

A Versatile and Efficient GPU Data Structure for Spatial Indexing

Jens Schneider and Peter Rautek

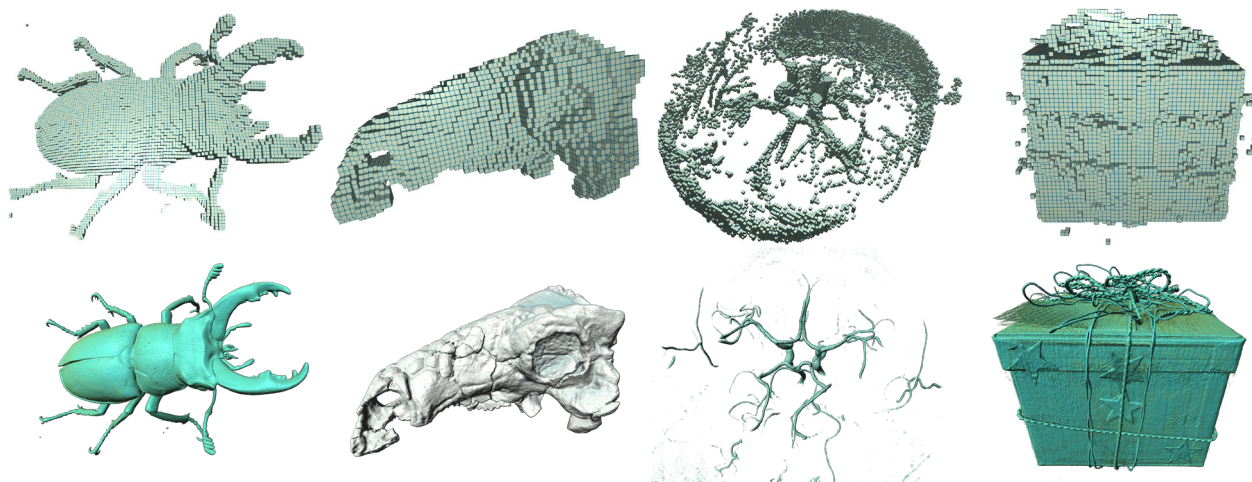


Fig. 1. Many real-world volume data sets are sparse, lending appeal to rendering and processing techniques that benefit from such a property. Using our novel, storage-efficient data structure allows us to efficiently store such data sets as bricked textures on the GPU. From left to right: Stagbeetle ($832 \times 832 \times 494$, bricksizes 7^3 , 5.01% non-empty, 245.5KB index); Pawpawsaurus Campbells skull ($958 \times 646 \times 1088$, bricksizes 15^3 , 15.1% non-empty, 50.21KB index); Angiography ($416 \times 512 \times 112$, bricksizes 3^3 , 2.27% non-empty, 220.5KB index); Present ($492 \times 492 \times 442$, bricksizes 7^3 , 17.3% non-empty, 78.80KB index). Our data structure introduces only minimal overhead in terms of memory and run-time, and it allows for efficient brick-level empty-space skipping.

Abstract—In this paper we present a novel GPU-based data structure for spatial indexing. Based on Fenwick trees—a special type of binary indexed trees—our data structure allows construction in linear time. Updates and prefixes can be computed in logarithmic time, whereas point queries require only constant time on average. Unlike competing data structures such as summed-area tables and spatial hashing, our data structure requires a constant amount of bits for each data element, and it offers unconstrained point queries. This property makes our data structure ideally suited for applications requiring unconstrained indexing of large data, such as block-storage of large and block-sparse volumes. Finally, we provide asymptotic bounds on both run-time and memory requirements, and we show applications for which our new data structure is useful.

