coordinate has to be transformed to quaternary code. A binary search on the sorted list of leaves, i.e., the non-zero elements of the data set, is applied to retrieve a specific value. As a result, the look-up performance now depends on the number of non-zero elements and not on the side-length of the data. A linear quadtree has an access performance of $O(log(n))$, where n is the number of non-zero leaf nodes. In comparison to the pointer quadtree, the linear quadtree has a lower memory consumption, but a higher runtime for most cases.

The linear octree is an extension of the linear quadtree to three dimensions. In our work, the voxel list access is very similar to the one in a linear octree. The quaternary code in base 2 is exactly the Morton code of the pixel as described in Section 2.1.1.

Branch-On-Need Octree

The branch-on-need octree was introduced by Wilhelms et al. [45] for volumes that are non-power-of-two. They use their octree for increasing the performance of isosurface generation. Each internal node of the tree stores the minimal and the maximal value of its corresponding sub-tree. In the following we explain the layout of the octree rather then their algorithm for generating isosurfaces.

When storing a full octree for non-power-of-two volumes, two possible split strategies can be observed. Either the octree splits the volume always in the middle, or the octree is constructed



(a) The traditional quadtree has a larger number of nodes.

(b) The branch-on-need quadtree requires fewer nodes.

Figure 2.9: Branch-on-need quadtree and traditional approach, the green region is the actual size of the data set.

for the next power-of-two cube that fits the data set. The branch-on-need octree applies the latter strategy. The problem with using the next power-of-two volume for such an octree is that values which fall outside the volume of interest are still stored inside the full octree. Figure 2.9 illustrates these two strategies.

The branch-on-need octree only stores values which are located inside the original volume data-set. As a result, it is more difficult to find the location of a specific node inside the branch-on-need octree data structure.

For a specific volume size, the number of nodes at each octree level can be read from the side-length. Lets say our volume is of size $5 \times 5 \times 4$, the theoretical octree required to store such a volume would be of size $8 \times 8 \times 8$. The binary locational code of the maximal position in binary contained in the octree is: 100, 100 and 011. From these values it is possible to calculate the number of nodes per octree level. Level 1 is the root level, it requires one node. At level 2, we consider the most significant bit and add one each, i.e., the octree needs $(1+1)*(1+1)*(0+1) = 4$ nodes. At level 3, we consider the two most significant bits, the octree needs $(3+1)*(3+1)*(1+1) = 32$ nodes. And finally at level 4 the octree has $(4+1)*(4+1)*(3+1) = 100$ leaf nodes. For a more detailed explanation we refer to the paper [45].

Finally, for each node, the branch-on-need octree stores a three bit code that defines whether each of the eight children branches and a pointer to the first child.

Octree Hashing

Frisken el al. [11] introduce a hashing scheme for octree traversal to improve performance. They use locational codes similar to the linear octree. In their case, three locational codes are generated for each dimension (x, y and z) separately like in the binary example in Figure 2.7.

They store a locational code at each octree node for each coordinate (x, y and z). This locational code refers to the minimal corner vertex of the node. For example, an octree of size $8 \times 8 \times 8$ requires three bits to encode the side-length. The minimal corner vertex of the root is at the location $x = (0)_{10} = (000)_2$, $y = (0)_{10} = (000)_2$ and $z = (0)_{10} = (000)_2$. For the node in the first octant this locational code is $x = (4)_{10} = (100)_2$, $y = (0)_{10} = (000)_2$ and $z = (0)_{10} = (000)_2$. For each leaf node, this code is exactly the location of the leaf in the volume.

When doing a look-up in the data structure, the look-up coordinate is seen as a binary locational code representation. For example, the coordinate $(3, 4, 5)_{10}$ would correspond to $(011, 100, 101)_2$. During the traversal of the tree, the index of the child node where the look-up position is located, is determined by bit shift operations. To do this, the three binary locational codes of the look-up position are compared bit-by-bit with the locational codes of each octree node. The index to the corresponding child node can be directly obtained by doing bit arithmetics on the locational codes. This achieves a speed-up compared to doing the traditional boundary checks in octree traversal.

Knoll et al. [24] use the approach of Frisken et al. [11] to achieve a lossless compression of volume data. Using such an octree, they apply a CPU based volume renderer for large, time varying data.

GPU Octrees

An efficient representation of a pointer octree is to represent each node either as an internal or as a leaf node. For a sparse data set, internal nodes can have a varying number of children (i.e., from zero to eight). In contrast to internal nodes, leaf nodes store from one to eight values instead of pointers. These requirements result in different approaches to represent such a pointer octree on the GPU.

First GPU implementations of sparse pointer octrees are found in the work of Lefebvre et al. [29], who use octrees for surface texture encoding. They introduce a data structure called N^3 tree. The tree only distinguishes between either N^3 children or no children per node. N can be an arbitrary integer (e.g., for an octree $N = 2$). To manage their data structure on the GPU they rely on RGBA 3D textures. The pointers or values of the data set are encoded in the RGB channels, the alpha channel is a flag that indicates if a node is an internal or a leaf node. The traversal of the octree is done in a fragment shader.

More recently Crassin et al. [7] use a pointer octree for volume rendering. They also distinguish between internal nodes and leaf nodes. Internal nodes are stored in a node pool and leaf nodes in a brick pool. For every internal node, they always store all eight child nodes in successive regions of the memory. This allows them to only store one pointer (i.e., to the first child node) per internal node. Since they also have to store the first node marked as empty, this requires the same memory as storing eight pointers per node. Leaves are cubicals of values with side-length M which are stored in the brick pool. The size of these leaf bricks for their test cases lay between 16^3 and 81^3, but they can be arbitrarily defined.

Octree Construction

Recently, algorithms were formulated to generate octrees on the GPU. For instance, Lauterbach et al. [27] 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 [49]. The basis of the octree construction algorithm are *histogram pyramids* [50] also used for generating voxel lists. We have based our voxel list and octree generation on this parallel algorithm.

2.2.3 Kd Tree

The kd tree or multi-dimensional binary search tree was introduced by Bentley [2]. Each node of a kd tree for two dimensional data contains two pointers one to the left and one to the right child, an x and y coordinate, an entry (or value) and the information whether it splits in x or in y direction. Often it is assumed that a kd tree splits along the dimensions in a specific pattern, for example alternating x and y. If this is the case, it is not necessary to store the split direction for each node. Figure 2.10 shows such a two dimensional kd tree, storing data points from A to F.

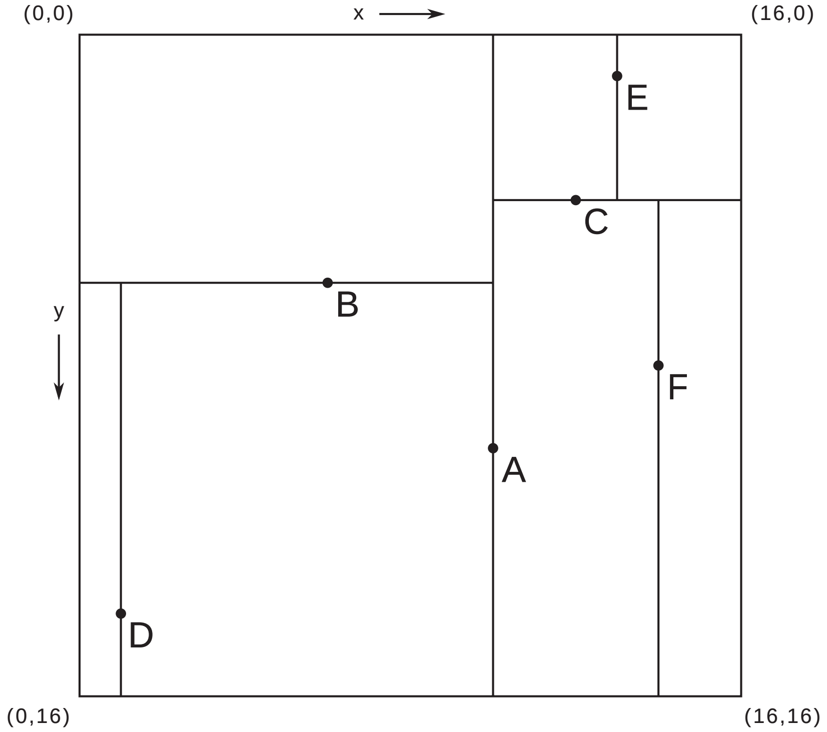
The most simple strategy to generate a kd tree assumes that each point is added subsequently to an empty tree. The problem with this approach, is that this does not result in a balanced tree and it is difficult to balance an unbalanced kd tree. A simple method to construct a balanced tree from a number of points is to always split at the median element of the points sorted with respect to the split direction.

A variant is the adaptive kd tree by Friedman et al. [10] that stores values in the leaves of the kd tree only. Another option is to interpret the leaves of a kd tree as the regions covered by the leaf.

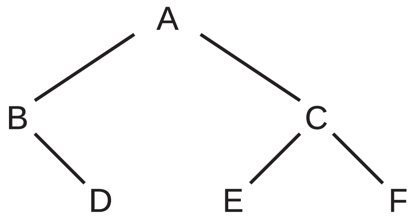
Foley et al. [9] use a kd tree acceleration structure for rendering triangle meshes on the GPU. They propose two stack-less routines for ray traversal, called the kd restart and the kd backtrack algorithm. In traditional kd tree traversal, one ray is intersected with the volume covered by a kd tree. A stack is used during the recursive traversal of the tree to keep track of branches that are traversed later. The kd restart algorithm eliminates the requirement of having such a stack, while increasing the run-time. After reaching a specific leaf of the kd tree it simply proceeds to traverse the tree from the root. The kd backtrack requires each node to store a pointer to its parent. This makes it possible to jump back to the parent node, if the current child node has been successfully processed.

2.2.4 Bucket Methods, Hashing and Bricking

Bucket methods were originally developed to hide access latency to external storage. They split the region of interest into buckets, the content of a bucket can be loaded from disk on demand. Each bucket can contain up to a specific number of values, this is called the *bucket size*.



(a) Two dimensional kd tree



(b) Tree representation of the kd tree

Figure 2.10: Kd tree

Bucket methods can be divided into non-hierarchical and hierarchical methods. The most basic non-hierarchical bucket method, is the fixed grid method, which can store multiple entries per grid cell. A problem of this method is underflow and overflow as stated by Samet [41]. Many cells may remain empty, others can contain a large number of values.

In volume rendering, empty grid cells can be discarded during ray traversal. A related concept to the fixed grid method is volume bricking, for example Parker et al. [35] describe an approach for CPU ray casting. The volume is split into *bricks* of the same size. An empty brick does not need to be stored. Therefore the approach reduces the memory requirement. In addition, bricking results in a better memory locality when caching nearby values.

Bricking on the CPU was used to improve cache coherency by Grimm et al. [14]. Kähler et al. [21] 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. [40] 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. Another related approach was presented by Purcell et al. [36]. They use a bricking approach where each brick contains a list of values per cell for photon mapping.

Other non-hierarchical methods are linear and spiral hasing. More recently hashing was introduced on the GPU in *perfect spatial hashing* [28]. The advantage of this approach is the fast access. Nevertheless, the generation of a suitable hash function and an offset table can be computationally demanding.

The Grid File method by Nievergelt et al. [34] introduces an interesting concept. It is similar to a volume bricking method (with varying brick boundaries), but bricks can store a pointer to shared data. So, two bricks could point for example to the same list of data values.

In contrast to non-hierarchical methods, hierarchical bucket methods have to traverse a tree to get to the actual buckets. Two of the previously mentioned methods can also be implemented as a bucket method, the PR quadtree and the adaptive kd tree. Both of these methods can store multiple data points in a leaf.

2.3 GPU Program Optimization and Data Structures

Our approach targets to optimize data structures for the GPU. Others have proposed similar approaches in the context of scientific visualization. Glift [30] provides an abstraction layer to build data structures on top of OpenGL. The library was used to define 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.

More recently, domain specific languages (DSL) were introduced as an abstraction layer to reduce the difficulty of implementing certain sets of algorithms in visualization and data processing. Diderot [3] defines a high-level DSL that was used for image analysis and visualization algorithms, such as ray-casting, line integral convolution and isocontour detection. This approach provides high flexibility to develop a variety of algorithms. A user of Diderot can write abstract code which is than translated either to sequential or parallel C code or to OpenCL code. They compare the performance of their approach to code using the Teem library and show that their approach is faster. A basic volume rendering in their DSL can require from two to 15 seconds. Vivaldi [4] is an extension of Diderot to multi-GPU systems. By using up to 12 GPUs they are able to improve performance.

The Halide [37] DSL was introduced for image processing, in specific to speed up stencil operations. They are interested in subsequent image processing steps, so called image processing pipelines. A basic example is a 3×3 separable filter, which is applied first in x then in y direction. They introduce a number of possible execution patterns. The two extrema of their approach are either a so called breadth first execution or the loop fusion approach. In case of the breadth first approach, they execute the operation first in x direction over the whole image and store it in a temporary array. Then they perform the operation in y direction starting from the previously stored array to generate the result. The other extreme solution, the loop fusion, is to calculate the value locally by iterating over all eight neighbor values, this avoids storing the intermediate result, but introduces redundancy in the processing step. Their intermediate approaches between these two extrema execute the operation in sub-regions of the data, so called overlapping tiles. Alternatively, they apply a so called sliding window approach, that transforms the approach into a sequential operation by reusing previously calculated results for example along scan-lines in the image. These possible tuning options are modeled as a *choice space* to vary the execution of the processing pipeline. Their auto-tuning optimizers intelligently sample the search space of possible execution patterns to find a high performing solution. In their results, they show that their auto tuning algorithm can outperform hand-optimized x86 code and also one case of CUDA code.

2.4 Partial Evaluation and Deferred Compilation

Our work is based on the partial evaluation of programs that is described by Jones et al. [20]. We partially evaluate the GPU programs for performance reasons but also to make our data structure more flexible. More recently Rompf et al. [38, 39] and Sujeeth et al. [44] 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.

The idea of data aware compilation is based on partial evaluation. A compact tutorial on partial evaluation is given by Consel and Olivier [5]. According to them, partial evaluation is a source-to-source program transformation technique for specializing programs with respect to their input. They split the input fed to a program into *static* input and *dynamic* input. A program can then be specialized with respect to the static input.

In the following we list the simplifications a partial evaluator can perform according to Consel and Olivier. These simplifications are done by most modern compilers. Nevertheless, partial evaluation can be done at arbitrary stages, either during the compilation of a program or even at the run-time, when additional information is known. In the case of our data structure, we treat the data itself as static input to the just-in-time compilation stage.

2.4.1 Constant Folding

Constant folding is typically done by the compiler by evaluating constant expressions already at compile time. An example is given in Listing 2.1. The arithmetic expression can be evaluated to a single value and does not need to be evaluated during run-time.

```
// original code
value = (10 + 15) * 10;
```

```
// after constant folding
value = 250;
```

Listing 2.1: Constant folding

2.4.2 Constant Propagation

Constant propagation propagates known values during compile time. An example is given in Listing 2.2. If the variable varA is not used in the succeeding code it can be removed entirely by dead code elimination.

```
// before constant propagation
int varA = 250;
int varB = varA * 2;
```

```
// after constant propagation of varA
int varA = 250;
int varB = 250 * 2;
```

```
// after constant propagation of varA and constant folding of varB
int varA = 250;
int varB = 500;
```

Listing 2.2: Constant propagation

2.4.3 Function Inlining

Inline functions can be also partially evaluated. Listing 2.3 shows such an inlining of functions.

```
// before function inlining
add(int x, int y)
{
  return x + y;
}
int value = add(100, 0);
```

```
// after inlining the add function
int value = 100 + 0;
```

Listing 2.3: Function inlining

2.4.4 Specialization of Functions

A slightly more complex scenario is the specialization of functions through partial evaluation. For this example, the function given in Listing 2.4a is a recursive function for x^n, where x and n are of type integer. When calling this function with a constant $n = 2$, it is possible to unroll the recursion.

In Listing 2.4b three different specialized functions are shown as intermediate steps for evaluating all power functions for $n = 0, 1, 2$. Listing 2.4c shows the final result. This function call should be much faster than the one shown in Listing 2.4a. It has no recursive method calls and no additional if/else condition for each recursion.

```
int pow(int n, int x)
{
  if (n <= 0)
  {
    return 1;
  }
  else
  {
    return x * pow(n-1, x);
  }
}
pow(2, varA); // specific method call
```

(a) Original code.

```
int pow_0(int x)
{
  return 1;
}
int pow_1(int x)
{
  return x * pow_0(x);
}
int pow_2(int x)
{
  return x * pow_1(x);
}
pow_2(varA);
```

(b) Unrolling of all intermediate steps.

```
int pow_2(int x)
{
  return x * x;
}
pow_2(varA);
```

(c) Final result.

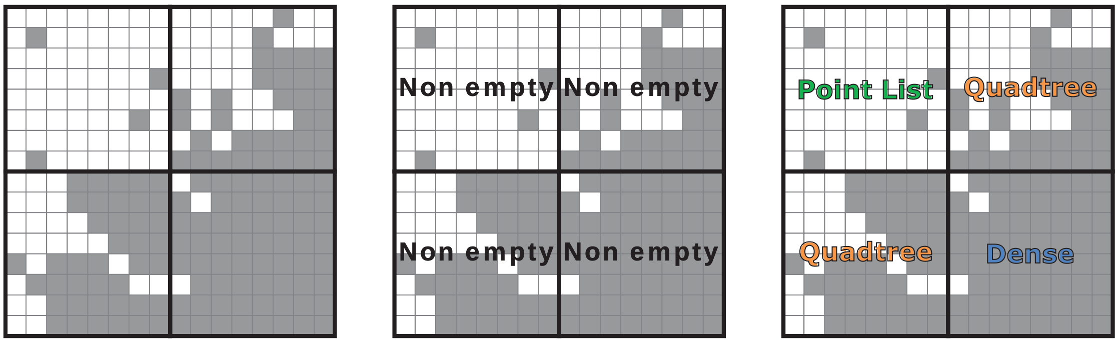
Listing 2.4: Special functions by residualizing calls.

For these low-level compiler optimizations, we rely on the OpenCL compiler. However, for higher level optimizations we employ our data aware compilation stage that is explained in Chapter 4.

CHAPTER 3

Hybrid Volume Bricking

The goal of our data structure is to adapt to the local sparsity of a volume data-set. We aim to store only non-empty values and to avoid to store zero values in the data set. A data set typically contains sub-regions 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 and octrees, while dense regions are most efficiently stored in a dense representation. To split data into sub-regions we apply a volume bricking approach. The advantage of bricking, as a non-hierarchical method, is that it makes high performance access possible. In contrast to hierarchical methods such as octrees or kd trees, the access cost of bricking is only a single indirection. Contrary to traditional bricking, we allow multiple different data types in every brick. We call this a *hybrid bricking* approach. The advantage of hybrid bricking over traditional bricking, is that we are able to pick a memory efficient data structure for each individual brick.



(a) Splitting of a two dimensional data set into four bricks. White pixels are zero, gray pixels non-zero.

(b) Bricking approach with two different brick types - empty and non empty.

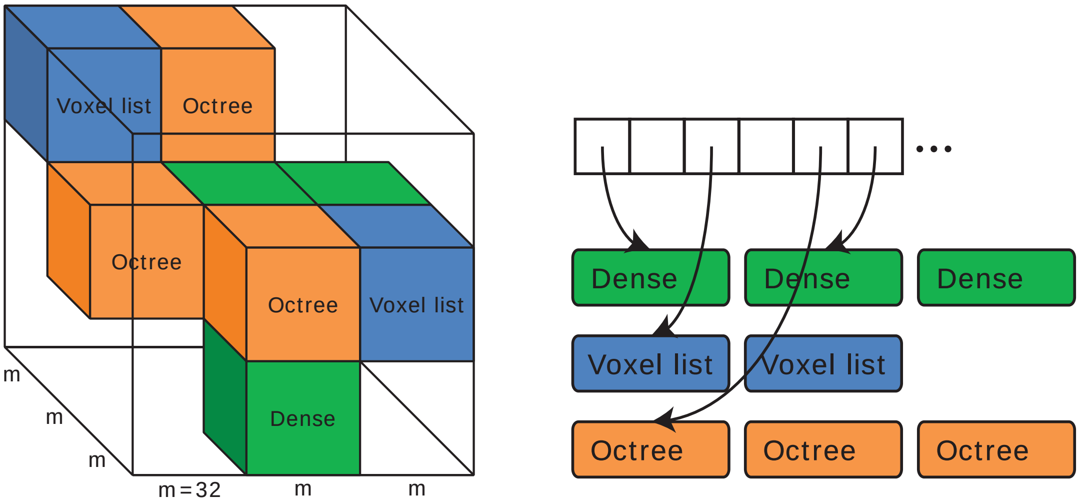
(c) Bricking approach with four different brick types - empty, point list, quadtree and dense.

Figure 3.1: Conventional bricking compared to hybrid bricking for two dimensional data.

Figure 3.1 demonstrates the advantage of our data structure over a conventional bricking. The conventional approach with two brick types only differentiates between empty and non empty regions. In case of the top left brick it has to store the full non-empty brick, even though it contains only four values. When using a larger number of different brick types, as shown in Figure 3.1c, the representation can better adapt to the data and therefore reduce the memory requirement.

For our hybrid data structure we chose a combination of four different representations: empty, voxel list, octree and dense. Each representation performs best for a different level of sparsity. We describe these *elemental brick types* from lowest to highest sparsity in Section 3.1.

Figure 3.2 shows an conceptual overview of our hybrid bricking approach. A constant brick side-length m is used. As shown in Figure 3.2b, the bricking is implemented as an array that contains the type of each brick and a pointer to the region of memory that holds the corresponding data. The specifics of the data structure layout and traversal are explained in Section 3.2.



(a) Volume

(b) Pointer structure

Figure 3.2: (a) shows a volume that is split into 3^3 bricks each of size m^3. Every brick holds one of the elemental brick types: dense, voxel list, octree or empty. In (b) the basic data-layout is shown. The volume consists of an array of bricks which point to elemental leaf nodes.

As a final remark, our approach assumes that the data set does not contain large homogeneous regions besides the empty regions. We do not make use of image compression techniques that try to reduce the memory requirement by efficiently encoding the value range. Doing so, would open up a large number of different possible techniques. However, compression is an orthogonal concept to our approach and we could introduce compressed brick types as new elemental node types. One example of such a situation is a homogeneous brick where all voxels are of the same value. In cases like this we could define a new elemental data structure that only stores a single value for a whole brick.

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

Table 3.1: Theoretical comparison of the properties of different elemental brick types. m is the side-length a data block. n is the number of elements inside the data block. d is the size of one data entry in bits. p is the pointer size and needs to be at least $\log_2(\frac{24}{7}m)$ to address all possible points in one continuous memory region.

3.1 Brick Types

Depending on the density of non-zero data values inside a sparse volume, a specific data structure minimizes the memory requirement to represent the data. In this work we consider four elemental brick types:

3.1.1 Dense Volume Brick

Dense volume bricks, colored green in Figure 3.3, are a good representation for regions of high density. They store all data values in one array. Empty values are also stored in this array. In our case a one dimensional array is used. Each three-dimensional spatial location is transformed to a one dimensional array position using the row-major order.

In contrast to octree and voxel list, no additional information has to be stored per value to access the data structure. If we disregard the overhead of bricking, the dense volume has the following bit overheads per data entry as also shown in Table 3.1. For the best case, a full volume, the memory overhead per data entry is zero. The overhead of the worst case, a single non-zero voxel in an otherwise empty volume, is the number of zero values we unnecessarily store in addition to this single value. As a result, the dense volume representation performs very well for dense regions and fails for extremely sparse regions.

When interested in look-ups at non integer positions, as for example in volume rendering, trilinear interpolation is often used. A dense representation can easily be stored as a three-dimensional texture on graphics hardware. This allows the hardware to perform fast trilinear interpolation between eight values.

A major advantage of a dense grid is fast access. In total, such a dense look-up requires one indirection through bricking and a constant, $O(1)$ array look-up. In some cases it makes sense to favor a dense representation over an octree or voxel list because of its high access performance.

3.1.2 Octree Brick

Octrees are an adaptive, hierarchical data structure. The traditional *pointer octree* (or MX octree) performs best for medium sparse data sets. Other octree variants perform well for very dense

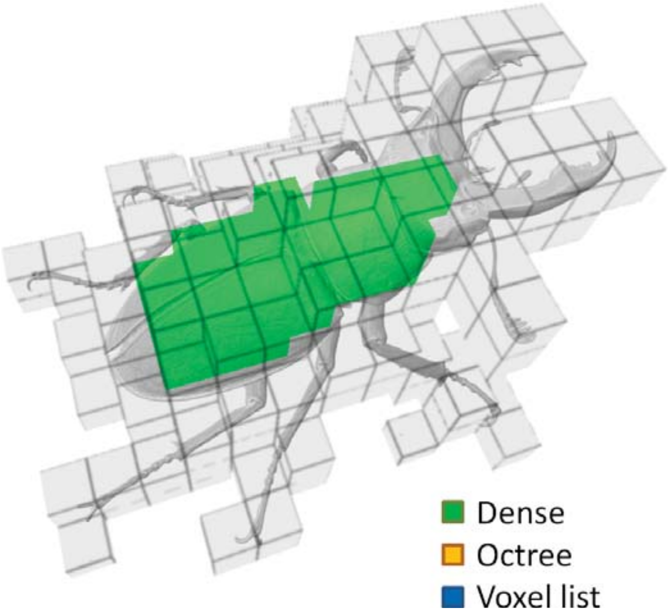


Figure 3.3: Dense $64 \times 64 \times 64$ bricks of the stagbeetle data set.

data such as the full (or pointerless) octree and some variants perform well for very sparse data such as the linear octree.

We have chosen to implement a traditional *pointer octree* since it provides an intermediate solution between the dense and the voxel list brick type. Each node of the tree stores exactly eight pointers to its children. Pointers are indices into a one dimensional *pointer array*. In our case the leaf nodes store bricks with eight values, which are located in a corresponding *value array*. Octree bricks are shown in orange in Figure 3.4. They occur mainly at medium sparse regions of the data set at the boundary of the beetle object.

The bit overhead per entry in an octree is not as good as the dense volume representation in the best case. A full *pointer octree* requires $(m^3 - 1)/7$ internal nodes, where m, the side-length of a brick, is a power-of-two. The full octree corresponds to the lower bound in Table 3.1. Each leaf node shares its parent nodes with all the other leaf nodes. The worst case occurs if we store a single value in an octree. Such a pointer octree requires at least $log_2(m)$ pointers. In our implementation the worst case memory consumption requires $8(log_2(m) - 1)$ pointers and eight values, this corresponds to the upper bound in Table 3.1. The voxel list clearly has a lower memory consumption for such extremely sparse cases.

The access performance of a pointer octree depends on the side-length of the volume and not on the number of elements with a worst case access of $O(log(m))$ traversal steps. As a result, in very sparse cases, voxel lists can also outperform the octree with respect to access performance. An important property of pointer octrees, is that they are shift sensitive [42]. The number of internal nodes in such an octree depends on the exact location of the octree and not only on the density of the data set. Slightly shifting an octree can result in a lower memory requirement. An

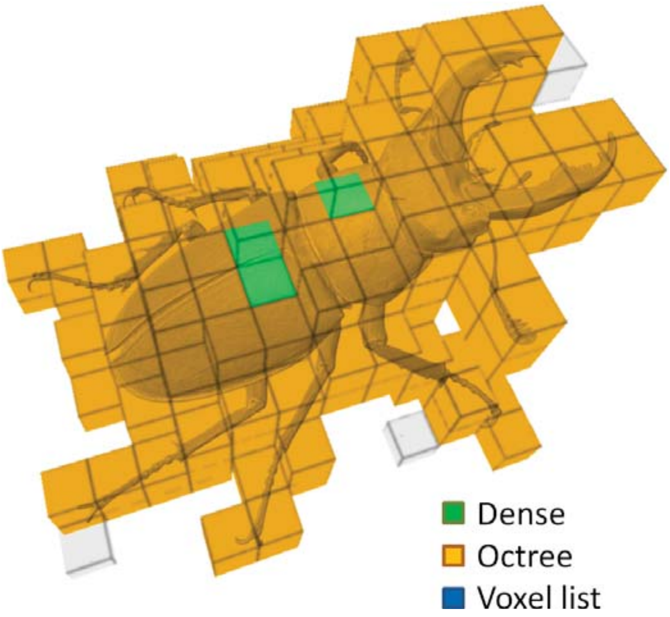


Figure 3.4: Octree $64 \times 64 \times 64$ bricks of the stagbeetle data set.

advantage of octrees is that they enable empty space skipping for ray casting [9].

3.1.3 Voxel List Brick

Voxel list bricks are represented as a set of quadruples $(x, y, z, value)$. They perform well for regions of high sparsity and are shown in blue in Figure 3.5.

Each of the x, y, z coordinates theoretically requires $log_2(s)$ bit where s is the side-length of the volume. In our case we have chosen 16 bit coordinates, since a value range of 65536 was high enough for our data sets. Voxel lists require a constant overhead per entry, in our case 48 bit. Clearly this overhead is lower than the overhead that is necessary when storing a single value in either a dense volume or a pointer octree.

A major drawback of naïve voxel list implementations is the high access time of $O(n)$ look-ups in the worst case, where n is the number of elements in the list. To increase the look-up performance to $O(log(n))$ we use binary search. The prerequisites for applying binary search are that the list entries are sorted and that random access is possible. We guarantee random access by storing the voxels in an array. The sorting of the list elements is explained in the following.

We use histogram pyramids to construct a sorted voxel list from a dense volume as discussed in section 5.1.4. The algorithm specifies the position in a list by traversing the intermediate steps of a reduction on a binary representation of the data. To find the position, the algorithm iterates through the reduction levels and sums the values in a specific order. By making sure that the order of summation is as shown in Expression 3.1 we guarantee that the resulting voxel list is

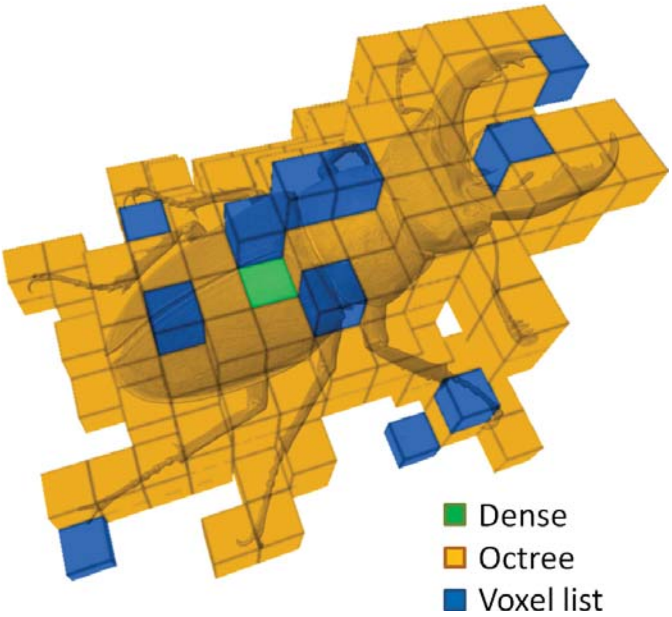


Figure 3.5: Voxel list $64 \times 64 \times 64$ bricks of the stagbeetle data set.

sorted in Morton order.

$$(0,0,0) \rightarrow (1,0,0) \rightarrow (0,1,0) \rightarrow (1,1,0) \rightarrow (0,0,1) \rightarrow (1,0,1) \rightarrow (0,1,1) \rightarrow (1,1,1) \tag{3.1}$$

We can exploit this sorting by transforming every three-dimensional (x, y, z) coordinate into a one dimensional Morton code using the routine given in Appendix A. Finding a coordinate in the point list can now be done with the following steps. We first initialize the voxel list in Morton order using histogram pyramids. Every three-dimensional (x, y, z) coordinate in the list is than transformed to a one dimensional Morton code and stored instead of the (x, y, z) coordinate. Our list elements are now sorted with respect to its Morton code. When doing a look-up, we transform the look-up coordinate into Morton code. Now we can apply binary search, by finding the look-up coordinate in the sorted voxel list. The general approach is also called *linear octree* [13] in the literature. Our initialization is done more efficiently on the GPU. Voxels lists provide a high access performance if the list only contains a low number of elements.

3.1.4 Empty Brick

Empty bricks can be used if a brick contains only zero values. They are the most beneficial brick type. An empty brick only stores its type. When accessed they return zero, no indirection is necessary. As a result they provide the highest access performance among the four brick types. They also possess the lowest overhead per entry in bit.

3.1.5 Choosing an Optimal Brick Type

As mentioned earlier, our selection of brick types can handle a wide range of different sparsity characteristics. While the dense and empty representation provide the lowest overhead per entry in bit, the octree and voxel list provide good intermediate results for medium sparse and very sparse regions. In the following, we term the representation with the lowest memory requirement the *memory-optimal* representation.

One further aspect is that the chosen data types also provide a relatively good access performance when used correctly. The dense volume representation is clearly the best (non-empty) representation with respect to performance. It has a constant access time as shown in Table 3.1. The voxel list is used for very sparse regions and the octree for medium sparse regions. Voxel list access depends on the number of elements. Since we use a voxel list in very sparse situations, the number of elements in such a list is low, therefore the access performance is good. Octree access depends on the side-length of the bricks. Medium sparse regions contain a larger number of nodes, therefore it is beneficial to depend on the brick size and not the number of elements.

For picking a memory-optimal representation, it is not enough to simply use a threshold based on the sparsity of a brick. This is because of the shift sensitivity of the octree as mentioned in Section 3.1.2. Even though a threshold based on the sparsity could be a viable method, determining the sparsity of a volume requires to count the values. A fast algorithm of doing so, is the parallel reduction. If the sparsity of a region is known, we know the exact size of a voxel list representation. In fact, we use a single reduction to determine the final size of the voxel list. To determine the size of an octree we require only two more reductions. The advantage of this approach is, that it suffices to calculate three reductions on the whole data set, to determine the memory requirement of each representation for every brick. Knowing the memory requirement, we can simply pick the representation with the minimal size. More details on the initialization of our data structure are given in Section 5.

3.2 Hybrid Bricking

Our hybrid bricking approach splits a volume into cubic bricks of side-length m. Bricks are implemented with a power-of-two side-length mainly because this avoids special cases for the dense and octree representations and leads to a better alignment in hardware memory. Even though boundary bricks may extend outside of the volume for three sides, the impact is negligible. Boundary bricks will still pick a memory-optimal data structure. If the border bricks are only off by a few rows and the border region is densely populated with values, a dense representation will be picked. Otherwise an octree representation is more probable. Volumes with empty boundary regions do not introduce any memory overhead, since the boundary bricks are empty.

3.2.1 Memory Layout and Traversal

The hybrid bricking approach stores a tuple $(type, pointer)$ for every brick at the root level. The size of the bricks is globally defined in our implementation. The $type$ can be one of the four elemental brick types. The $pointer$ is handled in a type specific way.

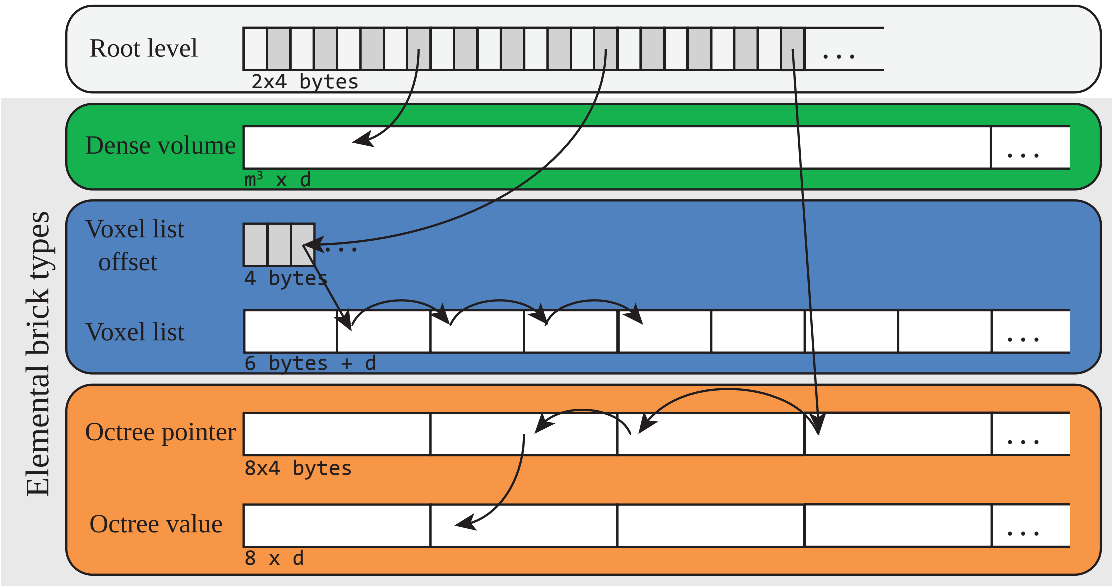


Figure 3.6: Conceptual data layout: root level nodes point to the elemental bricks. Traversal is specific to the type of a brick. Different traversals are shown for the different elemental brick types.

Figure 3.6 gives a conceptual overview of the data structure. At the root level, each brick needs to store a type and a pointer, which consists of two 4 byte integers in our case. Even though four bytes are not necessary to store a type with four possible values, we use this representation due to simplicity and due to struct alignment as explained in Section 5.3.2. A struct of arrays approach, which maintains one array for the pointers and another array for the types would be a possible alternative.

Elemental bricks depend on the size of the data type in bytes d. Dense volumes store full bricks of data values. Each voxel list stores one 4 byte pointer to its last element in the voxel list offset array. Each voxel list element stores a data value and three 2 byte values as its coordinates. These 3×2 byte coordinates are essentially a 6 byte Morton code. Internal octree nodes always store eight 4 byte pointers and leaf nodes store eight data values.

Every look-up starts with 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 total size of the dense volume array is m^3 times the total number of dense volume bricks.

- Voxel lists are also managed in a single array. To access one of multiple voxel lists it is necessary to define the start and end voxel of each list. This is done with the voxel list offset array in Figure 3.6. One look-up retrieves the start offset and another one the end offset from the offset array. Afterward, the list is traversed with binary search to retrieve the value at a specific coordinate as explained in Section 3.1.3. 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 (i.e., $log_2(m)$). The octree is traversed until a leaf node or an empty node is reached. Empty nodes are encoded with a special pointer value. The actual data values are stored in a separate region of the memory in 2^3 blocks.

3.2.2 Pointer Sizes

In our implementation, we assume that the necessary pointer value range in our data structure suffices to represent the data. We use unsigned integers of 4 byte for our pointers. This was visualized in Figure 3.6. Pointers in the root level, the voxel list offset array and the octree pointer array all use 4 byte pointers.

In Table 3.1 we show that the theoretical data size of our elemental brick types depends on the pointer size. In fact, for larger data sets or if we wish to achieve an even lower memory consumption, we have to pick the right data type for the pointers. Unsigned integer pointers can index $2^{32} = 4,294,967,296$ different values. To set this in context, directly indexing into a volume larger or equal to 1626^3 would fail due to the insufficient value range.

In practice, the most predominant case where this matters is during the upload of the data to the GPU. The octree is the only data structure we use that can theoretically have a larger number of pointers than there are values in the volume. Since our hybrid data structure prefers a representation that has a lower memory footprint than the dense data volume, it is unlikely that an octree will require more pointers than there are values (under the assumption that the number of bytes per value is smaller than the number of bytes per pointer). In this case, the data structure would prefer the dense representation over the octree. Care has to be taken if indexing into a volume with more than 2^{32} values, be it on the application side or on the GPU. In this case the index has to be of a data type with a larger value range.

In the final data structure we are in a more complicated setting. We index into different arrays and use three types of different pointers. The analysis of the required value range for each of these pointers is not straightforward. There are two cases to consider. On the one hand, there is the pointer range which is necessary to consider for storing a pointer, we call this *storage pointer range*. On the other hand, there is the pointer range required during computation and look-up, the *look-up pointer range*. We illustrate this with a simple example. Assuming our brick size is $8 \times 8 \times 8$, the number of voxels per brick is 512. If we wish to access the second brick, we can simply store 1 as a pointer (counting starts at 0). During look-up we multiply and find 512 as the starting location of our look-up in the dense volume array. In this example the storage pointer range differs from the look-up pointer range.

The maximum necessary pointer range for our pointers is shown in Table 3.2. From the root level we have to direct the look-up to either of the three elemental brick types. For the dense and the list bricks the storage pointer range is maximally the number of bricks. The more problematic storage pointer range is the one required for octrees. We store all octree nodes for all bricks in a single array, as a result the storage pointer range is the total number all internal nodes of all octrees.

The dense volume look-up requires no further indirection. The voxel list look-up needs a value range which is at most the total number of non-empty values in a volume (if all bricks are of type voxel list). The octree look-up needs to index into the whole set of leaf nodes, which are bricks of size 2^3 in our case.

	Look up from		
	Root level	Voxel list offset	Octree pointer
Dense volume	# bricks	-	-
Voxel list offset	# bricks	-	-
List element	-	# values	-
Octree pointer	all octree nodes	-	-
Octree value	-	-	# values/8 or lower

Table 3.2: Minimal storage pointer value range for the three different types of pointers in our data structure. The table shows look-ups along the rows of the table. For example, the root level can either point to the dense volume array, the voxel list offset array and the octree pointer array. In each cell, the minimal value range of the pointer is shown for the worst case.

We have shown that finding the necessary pointer range is unfortunately not so straightforward in a complex data structure. Nevertheless, it is important to keep track of the required value range for pointers, because they pose a limit to the scalability of the data structure. For a complex data structure this can be hard to determine, since pointers are used in many different cases. For each pointer two possible types for overflow, i.e., storage overflows and run-time overflows have to be considered. Efficiently choosing a minimal pointer type could decrease the memory requirement, but the cost of implementing this is suspected to be high. Although this would lend itself to just-in-time compilation and perfectly fit into our approach, we chose to set these low level parameters to a fixed value. On the one hand, the code has the potential to be more complex, on the other hand, dealing with pointer types with arbitrary bit lengths requires more complex look-ups and casting. Nevertheless, optimizing the size of pointers for hierarchical data structures could be an interesting future topic.