Index Terms—GPU-based Data Structures, Binary Index Trees, Sparse Data

1 INTRODUCTION

With the recent advances in general purpose GPU computing, data structures enabling efficient algorithms on this challenging architecture are becoming more important. Our novel approach adds one such data structure to the library of spatial indexing data structures for GPUs. The proposed data structure is versatile in the sense that it can be used in arbitrary dimensions and for arbitrarily complex data types. Unlike some competing data structures, our method offers unconstrained access to data elements, while at the same time requiring only a minimal memory overhead. Although our data structure is versatile, with potential applications including dynamic stream compaction, prefix computation, summed-area-table-like integration, etc., we believe that the special use case of block-sparse data representation is potentially of the highest impact.

As the immense compute powers of modern graphics cards and supercomputers alike increase, scientists from a wide range of fields are able to generate an ever increasing amount of data. Visualizing this data, even on the machine it was generated, remains a challenge. While GPUs have been widely accepted as one of the best technologies to tackle this problem, GPU memory remains a scarce resource. Our proposed data structure seeks to address this issue, while providing fast and unconstrained access performance.

The underlying principle of the proposed data structure is the storage of partial sums of the data. By interleaving verbatim storage of data elements with partial sums of 2^k elements (i.e., sums of two, four, eight, etc. elements), we form a tree that is intuitively best described as being half-way between input values and a summed area table. Its properties are therefore also half-way between a regular array and a summed area table—while summed area tables excel at prefix sums and point queries in constant time, they fail at achieving memory efficiency for large amounts of data elements. In contrast, our data structure trades speed for memory. By only storing partial sums, prefixes can be quickly reconstructed in logarithmic time while at the same time using considerably less memory.

It is interesting to note that the work-efficient parallel implementation of the prefix sum [14, 3, 19, 10] uses a binary indexed tree of partial sums that is equivalent to our data structure as an intermediate representation in the first (*up-sweep*) phase of the algorithm. However,

• Jens Schneider and Peter Rautek are with the Visual Computing Center (VCC) at King Abdullah University of Science and Technology (KAUST). E-mail: {jens.schneider|peter.rautek}@kaust.edu.sa.

Manuscript received 31 Mar. 2016; accepted 1 Aug. 2016. Date of publication 15 Aug. 2016; date of current version 23 Oct. 2016.

For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TVCG.2016.2599043

the second (*down-sweep*) phase of the computation discards the tree in favour of the summed area table. Fenwick [8], apparently unaware of the previous work on parallel prefix sums, was the first to propose a tree of partial sums as the actual data structure, instead of using it only as an intermediate result. Commonly referred to as Fenwick trees, the data structure allows for prefix sums and data updates in logarithmic time. To the best of our knowledge, we are the first to describe a GPU implementation of a spatial index data structure that uses a Fenwick tree as the underlying primitive.

Contributions. We provide an extensive asymptotic analysis of the memory- and run-time requirements for the general d -dimensional case. We furthermore show that point queries can be performed in constant time on average, a result that to the best of our knowledge has not been documented before. We also provide a specialized implementation that is optimized for the important case of block-sparse data representations on the GPU. We report run-time and memory requirements of our highly practical implementation and we demonstrate that this new data structure outperforms competing approaches in several aspects and in several important scenarios.

2 RELATED WORK

Fenwick [8] proposed a novel binary indexed tree originally for the efficient storage, update and retrieval of cumulative frequencies for large symbol alphabets. However, the data structure proved to be more general and was reused in many contexts. Mishra [18] presents a d -dimensional extension of the binary index tree. We base our work on these data structures and analyze their suitability as spatial indexing data structure on the GPU. In the following we review other GPU data structures suitable for spatial indexing, most notably spatial hashing, summed area tables, histogram pyramids, and octrees.