3.2.3 Choosing an Optimal Brick Size

Hybrid bricking provides a memory efficient representation by picking the memory-optimal representations from a set of possible data structures. It can therefore adapt to a large variety of data sets. The drawback is that it introduces type checks and it increases branching on the GPU due to the differing look-up routines for each brick type.

An additional difficulty is to pick the optimal brick size for a specific data set. Although we generally perform better than a traditional bricking with only non-empty and empty brick types, a large brick size cannot adapt as well to the data as a smaller one. If the brick size gets too small, the overhead introduced by storing brick $types$ and $pointers$ outweighs the benefit of picking more optimal elemental brick types. For our 15 tested data sets, the optimal brick size with respect to the memory consumption was either $8 \times 8 \times 8$ or $16 \times 16 \times 16$ as shown in Table 6.1.

Section 6.1 in the results Chapter deals with the benefits of using such a hybrid bricking data structure, which is a representation with a low memory consumption. In the following Chapter 4 we introduce JiTTree, a data structure that aims to improve the access performance of hybrid bricking. Chapter 5 provides an in depth explanation of our data-structure initialization-routine, which makes use of a selection of parallel algorithms on the GPU.

3.3 Modular Data Structure

The hybrid bricking data structure we have presented can be displayed as a tree shown in Figure 3.7. The layout of this tree is very simple. The root is always a bricking implementation, i.e., a grid. The leaf nodes of the tree can be of arbitrary types, in our case the four elemental brick types.

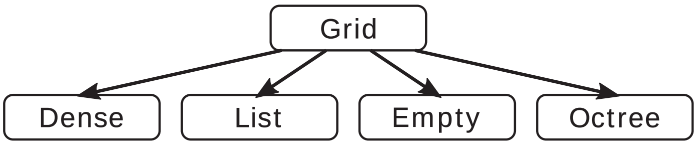


Figure 3.7: Hybrid bricking data structure, the root is a grid data structure the leaves can be of different types.

This simple bricking strategy can be replaced by a more complex, modular approach. As shown in Figure 3.8, such a modular approach would allow every node of the tree to be of a different data structure. The nodes in the example are of type grid, octree, hashing and list.

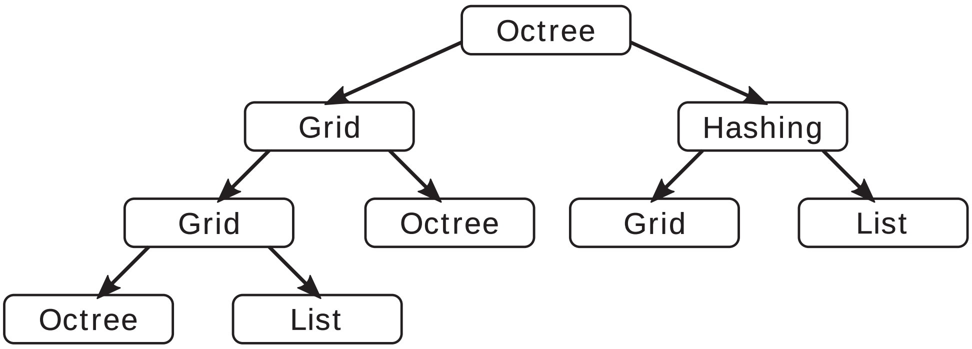


Figure 3.8: Modular adaptive data structure, each node can be of a different data type.

We hypothesize, that such a data structure can possibly even better fit a specific data set since it is more flexible than our approach. Although this approach might seem elegant, we have seen in practice that it introduces additional of problems. We made the following observations on our way to implement such a data structure:

- Internal nodes and leaves of the same type need to be implemented differently. For example, an octree as an internal node does not directly store values but pointers. An octree at the leaf levels has to store values directly, to be efficient. Therefore the number of types which have to be implemented are twice as high as initially expected.
- Leaf nodes have to be implemented as very specialized code. Otherwise the performance and the memory consumption is much higher than it needs to be. As mentioned before, an octree has to store the values directly to be memory and performance efficient.
- Any edge in the tree in Figure 3.8 is an additional indirection. Any additional indirection slows down the look-up performance of the data structure The lower the number of indirections, the better the performance of the data structure. Volume bricking is a good candidate since it requires only a single indirection. Another good candidate could be spatial hashing, since it also has a constant access performance.
- The initialization routine for such a data structure is more complex. It is harder to find and optimal representation for a specific data set, due to a larger number of possible data-structure layouts.

We conclude that, one of the top priorities for developing a data structure, should be to keep the number of indirections as low as possible. In addition, low level optimizations make a big difference. For example, let us consider once more an octree where leaves can either be pointers to other data structures or pointers to values. Let us further assume that each leaf contains a pointer to a single value. This would, for most cases (i.e., if the values and the pointers require the same number of bits) double the cost for each leaf. Clearly, it would not make sense to use such an octree, since it would probably have a higher memory requirement than a dense array.

Now let us consider an octree where each leaf is a single value. In our implementation, we store eight pointers per internal leaf. An internal node pointing to just a single leaf value would still have all eight pointers. This representation is not efficient enough.

For easier experimentation we found the data structure in Figure 3.7 to be a good trade off between generality and complexity. In the future the analysis of a more general data structure the one shown in Figure 3.8 would be interesting.

CHAPTER 4

JiTTree: Just-in-Time Compiled Tree

In the following chapter we introduce JiTTree, a just-in-time compiled tree data structure. The idea behind this data structure is to exploit information that is not known during earlier stages of compilation to optimize traversal code. Specifically, we use partial evaluation to incorporate the data set of interest into the traversal code of our tree data structure. A data set is typically loaded at the run-time of the program. We apply our additional data dependent just-in-time compilation step afterwards, hence we call our data structure just-in-time compiled tree.

Partial evaluation transforms general tree traversal code into data specific traversal code. We term this code transformation step *tree unrolling*, due to its similarity to loop unrolling. One important aspect of our data structure is *type routing*. In general a hybrid data structure consists of different 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 larger overhead during data traversal, due to the type checks. JiTTree can implicitly handle type routing without introducing additional traversal overhead. Instead of depending on the number of elemental data structures, the traversal overhead of a JiTTree depends on the homogeneity of the data set.

In the following sections we introduction JiTTree step-by-step. We start by explaining the general concept of tree unrolling with describing a binary search tree in Section 4.1. In Section 4.2 we add a type routing step which makes it possible to efficiently assign types to the nodes of our tree. Finally, in Section 4.3, we take this approach to three dimensions and explain the full JiTTree concept.

4.1 Tree Unrolling

The goal of tree unrolling is to improve the performance of the tree traversal code. It is a technique related to partial evaluation. We specialize the tree traversal code with respect to one specific data set. The data set is our static data which is fed into the partial evaluator. The look-up coordinates to our data structure remain the dynamic input data to our specialized traversal code.

To illustrate the concept of tree unrolling we consider the example of doing look-ups in a binary search tree. Figure 4.1 shows the binary tree of a sorted list of keys. Corresponding to each key a value is stored. In this case, the values have increasing indices starting from the left, but they can be arbitrary numbers. During look-up the tree is traversed from top to bottom. When the key is found the value of the element is returned.

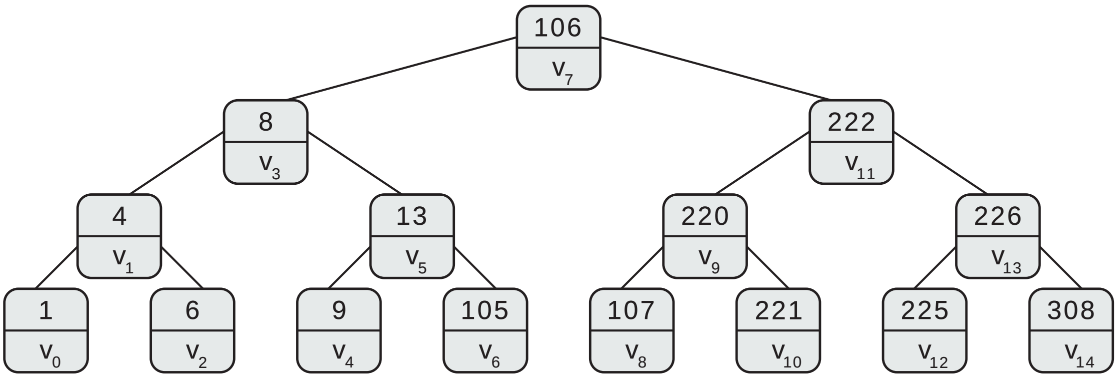


Figure 4.1: Binary search tree example, the upper value in each node is the position (or key) and the lower one is the value.

Such a binary tree can efficiently encode a sparse set of values in one-dimensional space. For instance a dense array of the 15 non-zero elements from v_0 to v_{14} at the positions 1, 4, 6, 8, 9, 13, 105, 106, 107, 220, 221, 222, 225, 226 and 308 would allocate memory that is large enough to fit at least 309 values. A sparse list representation of the array only stores the 15 values and the indices of the elements. In contrast to a list, a binary search tree reduces the overhead of retrieving a key from $O(n)$ to $O(log_2(n))$.

Listing 4.1 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 corresponding value. The search tree is a sparse representation which makes it possible to avoid storing the empty elements. 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 4.1 are the memory fetches that are introduced for the tree traversal, and the *node.value* instruction is the memory fetch that is introduced to retrieve the value. Unrolling such a tree directly incorporates the data of the tree into the code. It thereby eliminates the need for memory fetches which can reduce the traversal overhead.

```
// recursive tree traversal for a look-up at the position key
uint find(key, node)
{
  // compare key to node key
  // traverse left subtree
  if(key < node.key)
    if (node.left != NULL)
      return find(key, node.left);
    else
      return NOT_FOUND;
  // traverse right subtree
  else if(key > node.key)
    if (node.right != NULL)
      return find(key, node.right);
    else
      return NOT_FOUND;
  else return node.value; // return result
}
```

Listing 4.1: Recursive binary search tree look-up function in pseudo code.

```
// unrolled tree traversal with look-up at the position key
uint find(key)
{
  // compare key to root key
  if(key < 106) // traverse left subtree
    if(key < 8) // traverse left-left subtree
      if(key < 4)
        if (key == 1) return v0;
      else if(key > 4)
        if (key == 6) return v2;
      else return v1;
    else if(key > 8) // traverse left-right subtree
      if(key < 13)
        if (key == 9) return v4;
      else if(key > 13)
        if (key == 105) return v6;
      else return v5;
    else return v3;
  else if(key > 106) // traverse right subtree

    ...

  else return v7;
  return NOT_FOUND;
}
```

Listing 4.2: Unrolled binary search tree look-up function in pseudo code.

Listing 4.2 shows the pseudo code of the unrolled binary search tree. By explicitly compiling the data into the code the access becomes less memory-bound during tree traversal. The *find* routine does not require the node data structure as argument anymore. The memory fetches to *node.key* and *node.value* are avoided.

The representation saves memory. Even though the data is now stored in instructions (i.e., in the instruction cache), the pointers from parent to child nodes vanish. During traversal of such a data structure, no fetches into the node data structure are necessary. Caching such a node data structure is not very effective in a parallel environment. We will discuss the associated problems in Section 5.3.2. It can be more efficient to design the array of structs node data structure as a struct of arrays. In addition, the parent and child nodes may not necessarily lay close to each other in the node data structure, which can cause cache misses when traversing the tree.

Tree unrolling shifts some of the data caching problem towards the instruction cache. Even though the potential cache usage is lower, since we do not need to store pointers for example, this transformation is not for free. Assuming that the GPU does some sort of instruction caching, it might be of interest to arrange the code in a cache-efficient manner.

4.2 Type Routing

In the previous section we have used a simple binary tree to access sparse values. We now wish to incorporate additional information per node. In our hybrid bricking approach we store a type and a pointer for each brick. Similarly, in a binary tree it can be of interest to associate a type to each node. Tree unrolling provides an elegant solution to implicitly assign types to individual nodes.

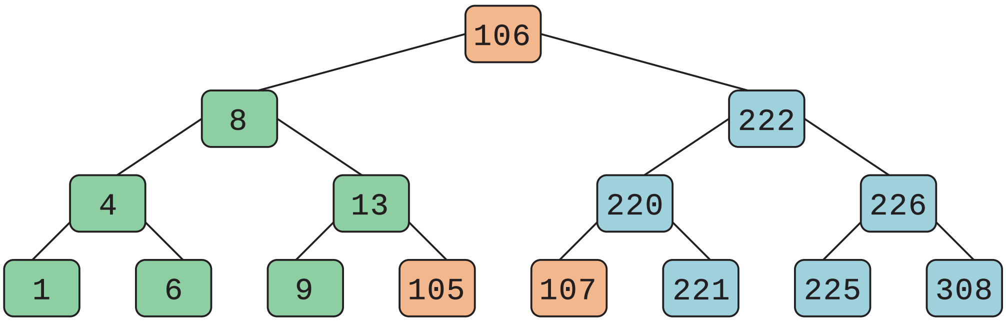


Figure 4.2: Binary search tree example. Color encodes the data type.

The types are encoded as colors in the example of Figure 4.2. If we want to execute a type specific function foo on the elements we first have to look-up their type in the corresponding array. To execute their specialized code, we call the function corresponding to that type. The pseudo code for such a type-specific method-call is shown in Listing 4.3. First the binary tree is traversed to find an index into the array of sparse elements. Then, depending on the type of the node, we execute one of the three methods.

```
// 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 4.3: foo member function evaluation pseudo code

When unrolling the tree traversal it is possible to inline the specialized foo function. Type routing and address passing can now be written explicitly. Listing 4.4 shows the result of inlining and specializing the foo member function as a result of partial evaluation.

```
// function foo inlined traversal
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)
        if (key == 1) return foo$_{green}$(v$_0$);
      else if(key > 4)
        if (key == 6) return foo$_{green}$(v$_2$);
      else return foo$_{green}$(v$_1$);
    if(key > 8) // traverse left-right subtree
      if(key < 13)
        if (key == 9) return foo$_{green}$(v$_4$);
      else if(key > 13)
        if (key == 105) return foo$_{orange}$(v$_6$);
      else return foo$_{green}$(v$_5$);
    else return foo$_{green}$(v$_3$);
  else if(key > 106) // traverse right subtree

    ...

  else return foo$_{orange}$(v$_7$);
  return NOT_FOUND;
}
```

Listing 4.4: foo function specialized and inlined

The overhead of resolving types is eliminated as specialized functions are directly called. The same concept can be applied as a routing step on top of our previously introduced hybrid bricking. 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 brick types.

In essence such a tree unrolling 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) [48]. 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.

4.3 JiTTree Approach

JiTTree is closely related to the previously introduced unrolled binary search tree for type routing. Since we are dealing with three-dimensional data, we use a kd tree instead of the binary tree. During run-time of the program, at the time when the data set has been loaded, we apply a partial evaluation scheme that specializes the code with respect to the data set. This can improve the access performance of our tree data structure. JiTTree is an extension of the hybrid volume bricking approach explained in Chapter 3. In our bricking approach, we pick an optimal brick for each sub-region of a data set.. Such an optimal brick layout can be for example a bricking with the minimal memory footprint. JiTTree is a kd tree of bricks, that is partially evaluated with respect to the data.

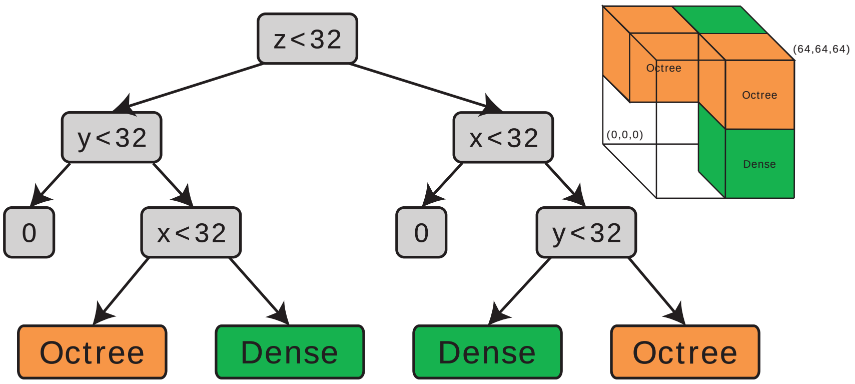


Figure 4.3: Kd tree of bricks, in this case each leaf node is a single brick. Empty bricks return zero.

Figure 4.3 shows a kd tree of bricks, where each leaf node contains either a single brick or an empty region. A kd tree makes it possible to access a sparse volume of bricks. The drawback of the approach is, that the traversal depth can be high. Even homogeneous regions, which are of the same type, will be split by the kd tree. If we use the kd tree for type routing this is not necessary. It is more efficient to generate a kd tree where each leaf node contains a sub-volume with a single brick 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 need for type branching.

Such a kd tree with leaf node bricking is shown in Figure 4.4. The volume contains three different brick types: dense, octree and voxel list bricks. The splitting planes of a kd tree are shown in gray. These planes split the volume into large homogeneous regions of the same brick type. For example, the plane at $x = 192$ splits the volume into the left sub-volume containing octree and dense bricks. The right sub-volume is further split, until each leaf of the kd tree contains only a single type. The resulting kd tree has a lower traversal depth compared to the kd tree with a single brick in each leaf node. In the leaf nodes we still have to do a bricking, but the type of the bricks is known. Therefore we get rid of the type routing conditionals, which are directly encoded in the kd tree.

4.3.1 Kd Tree Construction

Constructing a general kd tree where each leaf node contains a single brick is straightforward. It is possible to sort the bricks with respect to a coordinate and split at the median brick element. In contrast to point lists, which are often located randomly at non-integer coordinates, we regularly have to handle the case that bricks share the same x, y or z coordinate. Our algorithm sorts the bricks three times with respect to all coordinates. From each of these sorted lists, we chose the candidate where the left and the right sub-lists are closest to be equally sized.

However, the more beneficial kd tree variant, is a tree that groups bricks of the same type. It requires a different initialization routine with a splitting criterion that minimizes the *average entropy*, as introduced by Michalski et al. [32] in the context of optimizing decision trees. The entropy E of a region S with respect to the brick types is given in Equation 4.1.

$$E(S) = -\sum_{i=1}^{b} p_i \log_2 p_i \tag{4.1}$$

b is the number of different brick types, in our case four. p_i is the portion of the bricks with type i. The equation for the entropy has a negative sign, since it takes the binary logarithm of probabilities between zero and one.

Minimizing the average entropy is equivalent to maximizing *information gain*. We calculate the average entropy for each possible split as given in Equation 4.2. The volume is recursively split into two sub-regions S_j, $j \in [1, 2]$. Let us define v_j as the volume of a sub-region (i.e., a kd tree child) S_j, the *average entropy* $I(S)$ is then computed as

$$I(S) = \sum_{j=1}^{2} \frac{v_j}{v} E(S_j) \tag{4.2}$$

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 brick type.

Example: For example, consider the split at $x = 192$ in Figure 4.4. This split creates two sub-regions. The left sub-volume is weighted with $3/8$ and the right sub-volume with $5/8$.

The entropy $E(S_{left})$ of the left sub-volume is evaluated with Equation 4.2. The probabilities p_i are: $p_{octree} = 168/192 = 0.875$, $p_{empty} = 0$, $p_{list} = 0$ and $p_{dense} = 24/192 = 0.125$. As a result, the entropy evaluates to $E(S_{left}) = 0.5436$.

Let us consider the entropy of the right sub-volume $E(S_{right})$ in Figure 4.4. The volume contains $3 \times 8 \times 6 = 144$ voxel list bricks, $5 \times 3 \times 2 = 30$ dense bricks and $320 - 144 - 30 = 146$ empty bricks. The portions of brick types are: $p_{empty} = 146/320 = 0.45625$, $p_{list} = 144/320 = 0.45$, $p_{octree} = 0$ and $p_{dense} = 30/320 = 0.09375$. As a result, the entropy is $E(S_{right}) = 0.6775$. The weighted entropy of this split is $I(S) = 0.627$.

4.3.2 Kd Tree Traversal

Let us consider a volume of size $512 \times 512 \times 512$ that is split into bricks of size $64 \times 64 \times 64$ as shown in Figure 4.4. Our goal is to split the volume into homogeneous regions of the same type. The corresponding tree is shown in Figure 4.5. The resulting tree has a larger number of bricks than the previous example. Still the tree has the same depth.

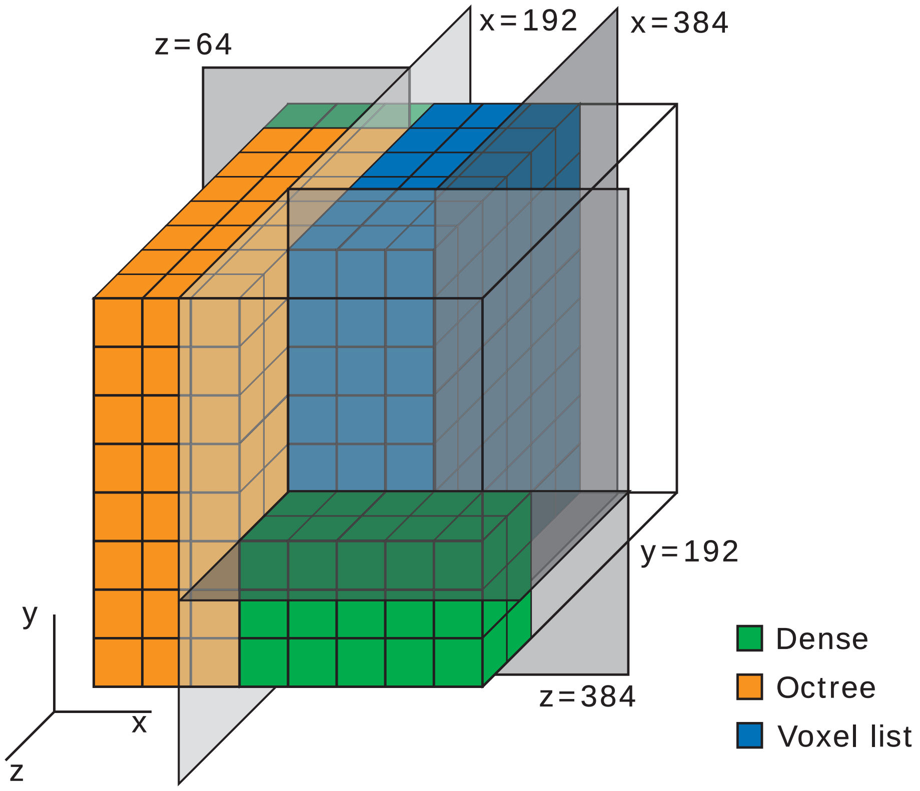


Figure 4.4: Kd tree of bricks where leaf nodes are of a single type, but multiple bricks can fall into a single kd tree leaf. Green bricks are dense, orange bricks are octrees and blue bricks are voxel lists.

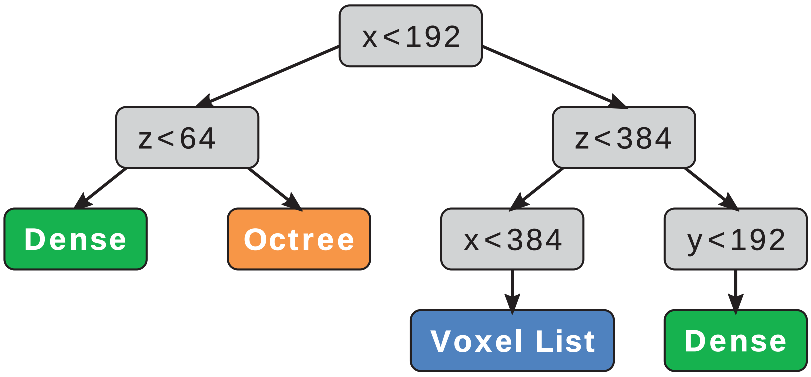


Figure 4.5: Kd tree of bricks, in this case each leaf node is a sub-volume containing multiple bricks of the same type.

```
// Node of the kd tree data structure
struct Node
{
  uint axis;       // x:0, y:1, z:2
  uint key;        // coordinate of the splitting plane
  uint type;       // INTERNAL, OCTREE, VLIST, DENSE
  Node leftChild;
  Node rightChild;
}
// Kd tree get function
int get(uint[] pos, Node node)
{
  // recursion
  if (node.type == INTERNAL) {
    if (pos[node.axis] < node.key)
      get(pos, node.leftChild);
    else
      get(pos, node.rightChild);
  }
  // type checks for all possible leaf node types
  else if (node.type == OCTREE)
    // octree bricking
    return getOctree(...);
  else if (node.type == VLIST)
    // voxel list bricking
    return getVList(...);
  else if (node.type == DENSE)
    // dense bricking
    return getDense(...);
  else return EMPTY;
}
```

Listing 4.5: Traversal code for the kd tree of bricks in a recursive fashion.

Traditional kd tree traversal for this case is shown in Listing 4.5. A node data structure is necessary, where the parent node connects with two pointers to its children. The *axis* variable defines in which direction the node splits, the *key* is the coordinate of the splitting plane. Notice that the *axis* variable is not necessary if the kd tree splits in x, y, z order successively. Splitting in arbitrary directions is more flexible and can be done for free in the unrolled kd tree. Each node stores a type which can be `INTERNAL` for all internal kd tree nodes and `VLIST`, `OCTREE` and `DENSE` for the other brick types.

After the recursive kd tree traversal, the algorithm determines the type of the leaf node. Depending on the type, a different function is called (i.e., `getOctree`, `getVList` or `getDense`).

Otherwise, the brick is empty and therefore the `EMPTY` value is returned. Type routing is done with an if/else condition, but can conceptually also be a switch-statement. OpenCL does not support switches and the performance of a switch and if-statements is comparable.

4.3.3 Kd Tree Unrolling

Listing 4.6 shows the just-in-time compiled kd tree code for the example of Figure 4.4. The `getSubVolume` functions are specialized bricking functions which can be either a sub-volume of bricks with the same type (Listing 4.7) or a single brick (Listing 4.8). In practice these functions are inlined. The pseudo code representation was chosen for better readability.

```
// jit get function with look-up position (x,y,z)
int get(int x, int y, int z)
{
  if(x < 192) {
    if (z < 64) {
      // dense bricking or single brick
      return getDenseSubVolume_0(...);
    } else {
      // octree bricking or single brick
      return getOctreeSubVolume_0(...);
    }
  } else {
    if (z < 384) {
      if (x < 384)
        // voxel list bricking or single brick
        return getVListSubVolume_0(...);
    } else {
      if (y <192)
        // dense bricking or single brick
        return getDenseSubVolume_1(...);
    }
  }
  return EMPTY;
}
```

Listing 4.6: JiTTree get function pseudo code.

Our kd tree unrolling applies five optimizations:

(i) **Traversal optimization**: No data 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 a bricking memory fetch. If the (x, y, z) position is not found as a leaf node in the kd tree, the `EMPTY` value is returned as the default in the last line of Listing 4.6.

(iii) **Implicit specialization:** No type routing between specializations of the *get* functions of different elemental data structures is required. The `getDense`, `getOctree` and `getVList` functions are inserted at the leaf nodes. No type checks are required. The type checks which were needed in Listing 4.5 are removed.

(iv) **Pointer in-baking:** For leaf nodes containing more then a single brick, the pointers are stored as a local constant array. In the case of a single brick inside a kd leaf node, it is possible to entirely eliminate the bricking. For these single-brick leaf nodes, the pointers are literals that can be directly written into the specialized function call. This eliminates one memory fetch for such leaf nodes.

(v) **Leaf node branch removal**: Kd tree branches having only a single type are reduced to a single-type bricking approach.

The specialized access functions `getDenseSubVolume`, `getOctreeSubVolume` and `getVListSubVolume` can either contain a cuboid sub-volume of bricks with a the same type or a single brick. An example of the access function for the octree is shown in Listing 4.7. For pointer in-baking, the function contains a local array of pointers $p_{[x,y,z]}$. First, the algorithm has to determine the index of the brick corresponding to position (x, y, z). The local array contains the pointer of the octree root and the general `getOctree` function is executed with the pointer as argument.

```
// specialized get function for the [0,0,64] to [192,512,448] sub-region
int getOctreeSubVolume0(int x, int y, int z)
{
  // stores the pointers for all bricks, in this case octree root nodes
  int pointer[168] = {p[0,0,64], p[64,0,64], ..., p[128,448,384]};
  // transform the (x, y, z) position into a brick index
  int i = toLinearIndex[0,0,64],[192,512,448](x, y, z);
  // starts an octree traversal at the root node defined by the pointer
  return getOctree(pointer[i], x, y, z);
}
```

Listing 4.7: Specialized bricking function

If a kd leaf contains only a single brick, the code can be further simplified as shown in Listing 4.8.

```
// specialized get function for a sub-region cointaining a single brick
int getOctreeSubVolume_i(int x, int y, int z)
{
  // starts an octree traversal starting at pointer p
  return getOctree(p, x, y, z);
}
```

Listing 4.8: Pseudo code for a single leaf brick. p in this case is a single, known integer value.

The `toLinearIndex` function in Listing 4.7 is partially evaluated by the OpenCL compiler. The boundaries of the corresponding leaf node are known at compile time. For the example of the octree, the sub-volume starts at $[0, 0, 64]$ and is of size $[192, 512, 448]$. Listing 4.9 shows the index transformation for the octree sub-volume before and Listing 4.10 after partial evaluation. At the compilation time all parameters to the function except for the variable position (x, y, z) are known. As a result, the compiler can specialize the `toLinearIndex` function at compile time and inline it.

```
// index transformation
int toLinearIndex(int3 pos, int3 start, int width, int height)
{
  return (pos.x-start.x) + (pos.y-start.y) * width + (pos.z-start.z) *
      width * height;
}
// example call for the octree
int i = toLinearIndex((int3)(x, y, z), (int3)(0, 0, 64), 192, 512);
```

Listing 4.9: Index transformation pseudo code. The code in the implementation additionally has to transform the (x, y, z) to the brick space, by dividing with the brick size.