Spatial Hashing. Lefebvre and Hoppe [16] propose to use perfect spatial hashing to efficiently encode non-empty data locations for computer graphics applications. While they guarantee $\mathcal{O}(1)$ access and very low storage requirements (for low fill density), the construction of the hash is expensive. Furthermore, *unconstrained* access, i.e., access that allows for the identification of empty data *not* stored in the table, requires additional storage and implementation overhead. García et al. [9] improve on the data coherence and construction speed of spatial hashing by utilizing coherent hash functions and parallel construction on a GPU. They also offer unconstrained access, but this comes at the cost of storing *key,value* pairs, a significant memory overhead that seems only feasible for very low fill densities. This is further aggravated by the fact that a key age is stored in an additional 4 bits to allow for fast unconstrained access. In contrast to hashes, Fenwick trees natively offer unconstrained access. Being a compact yet dense representation, Fenwick trees are *fill-agnostic*. Therefore, they cope with high fill densities extremely well. For instance, hashing a 8192^2 image requires 30 bits per non-empty entry. This includes 26 bits to store the key or pixel position plus 4 age bits. In contrast, a Fenwick tree would require 16MB (1d) or 24MB (2d). This means that a Fenwick tree is at least as compact as the hash if 6.67% (1d) or 10.0% (2d) or more of the entries are non-empty. Note that for larger images or higher fill ratios, the break-even point shifts further to the advantage of Fenwick trees.

Summed Area Tables. Crow [5] introduced summed area tables for efficient texture mapping and filtering. Today, they are used in many contexts to quickly compute the sum of all elements in a rectangular sub-space. Blelloch [2, 3] describes efficient parallel implementations of prefix scans, a closely related problem. Sengupta et al. [19], and Harris et al. [10] present GPU-based implementations that construct the summed area tables in parallel and in a work efficient manner. Hensley et al. [11] show real-time graphics applications using summed area tables. The benefits of summed area tables are (i) very fast access times and (ii) summed area computations. However, the update of a summed area table is expensive, since, essentially, it needs to be rebuilt from scratch. An even more important drawback

of summed area tables in the context of large data visualization is their memory inefficiency that stems from the need to store very large numbers to represent the cumulative sums.

Histogram Pyramids. Ziegler et al. [21] and Dyken et al. [7] present *Histogram Pyramids* that are used to hierarchically represent a histogram of non empty spatial regions. The data structure is used for parallel stream compaction which produces a densely packed output stream by omitting empty regions. This stream compaction is used in applications such as real time point cloud generation as well as iso-surface extraction using marching cubes. In contrast to our approach Histogram Pyramids are *over-complete* in the sense that they store more output elements than input elements. It is also worth noting that unlike our data structure, histogram pyramids perform strictly 1d prefix scans and cannot be used as a replacement for higher dimensional summed area tables.

Hierarchical Space Partitioning. Benson and Davis present octree textures [1] for storing sparse solid textures. Their approach uses a sparse octree that represents colors at the intersections of nodes with a surface model. Lefebvre [17] present a more recent implementation of octree textures on the GPU. Octrees have been generalized to N^3 trees, in which each node holds pointers to its N^3 children. Crassin et al. [4] present a fast implementation of N^3 trees that is used for voxel-based ray casting of interfaces in large scenes. Labschütz et al. [13] present JiTTree, a hybrid data structure that subdivides the input domain into regular bricks. For each brick, the memory-optimal data structure is computed. The traversal code is just-in-time compiled to speed up access times. We show that our data structure represents sparse volumes with memory requirements similar to JiTTree. This could make our data structure a good candidate for use in JiTTree's hybrid approach.

Adaptive Texture Maps. Kraus et al. [12] propose a sparse data structure in which an input image or volume is decomposed into bricks. To retrieve the bricks, an array of indices is stored verbatim. While the data structure is fast due to the fact that it needs only one memory indirection, maintaining a large amount of non-empty bricks results in a logarithmic storage overhead per index in the amount of bricks, similar to the memory requirements of summed area tables. This may result in a difficult tradeoff between the amount of bricks used to represent the data (and thus, the memory overhead) and the ability to discard large empty regions.

3 METHODOLOGY

In this section, we will first review the concept behind the 1d binary index trees presented first by Fenwick [8]. We will then provide a generalization to higher dimensions similar to [18], followed by an asymptotic analysis of both run-time and space complexity.