```
// inlined code
int i = x+(y-64)*192+z*98304;
```

Listing 4.10: Inlined index transformation can be partially evaluated by the OpenCL compiler.

CHAPTER 5

Implementation

The following chapter deals with the concrete implementation of our data structure. We start by explaining the initialization routine, which uses a number of parallel algorithms to keep the processing cost low, in Section 5.1. In Section 5.2 we briefly explain the just-in-time code generation. Finally, in Section 5.3, we deal with low level OpenCL specifics, such as branch divergence and structs in OpenCL.

5.1 Hybrid Bricking Construction

After defining a brick size, the construction of the hybrid data structure is done sequentially as illustrated in Figure 5.1.

The initial step is to *analyze the sparsity* of the data set for each brick (sub-section 5.1.1). This is done by performing a parallel operation which calculates the memory requirement for all bricks with all possible types. After the memory requirement is known for all types, it is possible to chose the *optimal type* for every brick (sub-section 5.1.2). With a fixed brick layout for the entire data set, it is possible to build a kd tree of bricks (i.e., a *JiTTree*) on the application side. At this point, the memory requirement is known and the GPU memory can be allocated (sub-section 5.1.3). The sequential *initialization and upload* of each brick is the final step (sub-section 5.1.4).

During run time, when compiling a kernel that uses our data structure, we inline the data structure traversal code into the kernel code. Depending on the settings, this inlined code can either be a hybrid bricking or a JiTTree get method. In the following, we explain these initialization steps in more detail.

5.1.1 Sparsity Analysis of the Bricks

Different brick types require a different amount of memory. To allocate the memory on the GPU, the memory consumption has to be known beforehand. Furthermore, if the memory requirement

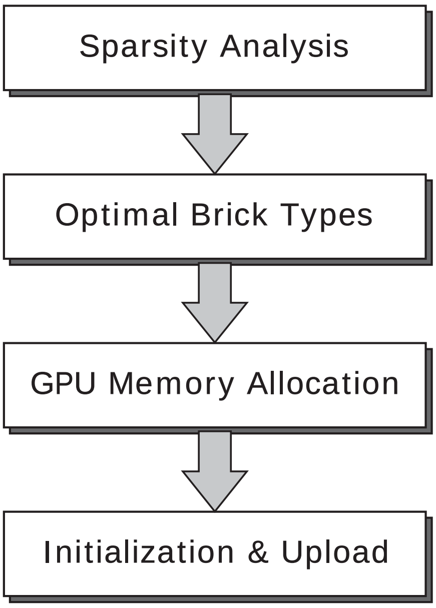


Figure 5.1: Data structure initialization steps.

for each possible brick type is known, it is possible to pick the representation with the minimal memory consumption.

The memory consumption of all elemental brick types is summarized in Table 5.1. We use these values to exactly calculate the memory requirement for each representation. The dense volume's memory footprint is constant for the brick size m and the data type d, while the octree and voxel list depend on the number of non-zero values. As a result, an octree and a voxel list representation will always require roughly the same amount of memory independent of the brick size. No matter how often a volume is split, the number of non-zero values to be stored stays the same, the only difference is the overhead of additionally storing brick type and pointers. Nevertheless, a small grid size is often beneficial in terms of memory consumption, because the elemental node types better fit the data.

Brick Type	Memory Consumption
Empty	0
Dense	$d \times m^3$
Octree	$32 \times n_{internal} + 8 \times d \times n_{leaf}$
Voxel list	$(6 + d) \times n$

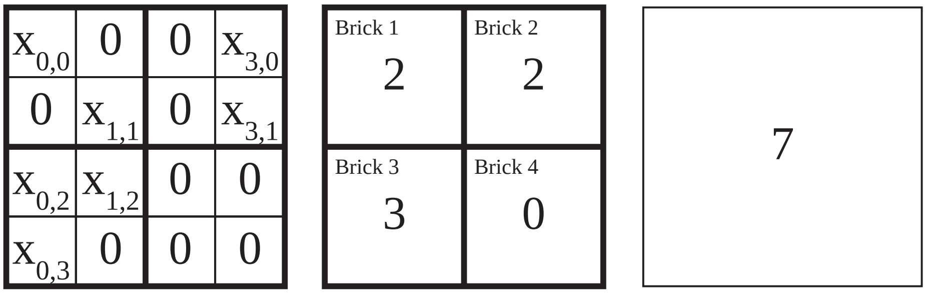
Table 5.1: Memory requirement in byte for each elemental node type. m is the brick size, n the number of non-zero values inside a brickand d the size of the data type in byte. n_{leaf} is the number of internal octree nodes at the deepest internal node level, i.e., the parent node of the leaves, $n_{internal}$ is the number of remaining internal nodes.

To determine the memory requirement for each type we require four different routines. An empty brick can be picked if the number of non-zero data voxels inside a brick is zero ($n = 0$). To determine the memory requirement of a voxel list, it is necessary to count the exact number of non-zero voxels in a brick. For octree bricks the number of internal octree nodes and leaf nodes has to be calculated. Dense bricks always require a constant amount of memory.

We use different variations of parallel reductions to determine these numbers. For large data sets we execute these reductions on sub-volumes of the data, since the data set may not entirely fit into GPU memory. In the following, we explain the process of figuring out the memory requirements for a two dimensional case. Figure 5.2a shows a two dimensional example data set. Each entry $x_{column,row}$ represents a non-zero value.

Basic Reduction

To count the number of non-zero elements for the empty or the voxel list brick type, we apply a pyramidal summation reduction on a binary representation of the data as shown in Figure 5.2b and 5.2c. If the number of elements is zero, the brick type can be defined as empty and we are done. Otherwise, the size of a voxel list can be calculated as given in Table 5.1. For the particular example, we figure out that the data set contains seven non-zero values.



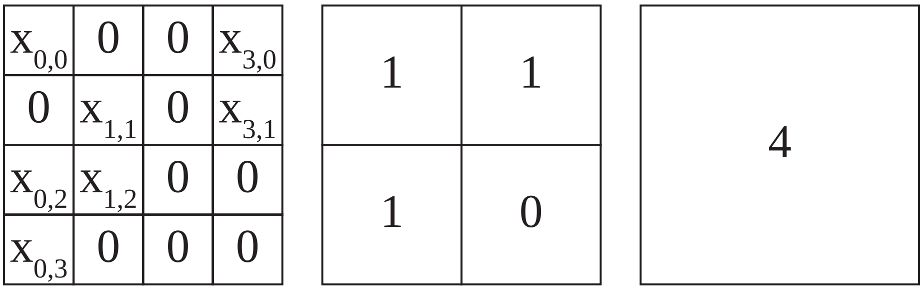
(a) Original data (b) First reduction level (c) Second reduction level

Figure 5.2: (a) An example data set. Non-zero values are represented by $x_{column,row}$. (b) and (c) show the results of a pyramidal summation reduction to find the number of non-zero values in the data set.

We calculate the number of elements for each brick in parallel over the entire data set with only a single reduction. Let us consider splitting the example data into four bricks of side-length $m = 2$. We first apply a summation reduction to the data set to determine the number of non-zero values per brick. In this case, we stop the reduction in the first level shown in Figure 5.2b. At this point, we already know that *Brick 4* is empty. The sizes of the dense and the voxel list representation of *Brick 1, 2 and 3* are known (i.e., can be calculated from Table 5.1). For the octree (quadtree) we still require to calculate the exact number of octree (quadtree) nodes.

Octree Reductions

An efficiently stored pointer octree requires two distinct arrays on the GPU, one for the leaf nodes and one for the internal octree nodes. To determine the size of the final octree, we need to count the number of these nodes. We explain the necessary octree reductions for the data set shown in Figure 5.3a. Figure 5.4 represents the pointer quadtree generated from the example data set. The quadtree has four internal nodes and exactly n leaves. Therefore it is possible to use a reduction to determine the size of the leaf node array.



(a) Original data (b) First octree internal node reduction level (c) Second octree internal node reduction level

Figure 5.3: (a) Example data set. (b) and (c) show a custom reduction to count the number of internal nodes of a pointer octree.

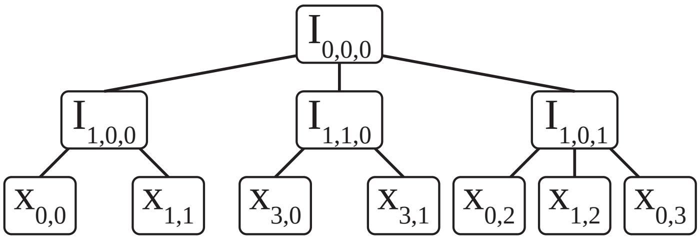


Figure 5.4: Quadtree representation of the example data set. Internal nodes $I_{level,x,y}$ and leaf values $x_{x,y}$

For determining the number of internal nodes I, we use a customized internal node reduction. A pointer octree needs one node for every non-empty sub-tree. We count the number of internal nodes with the approach shown in Figure 5.3. On the first level, we check if the leaf nodes are all zero. If they are, we start the reduction with 0, if they are not we start with 1. On every subsequent reduction level we sum the child values and add 1 for the internal node, if at least one child is non-zero. The final step of the reduction returns the number of internal nodes. In this case the number of internal nodes is 4.

A pointer octree with leaf nodes containing only one element is inefficient. Hence, our implementation uses a pointer octree with $2 \times 2 \times 2$ leaf bricks. To generate such an octree, we have to determine if a leaf brick contains at least one value. We achieve this, by applying the initial internal node reduction step once more before calculating the number of leaves and internal nodes. After the memory requirement calculation step, the size of all possible data structures are known. In the next step we pick an optimal type for every brick.

5.1.2 Choosing Optimal Brick Types

The number of possible data structure representations is exponential in the number of bricks. With four different elemental bricks there are $4^{\#bricks}$ possible data structure representations. In the following section we describe our strategies for finding a good data structure representation in a timely manner.

The two optimal solutions with respect to run time and performance can be easily found. The performance optimal solution consists of dense and empty bricks only. Dense volume look-ups are optimal, because they can be done in $O(1)$, while the octree requires $O(log(m))$ and the voxel list $O(log(n))$ additional look-ups to retrieve a specific data value. The optimal solution with respect to memory consumption can also be generated easily. For each brick, we simply pick the representation that requires the least memory.

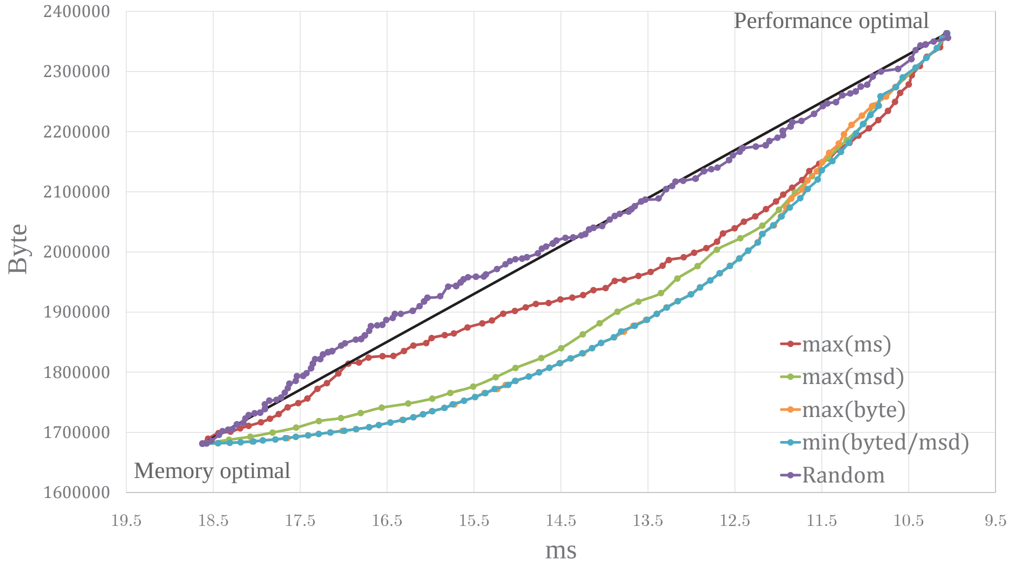


Figure 5.5: Space of possible data representations between the performance and the memory-optimal solution. Four different strategies are compared to a random sampling between both of these solutions.

Figure 5.5 shows a sampling of the space of possible data structure representations for the Hydrogen data set with a brick size of $m = 16$. Each sample point in the $(ms, byte)$-space is one specific data structure representation with a different brick assignment. The Figure gives the measured size of the data structure in bytes and its run time in ms, using a $3 \times 3 \times 3$ mean filter. The value at the bottom left is the memory-optimal solution. It consists of a mixture of dense volume, octree, voxel list and empty bricks. The value at the top right is the representation with the optimal performance, it only consists of dense volume and empty bricks. Notice that many more, less optimal data structure representations exist, which we did not plot.

For finding good in-between solutions, we start from the memory-optimal solution and subsequently replace octree and voxel list nodes by dense nodes. As a result the performance and the memory consumption increase. We test different approaches for finding optimal in-between solutions (best performance with the least memory consumption). We compared the following strategies:

- $max(ms)$: Replace the slowest non-dense brick with a dense representation first. This is achieved by executing an algorithm only on a single brick. We sequentially calculated the run time for every brick alone. The strategy finds the slowest brick and replaces it with a dense representation.

- $max(ms - ms_{dense})$: In Figure 5.5 the strategy is denoted as $max(msd)$. Replace the bricks with the largest difference between their run time and their expected dense-volume run-time first. The approach tests the dense-brick run-time and the actual run time and then calculates the difference. It exchanges the brick that introduces the largest performance gain. It is necessary to consider this difference, since dense bricks can have a varying performance. For example in the case of $3 \times 3 \times 3$ filtering, boundary cases are handled differently. The performance of a brick on the volumes boundary is generally higher than the performance inside the data structure since fewer look-ups have to be done. Look-up positions that fall outside of the volume of interest can be discarded before executing further data structure traversal code.

- $min(byte_{dense} - byte) = max(byte)$: Replace the bricks with the smallest memory increase first. Picking the globally largest value first generates the same result, since the dense representation is always the one with the highest memory requirement. The representation with the highest memory consumption is always the one that has the smallest difference to the dense representation.

- $min((byte_{dense} - byte)/(ms - ms_{dense}))$: In Figure 5.5 the strategy is written as $max(byted/msd)$. Replace the smallest slope in the $(ms, byte)$ space first. Figure 5.6 shows that in the previous two approaches we were trying to optimize by picking an optimum in either the horizontal or the vertical direction. The theoretically best solution is to minimize the slope.

The performance in ms is measured by executing a specific kernel on the data for every brick. Due to performance limitations, we approximate the run time of a brick, by executing the kernel on the brick only. We calculate the performance for every brick locally. The theoretically correct solution would require to compare the performance of the global run time for every possible brick assignment. Testing every globally possible representation requires the generation of as many data representations as there are non-dense bricks in the memory-optimal solution

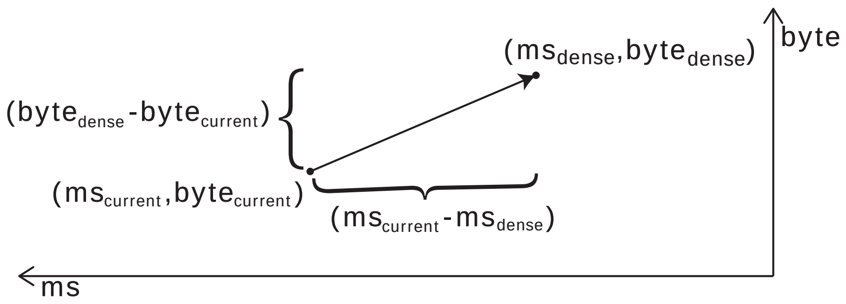

byte
$(ms_{dense}, byte_{dense})$
$(byte_{dense} - byte_{current})$
$(ms_{current}, byte_{current})$
$(ms_{current} - ms_{dense})$
ms

Figure 5.6: The lower left point represents a specific non-dense brick. The point in the upper right shows the same brick, represented as dense volume. The performance and the memory consumption of the dense representation is higher. In our second approach we maximize the performance gain only. In the third we minimize the memory overhead. In the fourth strategy, we pick the lowest slope.

for a single brick replacement. Therefore, the data structure would have to be re-initialized for every possible replacement. Our local measuring approach, only needs to generate the memory-optimal solution and test the performance of the kernel executed on each individual brick. In addition, for calculating the differences, the performance optimal solution is generated and its run times are measured.

All four strategies perform better than the random replacement in Figure 5.5. The theoretically optimal solution performs almost always best. There are three drawbacks of this approach. First, measuring performance on the GPU is noisy. Secondly, we try to find a globally optimal solution by using local measurements. Replacing the local measurements by global measurements would significantly decrease the performance of the data-structure optimization-step. Currently, this is not manageable for a large number of samples. Thirdly, picking an optimal representation by probing comes at a high cost. To find such an in between representation, we need to execute the algorithm on the data set. This only makes sense if we plan to execute the algorithm many more times in the future, since the result is already known during the measuring phase.

Our approach is able to pick one of the representations that fits the best to the desired use case. The memory-optimal or the performance-optimal solution are desirable in cases where either performance or data size are critical. In-between solutions can be picked by using the minimal slope strategy mentioned above.

5.1.3 Sequential Pre-Initialization and GPU Memory Allocation

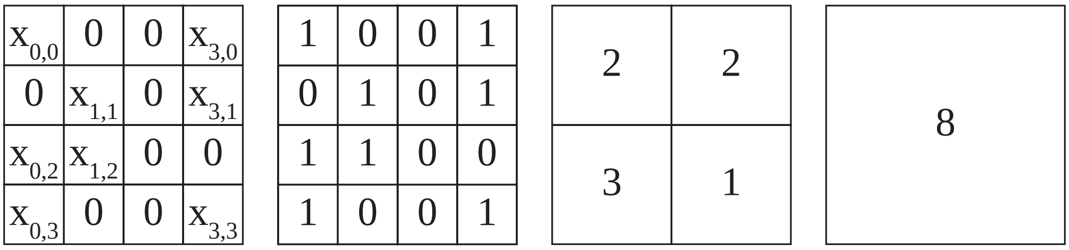
After a specific data structure layout is choosen, it is possible to allocate the GPU memory and subsequently upload the data to the GPU. Figure 3.6 shows a detailed view of the memory layout on the GPU. All bricks of a specific elemental type share global arrays for that type. To initialize each brick in its correct location inside its buffer, it is necessary to sequentially iterate over all bricks of a type and increment its start and end location. The global memory requirement is already known after step 5.1.1 and can be allocated on the GPU. The concrete size of the data structure also depends on the value range of the pointers, coordinates and data values stored inside the data structure.

5.1.4 Initialization and Upload of the Data Structure Nodes

Initialization of the nodes is done sequentially, but each node type uses a parallel algorithm for initialization. The dense volume initialization only requires to copy a sub-region of the input volume directly to the GPU. Voxel list and octree initialization require more effort as explained in the following sections.

Voxel List Construction Using Histogram Pyramids

To initialize a voxel list on the GPU, we need to count the number of non-zero elements in the input volume. Then, it is possible to initialize a list of voxels for all non-zero voxels. To achieve the necessary performance, we use *histogram pyramids* [50] to initialize the voxel lists on the GPU. A *histogram pyramid* stores the result of a reduction on the binary classified data, as introduced in section 5.1.1. Figure 5.7 shows the input data in (a), the binary classification of non-zero values in (b) and the histogram pyramid we store in (c).



$x_{0,0}$	0	0	$x_{3,0}$
0	$x_{1,1}$	0	$x_{3,1}$
$x_{0,2}$	$x_{1,2}$	0	0
$x_{0,3}$	0	0	$x_{3,3}$

(a) Input data

1	0	0	1
0	1	0	1
1	1	0	0
1	0	0	1

(b) Binary classified input

2	2
3	1

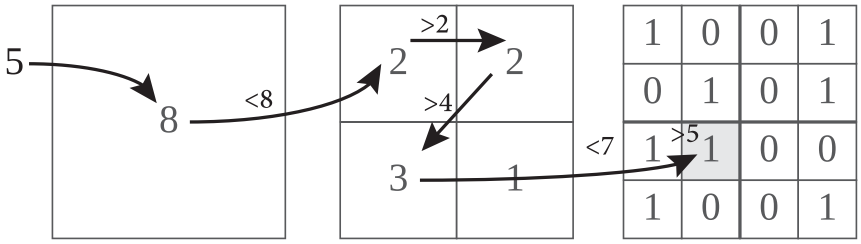
8

(c) Internal histogram pyramid levels

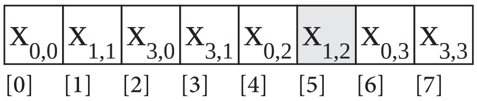
Figure 5.7: Input data and histogram pyramid. The binary classified input data is not stored in our approach.

After constructing the *histogram pyramid*, which is based on a parallel reduction, the algorithm can initialize the voxel list in parallel. A kernel is started, using n work-items. Where n is the number of elements in the list. The size of the list can be obtained by reading the root of the *histogram pyramid*. In this case the number of elements is eight. For each element we traverse the histogram pyramid as shown in Figure 5.8 to find the data entry in the input data that corresponds to a linear index. During traversal, we keep track of the boundaries corresponding to the current traversal cell. The histogram traversal for an index i starts at the root node. The traversal iterates over all elements in the current level in Morton order [33] and sums the entries until the next entry would exceed the index i. If the index i is exceeded, the traversal moves one level down the histogram pyramid hierarchy and repeats iterating in Morton order.

In the example in Figure 5.8, the index 5 is observed. 5 is smaller than 8, therefore the traversals moves down the hierarchy. In the next step, the summation of 2, 2 and 3 is stopped at



(a) Histogram pyramid traversal for the index 5. In the first step the boundary interval corresponding to the traversal is $[0, 0]$ to $[4, 4]$, in the second step $[0, 2]$ to $[2, 4]$. Finally the traversal stops at $[1, 2]$.



$X_{0,0}$	$X_{1,1}$	$X_{3,0}$	$X_{3,1}$	$X_{0,2}$	$X_{1,2}$	$X_{0,3}$	$X_{3,3}$
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

(b) Final voxel list

Figure 5.8: Voxel list construction for the list element at index 5. The histogram pyramid is traversed to find the coordinate and the value of the list element in the input volume. Traversal can be done in $log(m)$ steps, where m is the side-length of the input data.

7, since the value just exceeded the index 5. In the leaf level, the traversal stops when the index 5 just exceeded the summation of $2+2+1+1 = 6$. During the iteration we store the coordinate interval of the traversal and finally stop at location $[1, 2]$ at index 5. The same traversal is done for every other index of the list in parallel.

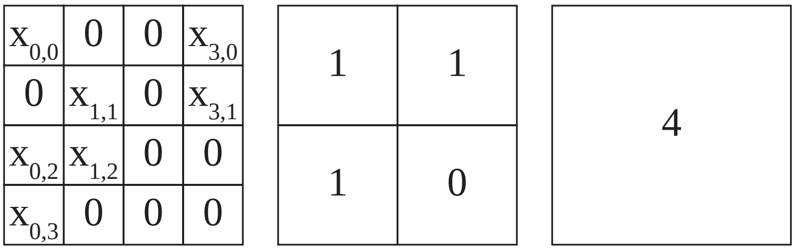
The algorithm sorts the elements of the list in Morton order (z-order). We exploit the sorting of the list elements in the look-up phase to perform binary search to increase the look-up time into our voxel list implementation. Instead of the coordinate in (x, y, z) coordinates we store the Morton code of the voxel. The code to transform a spatial location to Morton code is given in the Appendix A.

Octree Construction Using Histogram Pyramids

For initializing octrees, we also use histogram pyramids as explained in the work of Ziegler [49]. The idea of the approach is to split a pointer octree into two lists: the internal node list and the leaf node list. Both of these lists store a histogram pyramid during construction. The construction can be done in parallel for both lists.

In the following we explain the octree construction for a simple two dimensional example. First both histogram pyramids are generated as shown in Figure 5.9. The histogram pyramids use the reductions explained in Section 5.1.1 and temporarily store the histogram pyramids on the GPU.

The second step of the construction phase is executed in parallel for all of the final $n_{internal}+n_{leaf}$ octree nodes. For the example case these are $4 + 7 = 11$ nodes. We now distinguish between two different cases: internal node traversal and leaf traversal.



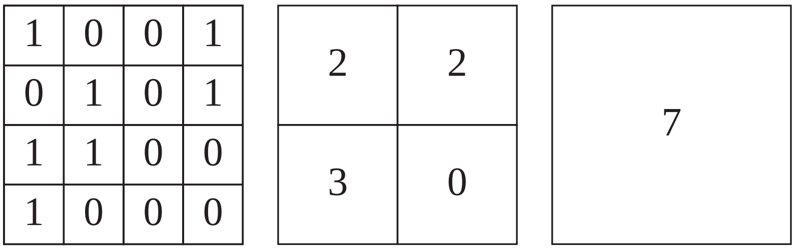
$x_{0,0}$	0	0	$x_{3,0}$
0	$x_{1,1}$	0	$x_{3,1}$
$x_{0,2}$	$x_{1,2}$	0	0
$x_{0,3}$	0	0	0

(a) Input data

1	1
1	0

4

(b) Internal node histogram pyramid



1	0	0	1
0	1	0	1
1	1	0	0
1	0	0	0

(c) Binary classified input data

2	2
3	0

7

(d) Leaf node histogram pyramid

Figure 5.9: Input data and octree histogram pyramids. The binary classified input data is not stored in our approach.

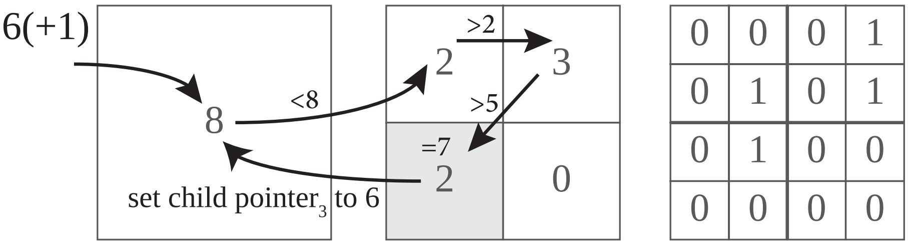
Internal Node Traversal

Internal nodes require to store all pointers to their non-zero children. We start by setting all pointers to -1, which is the not-set flag. To write the pointers we traverse the histogram pyramid down to the children. The children write their index into their parent's pointer. The internal node traversal can initialize all but the highest level of internal nodes. The highest level of internal nodes point to the leaves, therefore this level can only be written during leaf node traversal.

Internal node traversal starts at the root node. In contrast to the algorithm used in the voxel list construction in Section 5.1.4, the traversal can also terminate on internal levels of the histogram pyramid. In voxel list construction the traversal always stopped at the leaf level.

We achieve this, by introducing checks for equality in the histogram pyramid traversal. If an index is exactly matched during histogram pyramid traversal, we found an internal node. After finding an internal node I, we need to find its parent. The parent node is the index one level above the current node I in the histogram pyramid.

An example of finding the index 6 is shown in Figure 5.10. In this case, the index is increased by one. The value is checked with the entries of the histogram pyramid until we find the node corresponding to 7. The parent node in this case is at the index $8 - 1 = 7$.



(a) Histogram pyramid internal leaf traversal for the index 6

$I^{2}_{1,1}$	$I^{1}_{0,0}$	$I^{2}_{3,0}$	$I^{2}_{3,1}$	$I^{1}_{1,0}$	$I^{2}_{1,2}$	$I^{1}_{1,0}$	$I^{0}_{0,0}$
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

(b) Temporary internal node list. The root is the last element.

Figure 5.10: An example showing that the internal leaf traversal can stop in internal levels of the histogram pyramid.

Leaf Node Traversal

During the leaf node traversal, the algorithm needs to write the values into the octree leaf nodes and set the pointers from the lowest level internal nodes to the leaf nodes. Once again the algorithm writes the pointers in reverse order. It iterates to the leaf nodes and then writes the index of the current leaf node into its parent.

Traversing to the leaf nodes uses the same algorithm as in the voxel list generation. Finding a leaf node's parent requires to traverse the internal node histogram pyramid at the same time as the leaf node histogram pyramid (called co-traversal [49]). This is done by executing the exact same traversal steps in both histogram pyramids. In addition to the leaf node index, it is possible to retrieve the parent node index from the internal node histogram pyramid. Subsequently, the pointer and the value for each leaf node are written.

Figure 5.11 shows the octree constructed from the example in Figure 5.9. The arrows denote the pointer writes during the construction phase. The generalization from quad- to octrees requires only an adaptation of the Morton order computations.

5.2 JiTTree Construction

The general concept of the JiTTree was explained in Chapter 4. In this section we give a brief overview on the generation routine of the JiTTree representation. As shown in Figure 5.1, the construction of the JiTTree can be done after the optimal brick type representation was chosen.

We start by initializing a volumetric array of types. This array is then fed into an algorithm that minimizes the entropy with respect to the brick types. The algorithm tries all possible split planes through the volume and splits when the resulting two sub-volumes provide the solution

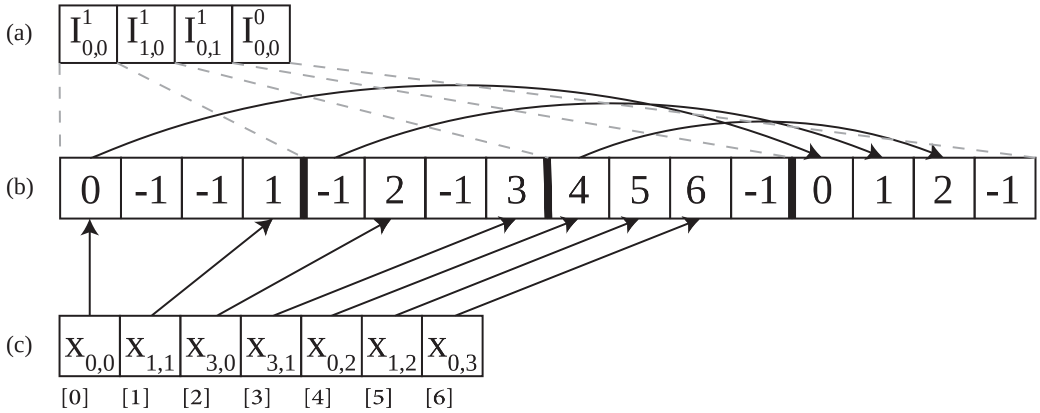


Figure 5.11: Pointer quadtree implementation.

with the lowest entropy. The splitting is recursively applied until each leaf contains only a single type.

After this tree representation is constructed on the CPU, a GPU code generator is started. We recursively traverse the tree representation and build OpenCL code. For each node of the kd tree the code generator builds either an *if/else*-statement for internal nodes or a single-type bricking for leaf nodes. The *if/else*-statement compares the input coordinates with a split plane to identify the traversal direction. The resulting code is very similar to the pseudo code presented in Listing 4.6 for the unrolled tree. At the leaf nodes of the kd tree, a local array is generated. It holds the pointers to do a volume bricking with a single brick type.

5.2.1 Low Level Optimizations

Low level code modifications result in better code readability and sometimes in performance gains.

Single Branch Optimization

Simplifications of *if/else*-statements in the tree traversal are possible as shown in Listing 5.1. Where no *else*-statement is necessary it is possible to omit it. If the *if*-side of the *if/else*-statement is unnecessary, we reverse the condition and also omit the *else*-statement. However measuring the performance did not show any benefits. We suspect that the OpenCL compiler removes unnecessary branches which results in the same binary representation. However, for debug purposes we apply the simplification since it makes the code shorter and easier to read.

```
// generated if/else-statement
if (y < 64) {
  if(x < 64)
  {
    return EMPTY;
  }
  else
  {
    return getOctree(...);
  }
}
else
{
  return EMPTY;
}
```

Listing 5.1: *If/else*-statement with an empty return value in the *if*-branch.

```
// simplified statement
if (y < 64) {
  if(x >= 64) // reversed condition
  {
    return getOctree(...);
  }
}
return EMPTY;
```

Listing 5.2: Simplification by switching the condition and moving the `EMPTY` return to the end of the code.

One limiting factor of our approach is the OpenCL compilation time, as will be shown in the results. As the code for larger trees can reach thousands of lines of code, the compilation can take on the order of minutes. As a result, we were only able to unroll kd trees up to several thousands of leaf nodes on the Nvidia OpenCL compiler. Intel's OpenCL compiler consequently showed worse results in terms of compile time.

The just-in-time compilation approach also allowed us to compile constants into the code such as the brick size or the volume size. This potentially allows the compiler to simplify statements through partial evaluation.

The approach also works well for the case that a volume is represented only by a single brick. In this case, the just-in-time compilation completely gets rid of any bricking code and just performs a single look-up into the type of the elemental data structure.

Octree Traversal Optimization

When traversing through an octree, we have to decide into which of the eight octants a value falls. This can be done by eight different if-conditions, as shown in Listing 5.3.

```
if (x < midX  && y < midY  && z < midZ)     node = node.child[0];
if (x < midX  && y < midY  && z >= midZ)    node = node.child[1];
if (x < midX  && y >= midY && z < midZ)     node = node.child[2];
if (x < midX  && y >= midY && z >= midZ)  node = node.child[3];
if (x >= midX && y < midY  && z < midZ)     node = node.child[4];
if (x >= midX && y < midY  && z >= midZ)  node = node.child[5];
if (x >= midX && y >= midY && z < midZ)   node = node.child[6];
if (x >= midX && y >= midY && z >= midZ)  node = node.child[7];
```

Listing 5.3: Octree traversal, checking into which octant a look-up coordinate falls.

However, there are different low level options to check into which of the eight octants the position falls. One option is to use bit shift operators as done by Knoll [24]. The code is shown in Listing 5.4, while this code has fewer lines of code and might suggest that it is faster, we did not measure a significant performance gain.

```
uint lastElementPos = (x >= midX) << 2 | (y >= midY) << 1 | (z >=
    midZ);
node = node.child[lastElementPos];
```

Listing 5.4: Octree traversal via bit shift

By trying multiple different settings we found that moving one conjunct out of the eight different if-conditions provides the best performance in our case. The optimized code is shown in Listing 5.5.

```
if (x < midX) {
  if (y < midY && z < midZ)     node = node.child[0];
  if (y < midY && z >= midZ)    node = node.child[1];
  if (y >= midY && z < midZ)    node = node.child[2];
  if (y >= midY && z >= midZ)   node = node.child[3];
} else {
  if (y < midY && z < midZ)     node = node.child[4];
  if (y < midY && z >= midZ)    node = node.child[5];
  if (y >= midY && z < midZ)    node = node.child[6];
  if (y >= midY && z >= midZ)   node = node.child[7];
}
```

Listing 5.5: Highest performing octree boundary checks

One of the benefits of our hybrid bricking approach is that it makes it possible to optimize each of the elemental data structures independently.

5.3 OpenCL Programming

The following section will give an overview of a few essential OpenCL programming concepts which we have used in our implementation. This section focuses on details, rather then on a lengthy introduction to OpenCL. For a more in depth introduction to OpenCL we would like to refer to the many presentations given on the topic, for example by Woolley [47] and to the OpenCL specification [15].

Calling a kernel in OpenCL typically starts a specified number of concurrent *work items*. Work-items are conceptually the concurrent threads executed on the hardware (therefore they are also called threads in Cuda). The code defined in the kernel function is executed for every work-item in parallel. Each of these work-items has a unique global id. Work-items are grouped into *work-groups*. Each work-item also has a local id inside its work-group. Work-items in work-groups share local memory, which is a faster but smaller region of memory.

These work-items are mapped in the following way to the Nvidia hardware. The actual hardware cores are called streaming multiprocessors. Each work-group is assigned to one of the streaming multiprocessors. The work-items inside a work-group are grouped together in groups of size 32 - these groups are called *warps*.

5.3.1 Branching

Work-items in a warp execute the same code concurrently in lockstep. When reaching a branching control flow, the executed code can differ. In this case, different branches are executed sequentially if the work items actually branch out during run time (called branch divergence). Listing 5.6 shows an example branching code. There are two branches, which call potentially expensive other functions. The path of the branching is determined by the global id.

```
kernel void branching(global uint * result)
{
  uint globalID = get_global_id(0);

  if (globalID < 4)
  {
    result[globalID] = expensiveFunctionCallA();
  }
  else
  {
    result[globalID] = expensiveFunctionCallB();
  }

}
```

Listing 5.6: Branching example code.

As a result, the code of the else branch has to wait until the if branch was executed and vice versa. The resulting paths of execution are shown in Figure 5.12. In some cases, branch divergence can be avoided. This can be done by having the same branches execute in every warp. It is much more efficient to have different warps execute different branches.

If the amount of code in a branch is small, the compiler uses a technique called predication as mentioned in the Cuda Handbook [46]. Typical branching instructions cause a jump in machine code. Predication is a mechanism to mark whether each line of machine code has to be evaluated or not. Predicated code is executed in any case, but the result is not written back if the code lies in a branch that is not reached. As a result, branching is cheaper if the code in branches is short.

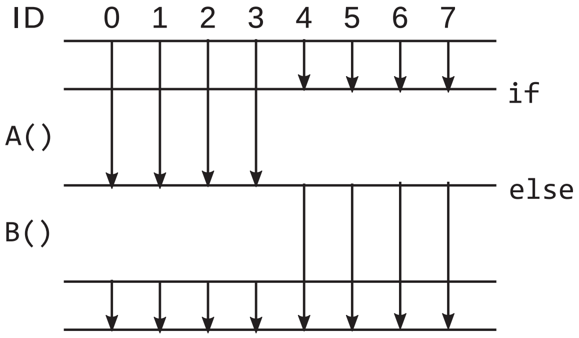


Figure 5.12: Temporal diagram for the code given in Listing 5.6 showing branch divergence. The eight work-items of the same warp have to wait for each other.

5.3.2 Structs in OpenCL

One of the goals of our implementation was to keep the content of our data structure in a single buffer. This makes it easier to use the data structure from other kernels. It was achieved by organizing the data structure as a struct. In the following, we explain a few pitfalls when using structs in OpenCL and how to avoid them. In addition, the efficient usage of structs is necessary to avoid wasting GPU memory.

Arrays of Structs vs Structs of Arrays in Global Memory

In our implementation, the data structure itself has to reside in global memory due to its size. Instructions that access global memory have a high latency. With certain compute capabilities [1] global memory access uses L1 caching. In OpenCL the size of these cache lines can be checked with the device info query of `CL_DEVICE_GLOBAL_MEM_CACHE_TYPE`. In most hardware, global memory cache lines are 128 bytes wide. Load operations from global memory by different threads in the same warp are coalesced. So, to achieve effective global memory access, work-items in a warp should read the global memory that falls within a cache line.

Structs in OpenCL work similarly as in C since OpenCL is based on the C99 standard. Correctly dealing with structs in a parallel environment can provide a performance benefit as reported in the presentation by Luitjens [31]. One example is the difference between arrays of structs and structs of arrays.

Array of Structs

Let us consider for example our voxel list implementation for the data type $ushort$ shown in Listing 5.7. This array of structs is stored linearly in memory as shown in Figure 5.13.

---

[1] With compute capability 2.0, 3.5, 3.7 and 5.2 `http://www.acceleware.com/blog/opt-in-L1-caching-global-loads`, Dec. 2015.

```
struct Voxel
{
  ushort x; // 2 byte
  ushort y; // 2 byte
  ushort z; // 2 byte
  ushort v; // 2 byte
}
// array of structs
struct Voxel voxelArray[SIZE];
```

Listing 5.7: Array of structs for a voxel list.

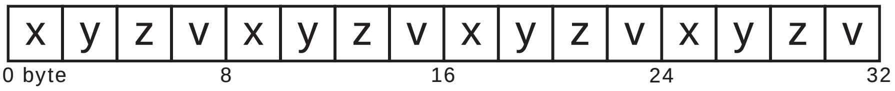


Figure 5.13: Array of structs as stored in memory.

Let us examine an algorithm executing on the voxel list. It accesses the data at the x coordinate of a voxel shown in Listing 5.8 and stores it to a different array. Typically such a kernel is executed in work-items where each group of 32 work-items form a warp. Starting from global index 0, a cache line of 128 bytes is read from global memory. The cache line can hold 16 voxels, therefore in the entire warp it is necessary to load two cache lines.

```
kernel void globalCopyX(global struct * voxelArray, global ushort *
    resultX)
{
  uint id = get_global_id(0);
  resultX[id] = voxelArray[voxelArray[id].x];
}
```

Listing 5.8: Array of structs global access.

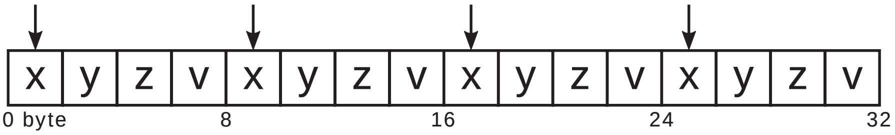


Figure 5.14: Access to an array of structs.

Figure 5.14 shows how the memory is accessed in an array of structs. Six of the eight bytes per voxel from global memory are unnecessarily cached.

Struct of Arrays

The alternative to an array of structs is a struct of arrays, shown in Listing 5.9.

```
// struct of arrays
struct VoxelList
{
  ushort x[SIZE]; // 2 byte
  ushort y[SIZE]; // 2 byte
  ushort z[SIZE]; // 2 byte
  ushort v[SIZE]; // 2 byte
}
```

Listing 5.9: Struct of arrays.

Accessing the array with the same algorithm is more efficient. Figure 5.15 shows the access pattern in this case.

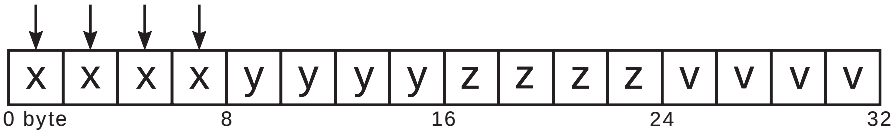


Figure 5.15: Access to a struct of arrays.

In the example, the struct of arrays would be able to load all 32 x-coordinates in a warp with a single cache line of 128 bytes. In conclusion, the struct of arrays approach is especially useful if the struct of interest is large, such as in a voxel list where the coordinates are stored with 4 or 8 byte precision.

In our implementation, we avoided using arrays of structs if the containing structs are too large. In practice, the voxel list example is not so bad since the struct itself only requires 8 bytes. The data structure itself is organized as a struct of arrays, while some arrays can contain small structs. The maximum size of a struct inside an array in our implementation is 8 bytes.

Data Structure Alignment & Struct Padding

Data types in OpenCL can not start at an arbitrary location in memory. Each data type has to be aligned to a multiple of its size in bytes. Structs are therefore generally aligned with respect to its largest member.

Consider once more the array of structs of a voxel list with 2 bytes per value as given in Listing 5.7. This struct is efficient since it is well aligned in memory. In contrast, a similar Voxel with 4 byte values as shown in Listing 5.10 will result in struct padding.