3.1 Review of Fenwick Trees

Fenwick trees [8] are binary index trees that allow for efficient updates and prefix computations in logarithmic time, and point queries in constant time. This is achieved by storing partial prefix sums across all scales. In what follows, we present a variation of the original Fenwick tree description. Unlike the original publication that treats the 0th data entry specially to mitigate 1-based counting of data elements, we treat all input data in the same fashion. This has several benefits, and, most importantly, it exposes a previously unnoticed close relation between the Fenwick tree construction and the lifting scheme for wavelets [20, 6].

Notation. Throughout this paper we will use the following notation.

d : the spatial dimensionality of the data

N : amount of data values along each axis

K : the amount of non-empty data values (fill)

ρ : fill density, $\rho = K/N^d$

For instance, a 1024^3 volume with 32M non-zeros will have $N = 1024$, $d = 3$, $K = 32M$, and $\rho = 1/32$.

We define the k th (exclusive) prefix of \mathbf{x} as

$$p(\mathbf{x}, k) := \sum_{i=0}^{k-1} x_i, \quad (1)$$

where $\mathbf{x} := \{x_i\}_{i=0}^{n-1} \in \mathbb{N}_0^n$ is the input data consisting of non-negative integers.

3.1.1 Construction

Following the wavelet lifting idea [20, 6] as illustrated in Fig. 2, we split the sequence of input data elements x_i into *even* and *odd* positions. Data elements at even positions are stored verbatim in the final Fenwick tree. Data elements at odd positions are lifted to the next level by adding their even predecessor. The new level (odd+even elements) consists of half the elements of the previous level and is the new input to the recursive lifting scheme. The result is a hierarchical representation of partial sums. Figure 3 provides an exemplary Fenwick tree

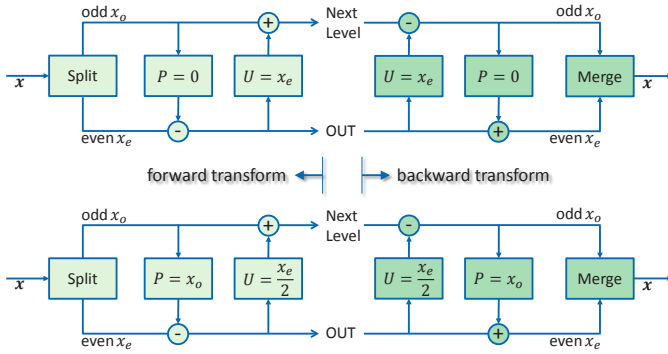


Fig. 2. Lifting scheme of the Fenwick tree (top) and, for comparison, the lifting scheme of the Haar Wavelet (bottom). The two schemes differ in their prediction (P) and update (U) steps. This results in range expansion and increased storage requirements for even elements x_e in the Haar Wavelet when compared to the Fenwick tree

construction. The top row shows an example of 1d input data. Each element consists of a value (top of rounded rectangle) and a range of indices that were summed up in this element (bottom of rounded rectangle). The second row shows the first partial sums. Odd elements are lifted to the next level (the third row in Fig. 3). This process is repeated recursively until only one element is left that contains the sum of all elements.

Algorithm 1 provides pseudo-code for a linear time, sequential construction of a 1d Fenwick Tree.

Algorithm 1 Fenwick Tree Construction (in-place)

```

procedure FWTBUILD( $\mathbf{x}$ )                                ▷  $\mathbf{x} := (x_0, \dots, x_{n-1})$ 
   $s \leftarrow 2$                                            ▷ stride
   $i_0 \leftarrow 1$                                        ▷ start position
  while  $i_0 < n$  do                                     ▷ iterate levels
    for  $i = i_0 \dots n-1$  step  $s$  do
       $x_i \leftarrow x_i + x_{i-1}$                          ▷ update odd positions
    end for
     $i_0 \leftarrow i_0 + s$                                ▷ update start position
     $s \leftarrow 2s$                                      ▷ update stride
  end while
  return  $\mathbf{x}$ 
end procedure