```
struct Voxel
{
  ushort x; // 2 byte
  ushort y; // 2 byte
  ushort z; // 2 byte
  uint v;   // 4 byte
}
```

Listing 5.10: Voxel with 4 byte value

The largest element of the struct is 4 byte, therefore its representation in memory contains empty regions to correctly align the struct and its content. The resulting memory layout is shown in Figure 5.16. The x, y and z coordinates are stored contiguously like in the previous example. After each z coordinate there are 2 bytes of unassigned memory. The value v starts with an offset of 2 bytes to be aligned as a multiple of four. In this data structure layout, $1/6$ of the memory is wasted.

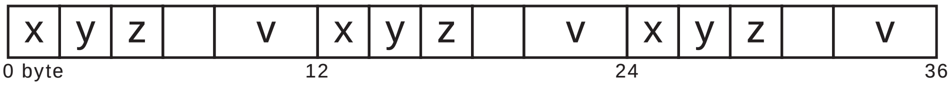


Figure 5.16: Array of voxels with 4 byte values.

To avoid such an alignment there are two options. Either the programmer packs the structs in a good way, or it is possible to use a struct of arrays approach. Struct alignment with the default compiler settings in OpenCL is not necessarily the same as in C++. OpenCL additionally supports alignment options, for more details we would like to refer to the OpenCL specification [15].

One of the pitfalls in our case, was that an array storing 2 byte values, which is followed by an array of 4 byte values, has to to contain an even number of entries. This is because the array of 4 byte size will always start at a position that is 4-byte aligned. An array containing 2 byte values with an odd number of elements does not stop at a 4-byte aligned position in memory. Problems can occur when uploading such structs from the CPU to OpenCL, since the alignment can differ.

CHAPTER 6

Results

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

We compare 15 test data sets. The kingsnake data set is courtesy of DigiMorph.org and Timothy Rowe. The vessels data set was provided by John Keyser at Texas A&M University. The stag beetle data set was created by Georg Glaeser, Vienna University of Applied Arts, Austria, Johannes Kastner, Wels College of Engineering, Austria, and Meister Eduard Gröller, TU Wien, Austria. The christmas tree was provided by TU Wien, Austria [22].

6.1 Memory Consumption

We compare the memory requirement for 15 data sets shown in Table 6.1. Larger renderings can be seen in Figure 6.7. 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 6.1 with the specific threshold indicated with th. 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 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.

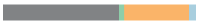
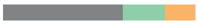
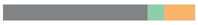
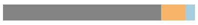
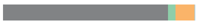
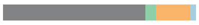
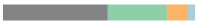
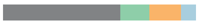
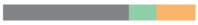
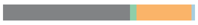
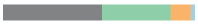
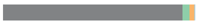
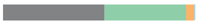
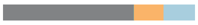
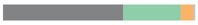
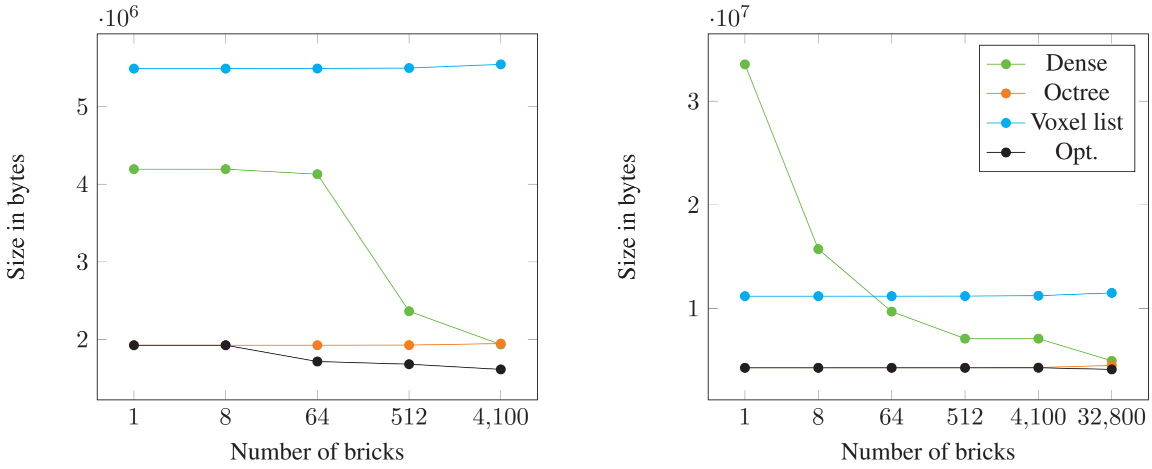
	Volume size	m	Non-zero values (%)	Size (% of dense)	Type of bricks (% of volume)
Bonsai th=0.08 	512×512×189	16	8.17	14.85	
Bunny th=0.08 	512×512×361	8	28.76	34.28	
Carp th=0.08 	256×256×512	16	16.27	19.82	
Christmastree th=0.03 	512×499×512	16	1.47	4.02	
Engine th=0.08 	256×256×256	8	8.34	12.32	
Frog th=0.08 	256×256×44	8	11.22	18.23	
Hydrogen 	128×128×128	8	32.72	38.58	
Monkey-CT 	256×256×134	16	17.11	29.68	
Piggy Bank th=0.03 	512×512×134	8	23.72	30.53	
Porsche th=0.03 	559×1023×347	16	40.31	52.46	
Schaedel th=0.03 	512×512×333	8	15.64	23.14	
Stagbeetle 	832×832×494	16	4.06	4.75	
Kingsnake th=0.15 	1024×1024×795	16	43.68	44.85	
Vessels th=0.15 	1024^3	16	1.67	4.22	
Connectomics 	1024^3	16	29.68	34.04	

Table 6.1: Memory consumption results. The brick types are empty (gray), dense (green), octree (orange) and voxel list (blue)

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 6.1 shows the memory requirements for two data sets for different brick sizes and different elemental data. In Figure 6.1a 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 6.1.

In Figure 6.1b 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

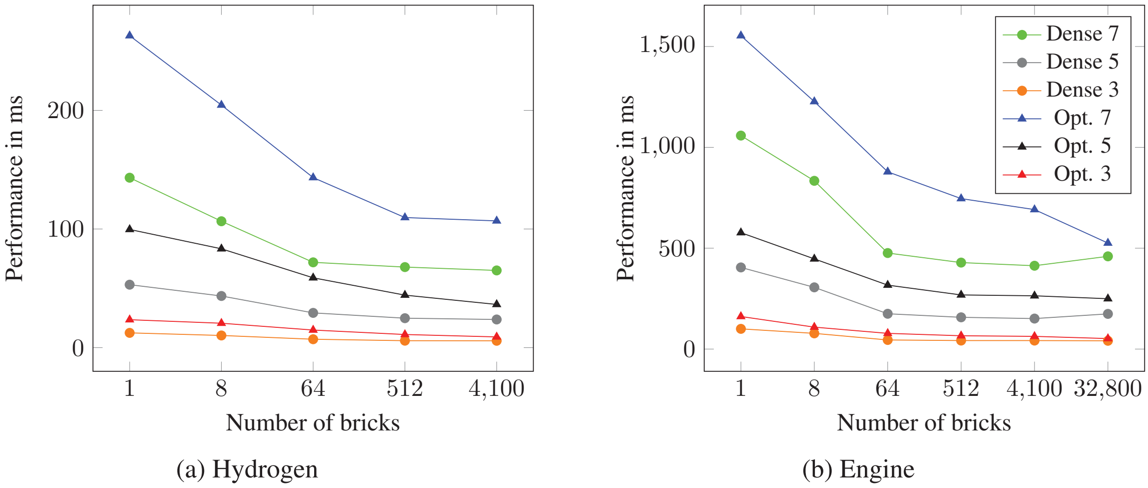
Figure 6.1: Memory requirement of dense, octree, voxel list and memory-optimal bricking. The memory-optimal bricking outperforms any other bricking for most cases.

6.2 Performance

The performance was tested for a mean filter with different kernel sizes as well as for volume ray casting. Figure 6.2 shows the performance for the mean filter. As expected the dense brick 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.

Figure 6.3 compares the performance of the direct volume bricking approach of the memory-optimal solution (Chapter 3) with the JIT compilation approach of the memory-optimal solution (Section 4.3). 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 6.3, a factor of 1.79, is given for the stag beetle data set at a brick size of 32^3. Only for one case (kingsnake data set with brick size 128^3) we observe a decrease in performance by a factor of 0.96.

Figure 6.4 shows the performance of a $3 \times 3 \times 3$ 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 6.4, since their performance is much worse for an entire data set. Nevertheless, voxel lists perform



(a) Hydrogen (b) Engine

Figure 6.2: Performance of a mean filter with kernel sizes 7^3, 5^3 and 3^3 in ms.

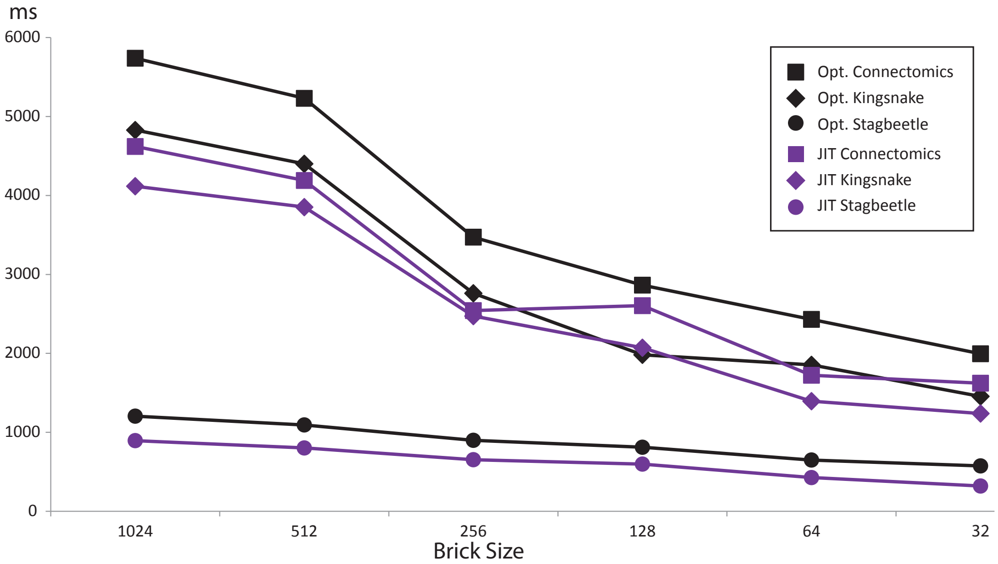


Figure 6.3: Performance comparison for a mean filter with $3 \times 3 \times 3$ kernel size. JIT compilation (purple) compared to direct volume bricking (black) for three different data sets.

optimal in terms of memory for very sparse nodes, which makes them attractive for a hybrid data structure.

Table 6.2 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 6.2 also shows the source-to-source kd tree construction time and the OpenCL compile time.

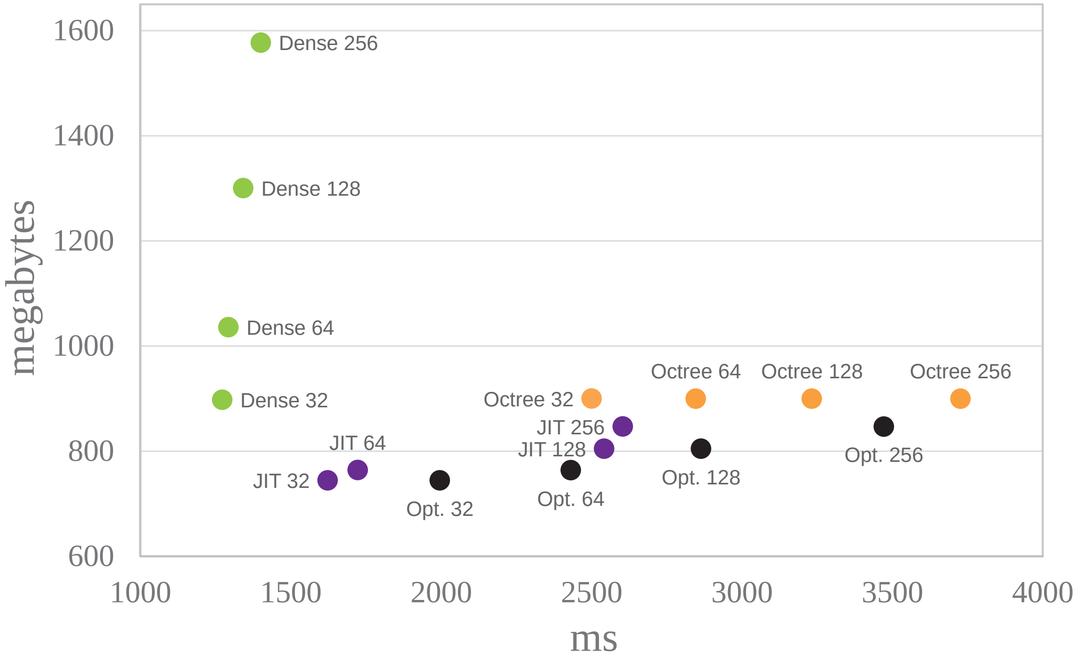


Figure 6.4: $3 \times 3 \times 3$ mean filtering of the Connectomics data set. Different brick sizes are tested for dense, octree and memory optimized bricking approaches.

# bricks	# leaves	max. branch depth	kd tree construction (ms)	OpenCL compilation (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 6.2: JIT generated code properties and build times for the Connectomics data set.

In contrast to filtering, the JIT approach does not perform well for ray casting as shown in Figure 6.5. 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 a much higher branch divergence than the stencil access pattern for the mean filter calculation.

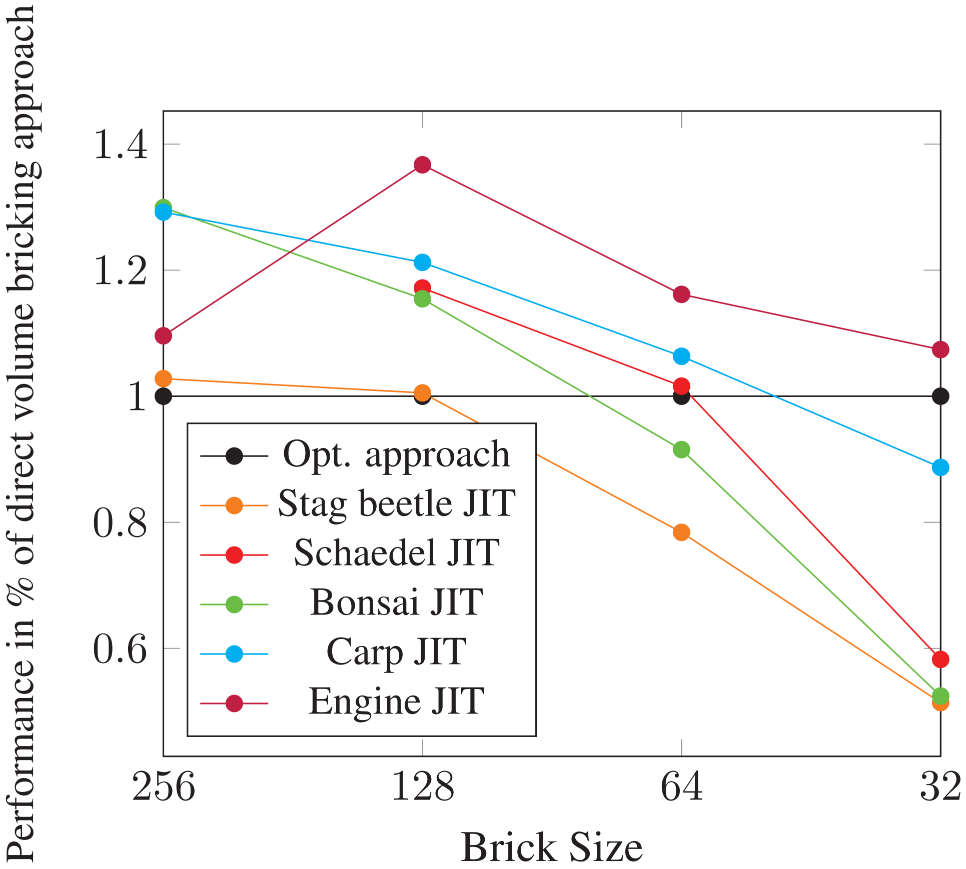


Figure 6.5: Performance of ray casting with the JIT approach compared to the memory-optimal bricking (black line).

6.3 Varying Brick Size

Although we always use a fixed brick size for one instance of our data structure, we are able to generate our data structure for different brick sizes. Changing the brick size has multiple implications. A small brick size makes it possible to use more dense volume and empty brick types, which are a more beneficial representation with respect to the memory requirement of the data structure. Figure 6.6 shows bricking with different brick sizes for the vessel data set. Voxels are colored with respect to their brick type.

The largest brick size is $32 \times 32 \times 32$. In this case, there is a large number of octrees and voxel lists. As the brick size is decreased, the number of empty, voxel list and dense bricks increases. In Figure 6.6c thick vessels are mostly represented by dense bricks.

Table 6.3 gives an overview of the corresponding brick distribution and memory size for each of the three brick sizes. Looking at the brick distribution, the brick size of 8 has the highest portion of empty bricks and also the relatively highest portion of dense bricks. Nevertheless, the representation with a brick size of 16 provides the best result with respect to the memory requirement. This is because bricking itself is not for free. At a small brick size the storage cost of types and pointers exceeds the benefit of a better data representation for each individual brick.

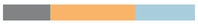
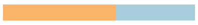
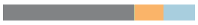
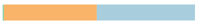
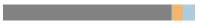
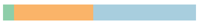
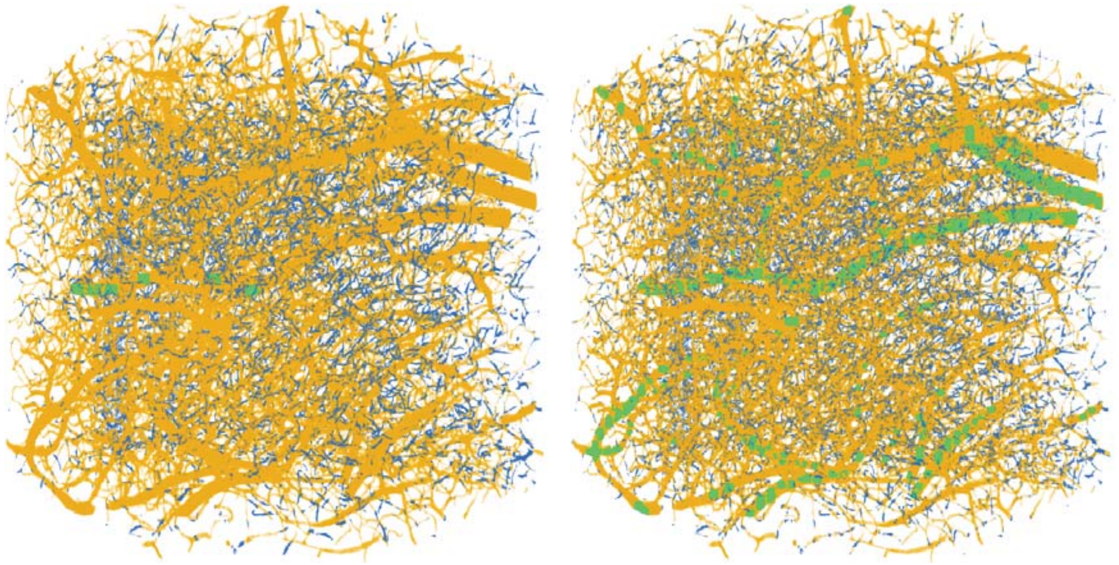
Brick size	Memory requirement	Type of bricks	Type of bricks w.o. empty	Number of bricks
$32 \times 32 \times 32$	$93,1$ MB			32768
$16 \times 16 \times 16$	$90,6$ MB			262144
$8 \times 8 \times 8$	$98,2$ MB			2097152

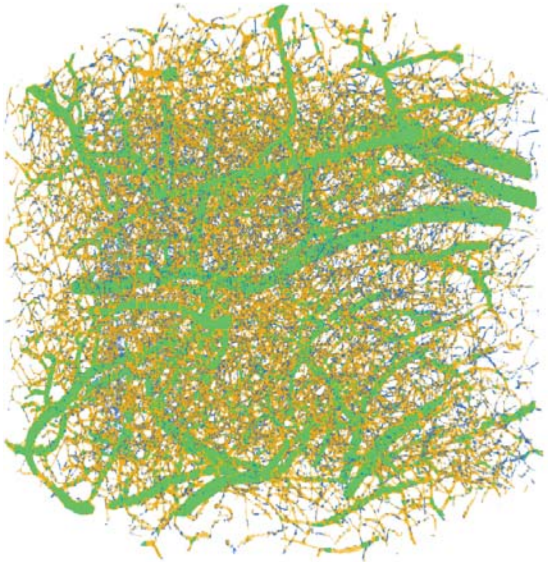
Table 6.3: Memory requirement and brick distribution for the three different brick sizes of the vessel data set.

We also observe that a smaller brick type can increase the performance because of the following reasons. The number of empty and dense volume bricks increases, where these have a fast access time. Octrees are constructed for a smaller brick size, which improves the octree traversal. Finally, also point lists tend to get shorter, with a smaller brick size. The drawback with respect to the performance is that the amount of branching on the GPU can increase. For example, for stencil look-ups (e.g., filter kernels) it is more likely to hit brick boundaries with a smaller brick size.



(a) Brick size $32 \times 32 \times 32$.

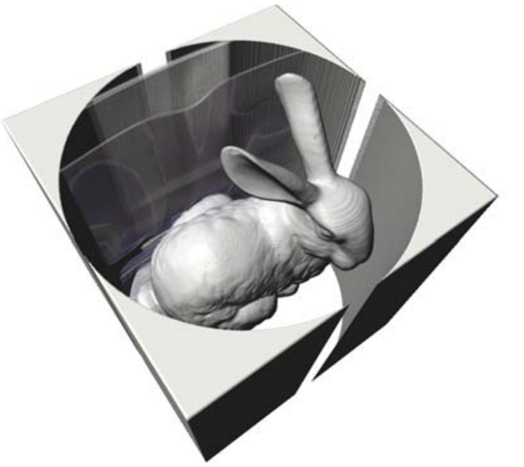
(b) Brick size $16 \times 16 \times 16$.



(c) Brick size $8 \times 8 \times 8$.

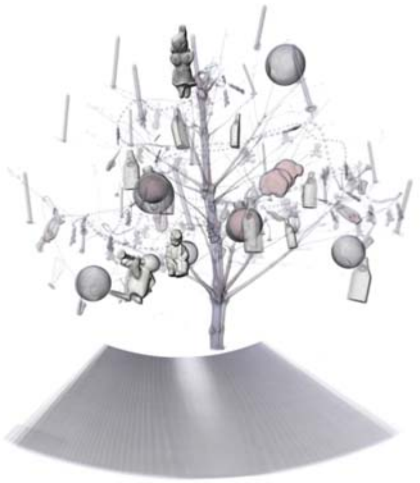
Figure 6.6: Rendering of the vessel data set with different brick sizes, voxels are colored depending on the type of their including brick. Dense voxels are green, octree voxels are orange and list voxels are blue.

(a) Bonsai

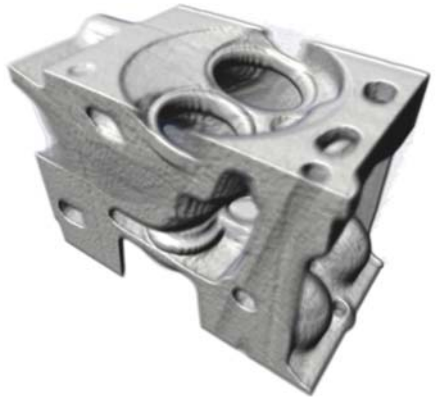


(b) Bunny

(c) Carp

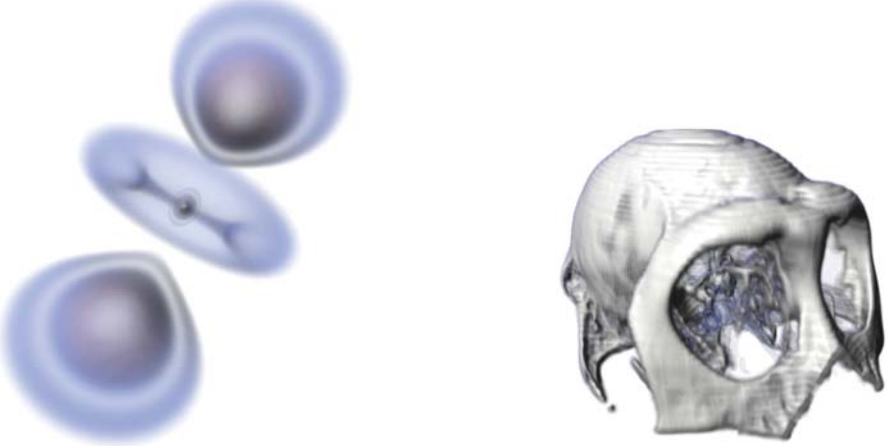


(d) Christmas tree



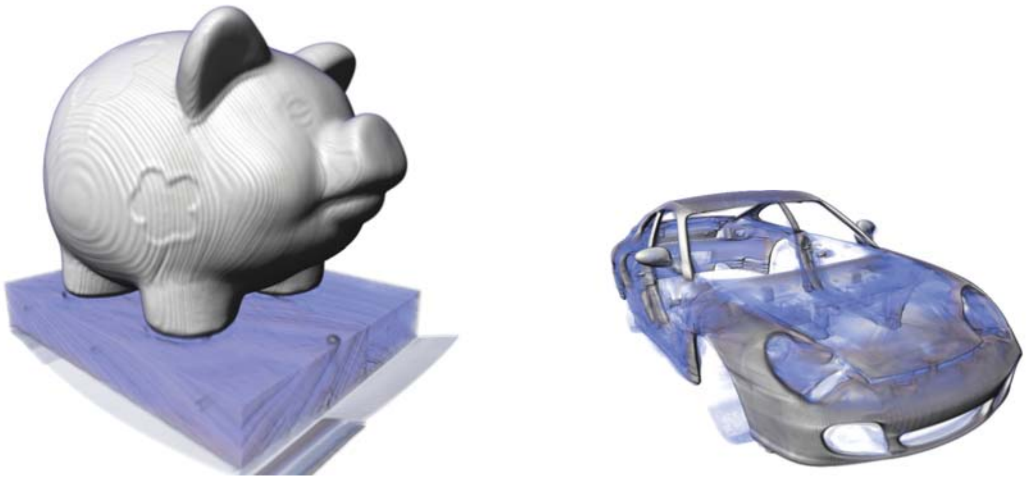
(e) Engine

(f) Frog



(g) Hydrogen

(h) Monkey-CT

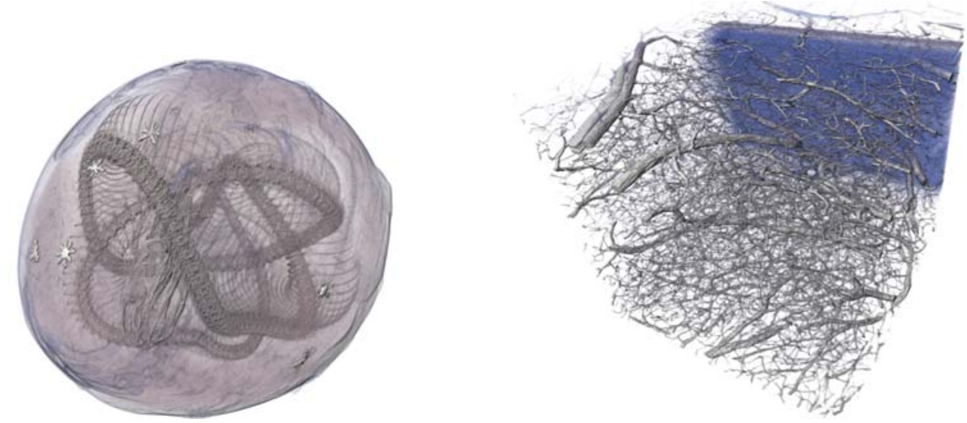


(i) Piggy Bank

(j) Porsche

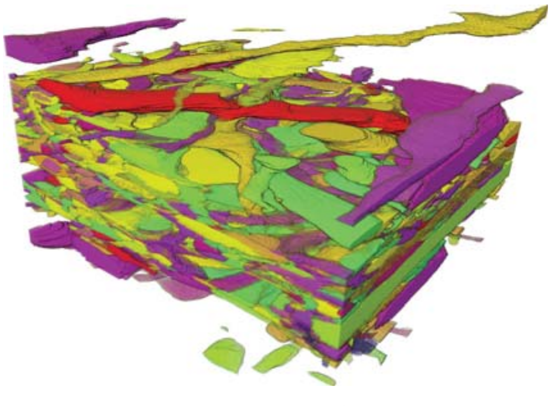
(k) Schaedel

(l) Stag beetle



(m) Kingsnake

(n) Vessels



(o) Connectomics

Figure 6.7: Volume rendering of the benchmark data sets.

CHAPTER 7

Summary and Discussion

We were interested in finding an efficient data structure for sparse volumes on the GPU. It should have a low memory requirement and a high access performance. We started experimenting with a modular approach (see Section 3.3). The idea was to combine different data structures with varying properties to generate an adaptive representation. The goal of this data structure was to adapt to different data sets automatically and also to take the algorithm that is executed on the data into consideration. Our initial concept of a modular data structure combines a large variety of different data structures and connects them by pointers.

Experimenting with such a modular data structure led to *hybrid bricking*. Hybrid bricking is a volume bricking variant that incorporates four different brick types: dense, octree, voxel list and empty bricks. The advantage of a bricking approach over other hierarchical data structures is that it provides good access performance. It requires only a single indirection, but includes enough flexibility to efficiently adapt to many data sets. The approach makes use of the observation that a single volume data-set typically contains regions of varying sparsity. The brick types we have chosen complement each other well and provide an efficient representation for regions of varying sparsity.

Our results show, that hybrid bricking is very efficient in generating representations with a low memory consumption. The proposed initialization routine can generate the representation with the lowest memory consumption for a given brick size efficiently. This was achieved by calculating the memory consumption for each brick type in parallel as explained in Section 5.1.

The drawback of using a sparse data structure is that a representation which reduces the memory requirement typically introduces a performance overhead. Non-hierarchical representations such as bricking and hashing introduce at least one indirection. Hierarchical representations such as the octree require a costly tree traversal step to retrieve a specific data value. In our hybrid bricking approach we introduce an access overhead through the bricking on the one hand, on the other hand we introduce an overhead for each brick type. The overhead through bricking involves type routing and following a pointer to the elemental brick data structure. Additionally, each elemental brick type, i.e., the dense, octree, voxel list and empty brick, has a different performance overhead as explained in Chapter 3. The empty type and the dense representation are the fastest, the octree and voxel list access performance depends on the data.

Reducing the overhead of the elemental data structures can only be done by improving the performance of their concrete implementations. We have done this for voxel lists by employing a binary search. For the octree we were manually experimenting with low level optimizations as mentioned in Section 5.2.1. Additionally, we keep the node data structure as minimal as possible to reduce the cost of global memory access. The tuning of the elemental data structures can be done independently from the implementation of the hybrid bricking data structure. Improving a single elemental brick type can further increase the performance of the overall approach.

To reduce the overhead of hybrid bricking we apply a data dependent specialization technique, which we call *JiTTree*. JiTTree is a kd tree of bricks which is unrolled in a data dependent fashion to OpenCL source code. After the elemental data type with the lowest memory consumption for each brick is determined, the kd tree is constructed. Each kd leaf either stores a single brick or a number of bricks with the same type. After the kd tree is constructed on the application side, data dependent OpenCL code for kd tree traversal is constructed and compiled. One limitation is that large trees result in a large number of lines of code which in turn results in a high OpenCL compilation time on current OpenCL platforms. Our approach shows that an unrolled kd tree can provide a high performance alternative to a traditional kd tree if the tree is not too large. Such an unrolled kd tree is able to shift some of the data memory fetches from global memory to either instructions or local memory look-ups.

Our experiments with the JiTTree reveal that it can provide a performance benefit for some access patterns. In specific, the approach was beneficial for regular access patterns such as the stencil look up as used in mean filtering. Random access such as ray traversal could not benefit from the JiTTree yet. In the following two sections we point out benefits and limitations of the approach as well as there implications which point to potential new challenges.

7.1 Benefits

A central benefit of our *hybrid bricking* approach is that it allows to store sparse volume data with a significantly reduced memory consumption. The concept of combining different data structures to generate an overall better representation turned out to be useful especially for the memory consumption. Hybrid bricking provides enough flexibility but keeps the traversal overhead to a minimum, since it only introduces one additional indirection. Our approach is extensible, since it is not restricted to the four brick types we have used. It can easily incorporate different, new data structures. A higher number of different elemental data structures could results in an even better adaptation to specific data sets. For example it could be possible to incorporate compression based elemental brick types. Such compression based data structures make use of the distribution of data values in the value range to further reduce the memory consumption.

Such a hybrid data structure can also be useful in a production setting. It is easy to start with a basic bricking with two brick types such as dense and empty. The usage of a hybrid data structure can steadily improve the data representation without invalidating previous developments. It is possible to subsequently develop additional elemental data structures which improve the previous representation.

Another advantage of our hybrid bricking approach is, that for a given brick size and a fixed number of brick types, it can automatically find the representation with the lowest memory consumption. Nevertheless, besides the representation with the lowest memory consumption, there exist many other possible brick assignments. The representation with the best run-time performance is generally the one using only empty and dense bricks. Our data structure can also accommodate intermediate representations between the one with the best run-time performance and the lowest memory consumption. We have investigated simple strategies to pick such intermediate representations, but more work has to be done to come up with a more effective approach.

We have shown that the *JiTTree* increases the access performance in certain cases by employing a data dependent tree unrolling approach. For the stencil access pattern we were able to achieve an improvement of the access performance by 1.29 over all our test runs. For the ray traversal access pattern, we were only able to improve the access performance for smaller data sets. In pure hybrid bricking, the introduction of a larger number of brick types will result in a higher type routing cost. This is not the case for the JiTTree, since its traversal cost is independent of the number of different data types, but still depends on the homogeneity of the brick distribution.

7.2 Limitations

A general problem with hybrid bricking is its performance overhead. The more obvious performance cost comes from type routing between different traversal routines. We have shown that this type routing can be improved by the JiTTree.

In addition to type routing, the approach introduces an additional cost through branch divergence when executed on the GPU. Let us consider the example of an octree brick next to a voxel list brick. An algorithm executed on the data introduces branch divergence if it accesses two values, one from each brick from two threads of the same warp. As mentioned in Section 5.3, this effectively results in sequential traversal of the code in both branches. This branch divergence cannot easily be resolved, since the type assignment of the neighboring bricks is fixed. For certain access patterns, such as stencil access, it would be possible to reduce this branch divergence by storing redundant data values for neighboring bricks. For other techniques, such as ray casting the reduction of branch divergence is harder to manage.

As mentioned previously, the JiTTree can improve the performance of hybrid bricking. Another aspect that should be considered is that the JiTTree introduces an increased implementation complexity. Generating a prototype can be done fast, but fine tuning is necessary. The code generation step in the JiTTree creates very simple traversal code containing mainly if/else branches. This approach is less error prone to unnecessary operations. To generate a fair comparison between hybrid bricking and JiTTree it was necessary to optimized both approaches.

Finally, in addition to the access performance and implementation cost, JiTTree introduces an additional compilation step. The performance of the compilation depends on the size of the underlying kd tree. A higher number of nodes results in a the higher compilation cost. While the application is able to generate such a tree and the source code fast, the current OpenCL compiler is the bottleneck. For bricking a case with more than 32^3 bricks, the current OpenCL compiler might fail. It results in either very high compile times, or the compiler can terminate with errors.

CHAPTER 8

Conclusion and Future Work

We have presented two promising ideas. The first was the introduction of a hybrid data structure, which was deduced from a modular data structure. We believe that a more general modular data structure could eventually lead to a variety of high performing data structures which are not known or widely used today. The second concept was to use data dependent code specialization on the GPU. We suspect that other GPU based applications could benefit from data dependent code specialization. A further investigation could lead to new insight about the benefits and drawbacks in a more general setting. From the advantages and limitations we can infer some possible directions for future work for hybrid bricking and for the JiTTree in Section 8.1. In the last Section 8.2 we shortly cover the concept of iteration delaying as a technique that could be used to reduce branch divergence.

8.1 Hybrid Bricking and JiTTree

One obvious extension to hybrid bricking would be to implement additional brick types. An approach taking compression into consideration can decrease the memory consumption for certain types of data. A spatially hashed brick type could be another interesting data structure, which could complement the existing ones.

In the current implementation, our data structure is statically generated and write access is impossible. To support writing into the data structure, we would have to rearrange the data structure layout and implement a more complex memory management. For example, deletion can result in memory fragmentation. To counterbalance fragmentation, it is necessary to rearrange the data at run-time to free memory and keep large, contiguous regions of memory available for future allocations. Memory fragmentation is easily handled in a dense volume bricking approach, due to the fixed memory requirement of each brick. To avoid fragmentation, it is possible to insert new bricks into previously deleted locations. Octree and voxel list brick types are of variable size and would therefore be more difficult to use in a dynamic memory environment.

For very large data sets a multi-resolution out-of-core representation similar to the work of Hadwiger et al. [16] might be of interest. High resolutions can only be transferred to the GPU on demand, low resolutions can be used if less detail suffices. The problem with such an approach is that it also requires dynamic memory handling. To counterbalance memory fragmentation, the memory efficient representation could only be used for the transfer to the GPU. This representation can then be directly written to a traditional dense volume brick representation on the GPU.

Even though we have shown that hybrid bricking is a feasible solution, it could be of interest to further investigate other modular data structures. The most basic example would be a data structure that uses a hashing approach instead of the bricking. But an even more interesting direction of further work could concentrate on the concept of a modular data structure. In Section 3.3 we mentioned, that a modular data structure would be more complex to initialize. It could be interesting to come up with such an initialization routine, as in some cases a modular data structure could outperform hybrid bricking. For example, typical octrees in volume rendering store bricks in its leaves. Hybrid bricking does not allow such a data structure that contains an octree at the root level and bricks instead of single values at its leaves. There could be a number of data structures which have not been used yet. An automatic routine to generate a larger variety of data structures could come up with promising candidates for more efficient data structures.

One further drawback of the bricking approach, is that it imposes a regular splitting of the data in equally sized bricks. We counterbalance this by allowing different brick types, but it could be of interest to chose different ways of grouping data values. The bricking approach stores locally adjacent values in close regions of memory for most cases. Even though we hypothesize that this is advantageous for most access patterns, it could be that other access patterns benefit from different ways of grouping. For example, a slicing approach could benefit from a data structure where individual slices of data are stored in adjacent regions of memory. An algorithm that can automatically find the optimal representation for a given algorithm could find new data structures for certain access patterns.

The just-in-time compilation approach also opens up a large number of possible scenarios. Our evaluation shows that the JiTTree was not able to improve the performance for *all* data sets and algorithms. We compared the performance, i.e., of a bricking which requires $O(1)$ steps to an unrolled kd tree traversal which requires $O(log(k))$ steps. Still, we were able to achieve a performance gain with our kd tree approach. This leads to the assumption that an unrolled tree representation on the GPU could perform very well for other cases, where hierarchical representations are used.

One restriction of our approach is that we construct the just-in-time compiled kd tree on top of a bricking approach. But, the kd tree would also make it possible to employ arbitrary splits through the data set, which do not have to be aligned to the brick boundaries. This would open up a large number of different representations. For example, it would be possible to position octrees in arbitrary locations in the volume to make use of their shift sensitivity. Shifting an octree slightly, can reduce the memory consumption since it may require a different number of nodes. Further it could be possible to generate kd leafs spanning multiple bricks. This is especially easy to implement for larger dense regions or voxel lists.

A further restriction is connected to the algorithm which assigns types to the bricks. The optimal memory representation is not necessarily the highest performing one. Even though for our specific brick types the performance adapts well (see Chapter 3), different strategies to assign brick types can be of interest. Two such strategies seem promising. The first would prefer brick types with a high access performance over the brick type with the lowest memory requirement. This approach was introduced in Chapter 5. The second strategy evolves out of the JiTTree setting. The traversal depth of the JiTTree depends on the size of the leaf nodes. Large leaf nodes result in a higher performance. Intelligently swapping brick types to increase the size of the kd leaves could lead to a performance benefit in this case.

In the future we would like to evaluate our approach in a storage and data transfer scenario. Our data representation can provide an efficient method to store volume data-sets on the hard-disk. In addition to the lower memory consumption on the disk, this can also increase the transfer rate of data sets from and to the GPU.

We have shown, that modern GPUs can handle complex data structures reasonably well. Complex data structures on the GPU are often hard to profile with respect to performance. We presented certain factors which seem plausible to influence the performance of the JiTTree. A further investigation of specialized, data dependent traversal code of a more general hierarchical data structure could lead to more insights. It would be useful to restrict the scenario to simpler cases which can be explored more easily.

There are two possible improvements to our data structure which can be important in volume rendering. For access at non-integer locations, trilinear interpolation is necessary. A possible extension to our approach would be to store data in textures where possible. This could allow the hardware to do the interpolation.

8.2 Iteration Delaying

Iteration delaying was introduced by Han and Abdelrahman [17]. It can be used if threads (work-items) iterate over a loop and the body of the loop contains branching code. If all threads run in the same warp, the branching can cause branch divergence (see Section 5.3).

We explain the benefit of iteration delaying with an example. Let us consider a stencil operation on one dimensional data, such as a mean filter with a kernel size of 3. Figure 8.1 shows a visualization of the execution of such a filter. We assume that our data is organized as a bricking approach with a brick size of two. We only have two different bricks: the F-brick shown in dark gray and the the T-brick shown in bright gray. For retrieving a value, we first make a look-up into the type array and determine the type. If the type is T we route to the T-type traversal code, otherwise we route to the F-type traversal code. A simple pseudo code of the filter is shown in Listing 8.1. For each stencil, a for-loop with three iterations is executed, each look-up is routed to either the T-type or the F-type.

We now consider three different threads (A, B and C) each in the same warp. Each thread starts its for-loop iteration shifted one element to the right. For example, thread A starts in the branch corresponding to type F and ends in the branch of type T.

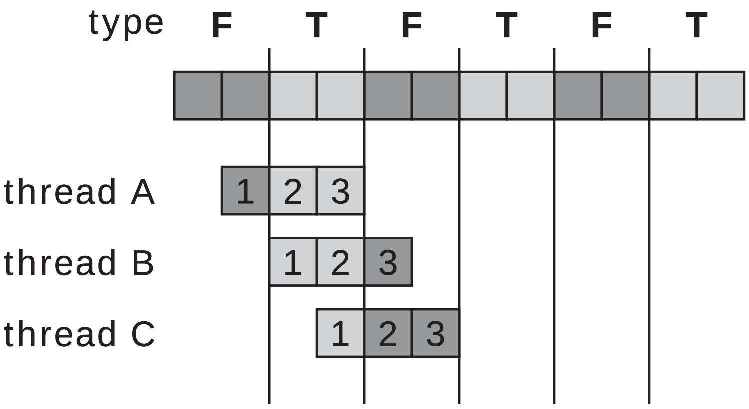


Figure 8.1: Code visualization of a stencil access for three different threads A, B and C. Thread A accesses the F branch in the first look-up and the T branch in the second and third.