```

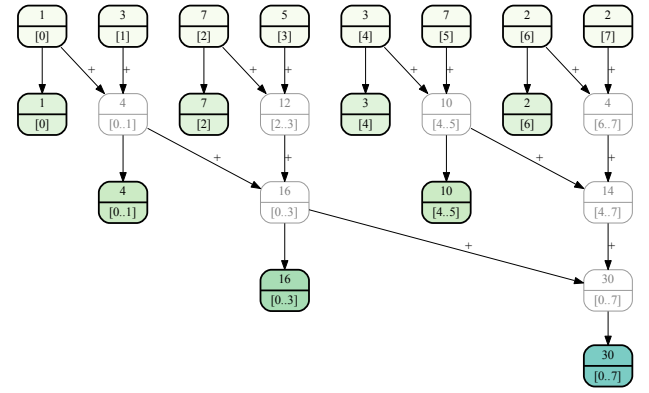


Fig. 3. Iterative construction of a 1d Fenwick Tree. Even elements are copied verbatim to the next level, whereas at odd positions, even and odd elements are added. Each node shows the data value at the top and the range of the partial sum at the bottom.

The resulting tree is a full binary tree. It is typically stored implicitly in an array whose length is equal to the input array. However, for our optimized GPU implementation we will later describe how the construction is done in parallel and the levels are stored separately for better memory efficiency.

3.1.2 Prefix Sum

To compute a prefix sum of a Fenwick Tree, the partial sums in the tree are accumulated. An example is illustrated in Fig. 4. To keep our description consistent with the pseudo code listed in Alg. 2 and the actual implementation, we illustrate the computation of the exclusive prefix sum for element $i = 7$ (which is equivalent to the inclusive prefix sum of element $i = 6$). Fig. 4 shows the traversal that starts at element $i = 6$ and traverses the tree, summing up elements x_6 , x_5 , and x_3 .

The series of indices (6,5,3 in the example of Fig. 4) that is necessary to traverse the tree is efficiently computed using bit arithmetic on the query address (7 in the example of Fig. 4). To describe the arithmetic of the traversal we define the two operations, $\text{flip}_1(x)$, that flips the least significant (l.s.) 1-bit of a binary string x , and $\text{flip}_0(x)$ that flips the least significant 0-bit of a binary string x . For example, $\text{flip}_1(10110_b) = 10100_b$ and $\text{flip}_0(10110_b) = 10111_b$. Using the flip_1 operation, we provide pseudo-code for the exclusive prefix computation in Alg. 2.

In the example of Fig. 4, k initially equals 7, first adding x_6 to the accumulation register. k equals to (00111_b) in binary form. $\text{flip}_1(00111_b)$ produces (00110_b) which sets the index to $k = 6$. The next iteration adds x_5 to the accumulation register and sets $k = 4$ by executing the bit arithmetic $\text{flip}_1(00100_b)$. In the third iteration x_3 is added to the accumulation register and k gets set to 0, which terminates the prefix sum computation.

Algorithm 2 Fenwick Tree Exclusive Prefix

```

procedure FWTPREFIX( $\mathbf{x}, k$ )                                ▷ prefix of  $x_k$ 
   $\alpha \leftarrow 0$                                        ▷ accumulation register
  while  $k > 0$  do
     $\alpha \leftarrow \alpha + x_{k-1}$                          ▷ accumulate contribution
     $k \leftarrow \text{flip}_1(k)$                              ▷ clear  $k$ 's l.s. 1-bit
  end while
  return  $\alpha$ 
end procedure

```

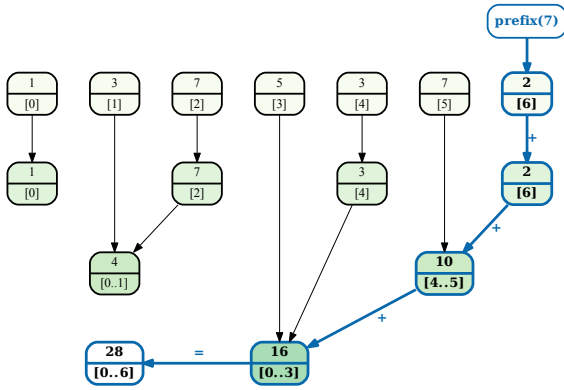



Fig. 4. Exclusive prefix computation on a 1d Fenwick Tree. Using bit arithmetics of the prefix position (in this example, $7 = 0111_b$) results in three terms to be added, ($x_6 = 2, x_5 = 10, x_3 = 16$), one for each set bit in the prefix position.