```
// for loop accessing three elements, some branch to the T-type
// others to the F-type
for (int i=start; i<start+3; i++)
{
  float result = 0;
  if (type[i])
  {
    result += routeTBranch();
  }
  else
  {
    result += routeFBranch();
  }
}
result /= 3;
```

Listing 8.1: Pseudo code of a kernel for mean filtering on one dimensional data.

Figure 8.2 shows a visualization of the parallel execution of the three threads from top to bottom: *Thread A*, *Thread B* and *Thread C*. Each thread executes a loop with three iterations. Inside this loop the thread can take either the T- or the F-branch. For example, *thread A* executes the following branches: F, T, T. The total execution time of this code is six (abstract) cycles. So for example, in the first loop iteration, thread A has to wait while thread B and C execute the T branch. Then, B and C wait while A is in the F branch.

With iteration delaying the run-time can be improved. It provides a mechanism to rearrange the execution of branches while keeping the local order for each thread, this results in a more efficient traversal. Figure 8.3 shows the result of iteration delaying. The execution time now only requires five cycles.

Our approach could benefit from iteration delaying. In addition to iteration delaying, some access patterns, such as the mean filter, are independent of the retrieval order. So it could be further possible to speed up the execution time to four cycles by changing execution order of

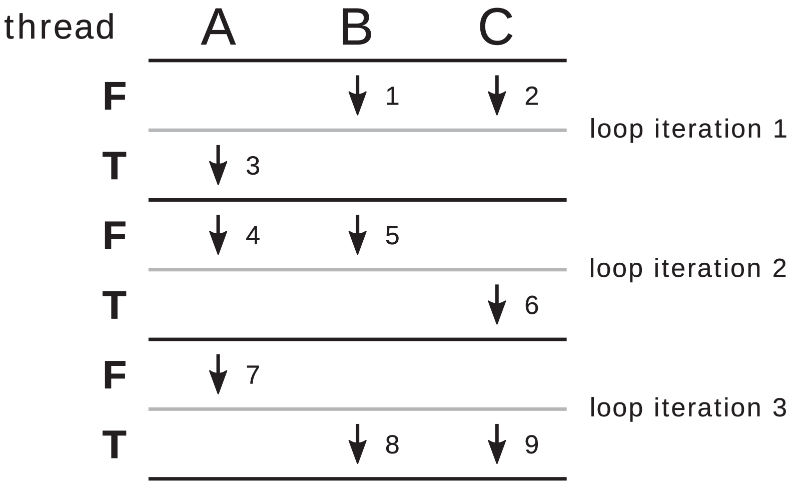

thread
A
B
C
F
1
2
loop iteration 1
T
3
F
4
5
loop iteration 2
T
6
F
7
loop iteration 3
T
8
9

Figure 8.2: The three threads require six cycles to be executed without iteration delaying. Due to branch divergence the code in one thread has to wait for the other threads while they execute different code.

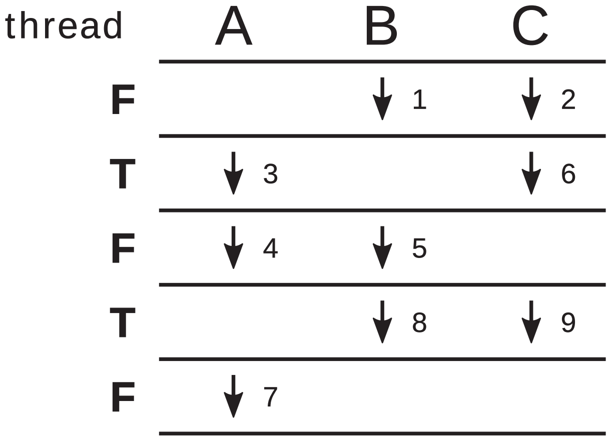

thread
A
B
C
F
1
2
T
3
6
F
4
5
T
8
9
F
7

Figure 8.3: With iteration delaying the code only needs five cycles. The relative order inside each thread remains the same, but the rearrangement can reduce the number of cycles.

some branches. To achieve this we could execute 4, 1 and 2 in the T branch, then 3, 8 and 6 in the F branch, 5 and 7 in the T branch and finally 9 in the F branch.

APPENDIX A

Morton code

```
ulong zorder3d(ulong x, ulong y, ulong z)
{
    const ulong B[] = {0x00000000FF0000FF, 0x000000F00F00F00F,
        0x00000C30C30C30C3, 0x0000249249249249};
    const ulong S[] = {16, 8, 4, 2};

    x = (x | (x << S[0])) & B[0];
    x = (x | (x << S[1])) & B[1];
    x = (x | (x << S[2])) & B[2];
    x = (x | (x << S[3])) & B[3];

    y = (y | (y << S[0])) & B[0];
    y = (y | (y << S[1])) & B[1];
    y = (y | (y << S[2])) & B[2];
    y = (y | (y << S[3])) & B[3];

    z = (z | (z <<  S[0])) & B[0];
    z = (z | (z <<  S[1])) & B[1];
    z = (z | (z <<  S[2])) & B[2];
    z = (z | (z <<  S[3])) & B[3];

    return ( x | (y << 1) | (z << 2) );
}
```

Listing A.1: Transforms a 64 bit three-dimensional coordinate to Morton code.[1]

---

[1] `http://stackoverflow.com/questions/1024754/how-to-compute-a-3d-morton-number-interleave-the-bits-of-3-ints`, Dec. 2015

Bibliography

[1] Jeroen Baert, Ares Lagae, and Philip Dutré. Out-of-core construction of sparse voxel octrees. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 27–32, New York, NY, USA, 2013. ACM.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[3] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. Diderot: A parallel DSL for image analysis and visualization. In *Proceedings of the ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation (PLDI '12)*, pages 111–120, 2012.

[4] Hyungsuk Choi, Woohyuk Choi, Tran Minh Quan, David GC Hildebrand, Hanspeter Pfister, and Won-Ki Jeong. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2407–2416, 2014.

[5] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 493–501, New York, NY, USA, 1993. ACM.

[6] NVIDIA Corporation. CUDA FAQ. `https://developer.nvidia.com/cuda-faq`, 2015. [Online; accessed 1.12.2015].

[7] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar 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.

[8] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, March 1974.

[9] Tim Foley and Jeremy 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.

[10] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, September 1977.

[11] Sarah F. Frisken and Ronald N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(3):1–11, 2002.

[12] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys (CSUR)*, 30(2):170–231, 1998.

[13] Irene Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, December 1982.

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

[15] Khronos OpenCL Working Group. The OpenCL specification. *version*, 1.29, 2008.

[16] Markus Hadwiger, Johanna Beyer, Won-Ki Jeong, and Hanspeter Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, Dec 2012.

[17] Tianyi David Han and Tarek 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, New York, NY, USA, 2011. ACM.

[18] Mark Harris. Mapping computational concepts to GPUs. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH ’05, New York, NY, USA, 2005. ACM.

[19] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive kd tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174. ACM, 2007.

[20] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[21] Ralf Kähler, Mark Simon, and Hans-Christian Hege. Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):341–351, July 2003.

[22] Armin Kanitsar, Thomas Theußl, Lukas Mroz, Milos Srámek, Anna Vilanova Bartrolí, Balázs Csébfalvi, Jirí Hladuvka, Dominik Fleischmann, Michael Knapp, Rainer Wegenkittl, et al. Christmas tree case study: Computed tomography as a tool for mastering complex real world objects with applications in computer graphics. In *Proceedings of IEEE Visualization*, pages 489–492. IEEE Computer Society, 2002.

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

[24] Aaron Knoll, Ingo Wald, Steven Parker, and Charles Hansen. Interactive isosurface ray tracing of large octree volumes. In *IEEE Symposium on Interactive Ray Tracing 2006*, pages 115–124, 2006.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[26] Matthias Labschütz, Stefan Bruckner, M. Eduard Gröller, Markus Hadwiger, and Peter Rautek. JiTTree: A just-in-time compiled sparse GPU volume data structure. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):1025–1034, 2016.

[27] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[28] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *ACM SIGGRAPH*, pages 579–588, New York, NY, USA, 2006. ACM.

[29] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Octree textures on the GPU. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 37, pages 595–613. Addison Wesley, March 2005.

[30] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006.

[31] Justin Luitjens. Global memory usage and strategy. `http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GlobalMemory.pdf`, 2011. [Online; accessed 1.12.2015].

[32] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

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

[34] Jürg Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.

[35] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[36] Timothy J Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50. Eurographics Association, 2003.

[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

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

[39] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, Hyouk Joong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. *ACM SIGPLAN Notices*, 48(1):497–510, 2013.

[40] Daniel Ruijters and Anna Vilanova. Optimizing GPU volume rendering. In *Winter School of Computer Graphics (WSCG) 2006*, pages 9–16, 2006.

[41] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[42] Hanan Samet. Spatial data structures. Addison-Wesley, 1995.

[43] Nadathur Satish, Mark Harris, and Michael 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.

[44] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle 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.

[45] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics (TOG)*, 11(3):201–227, July 1992.

[46] Nicholas Wilt. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[47] N. Cliff Woolley. Introduction to OpenCL. `http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/06-intro_to_opencl.pdf`, 2010. [Online; accessed 1.12.2015].

[48] Yao Zhang and John 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.

[49] Gernot Ziegler. *GPU data structures for graphics and vision*. PhD thesis, Universität des Saarlandes, 2010.

[50] Gernot Ziegler, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. GPU point list generation through histogram pyramids. In *Proceedings of VMV (2006)*, pages 137–141, 2006.