3.1.3 Point Query

A point query $q(\mathbf{x}, k)$ at index k is the difference of two adjacent exclusive prefix computations $q(\mathbf{x}, k) = p(\mathbf{x}, k+1) - p(\mathbf{x}, k)$. It is easily seen that the computation of the two exclusive prefix sums $p(\mathbf{x}, k+1)$ and $p(\mathbf{x}, k)$, share partial sums which after the subtraction cancel each other out. For point queries, the two adjacent prefix computations thus do not need to be computed entirely but only up to the partial sums that cancel each other. Fig. 5 illustrates the paths that are taken to compute a point query at index $k = 5$. The adjacent prefix sum computations both reach the node with index 3 (with value $16 = \sum_{i=0}^3 x_i$) which ends the traversal.

Algorithm 3 Fenwick Tree Point Query

```

procedure FWTVALUE( $\mathbf{x}, k$ ) ▷ value  $x_k$ 
   $l \leftarrow k + 1$  ▷ prefixes  $k, l := k + 1$ 
   $\alpha \leftarrow 0$  ▷ accumulation register
  while  $l > 0$  and  $k \neq l$  do ▷ early out for  $k = l$ 
     $\alpha \leftarrow \alpha + x_{l-1} - x_{k-1}$  ▷ difference of prefix
     $k \leftarrow \text{flip}_1(k)$  ▷ clear  $k$ 's l.s. 1-bit
     $l \leftarrow \text{flip}_1(l)$  ▷ clear  $l$ 's l.s. 1-bit
  end while
  return  $\alpha$ 
end procedure

```

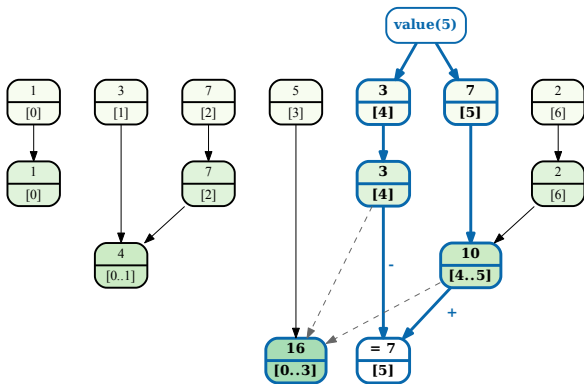


Fig. 5. A point query is a difference of adjacent prefixes. As soon as the paths of these prefixes reach the same node, the result is finalized.

At this point, the remaining contributions of the two prefix sums are equal and will cancel each other out. The omitted computation of adjacent partial sums of Fenwick trees has a profound impact on the

performance of point queries. The average point query requires look-ups of only 2 elements on average. The corresponding pseudo-code is provided in Alg. 3.

3.1.4 Updates

One of the remarkable properties of Fenwick Trees are efficient updates. Fig. 6 shows an update of element 0 from value 1 to a new value of 5. In Fig. 6 the update (+4) needs to be propagated to all elements that contain element 0 in their partial sum.

In contrast to prefix sum computations which reset the l.s. 1-bits in the query position, updates compute their position by flipping l.s. 0-bits. Algorithm 4 provides the matching pseudo-code. Note that if a differential update is performed (e.g., incrementing or decrementing), the differential update value of Δ in Alg. 4 is known and a trivial optimization is to skip the initial point query $\text{fwValue}(\mathbf{x}, k)$.

Since each entry in the Fenwick Tree is a partial *backward* sum, updates need to be propagated *forward* in the tree.

Algorithm 4 Fenwick Tree Point Update

```

procedure FWTUPDATE( $\mathbf{x}, k, y$ ) ▷  $x_k \leftarrow y$ 
   $\Delta \leftarrow y - \text{fwValue}(\mathbf{x}, k)$  ▷ differential update value
  while  $k < n$  do
     $x_k \leftarrow x_k + \Delta$  ▷ update  $x_k$ 
     $k \leftarrow \text{flip}_0(k)$  ▷ set  $k$ 's l.s. 0-bit
  end while
end procedure

```

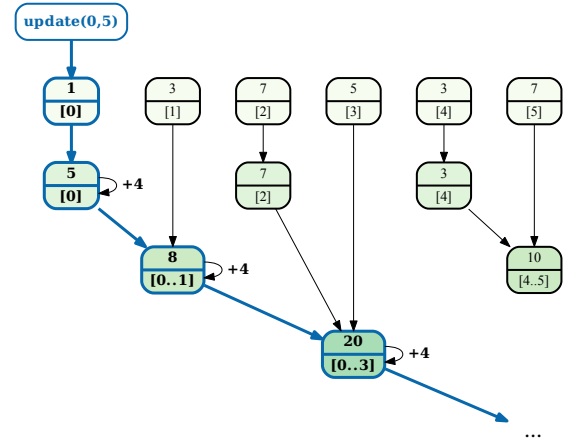


Fig. 6. Updates have to be propagated *forward*, since the Fenwick tree stores *backwards* partial sums. Depicted is an update of position 0 from an old value of 1 to a new value of 5. The difference of $5 - 1 = 4$ is added to all nodes containing position 0 in their partial sum.

3.2 Generalization

Generalization to higher dimensions is performed using binary indexed trees of binary indexed trees [18]. The easiest way to understand the concept is to consider practical storage layouts that arise if all nodes of one level are packed tightly together. Figure 7 depicts the layout for the 1d case and a Fenwick tree of 192 (bit-)elements. Level L_i stores $2^{-i-1}N$ verbatim data values, where N is the total amount of input data values.

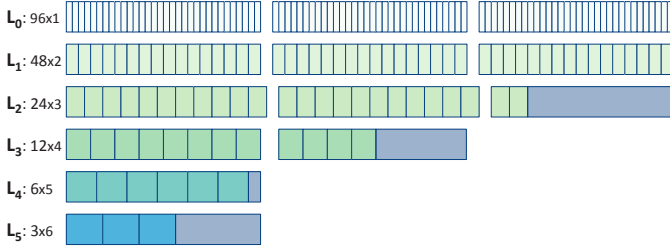


Fig. 7. 1d storage layout of a 192-element Fenwick tree. Level L_0 contains 96×1 bit, L_1 48×2 bit etc. For practical reasons, we pad elements per level tightly while aligning each level to 32 bits (gray-blue).

In contrast, for two and more dimensions, multiple level indices are used to refer to reductions along each axis. For two dimensions, the storage layout resembles a RIPMap [15] and is depicted in Figure 8. Generally, a level L_{ij} stores the sum of $(2^i N) \times (2^j N)$ data values and there are $(2^{i-1} N) \times (2^{j-1} N)$ elements in level L_{ij} . It is interesting to note that certain levels store the exact same amount of elements, which are sums of the exact same amount of data values. To formalize this, we introduce the concept of a *sum-of-levels* $\lambda(L_{ij}) := i + j$ (the same concept is also valid for higher dimensions). All levels sharing the same sum-of-levels λ obviously contain the same amount of elements and sum over the same amount of data values. In what follows, we will call such levels *similar*. One important quantity for the analysis of generalized Fenwick trees in d dimensions is the amount of such *similar* levels. From the sum-of-levels concept it follows that for each λ , the amount of similar levels can be computed using a *composition* of ℓ by d integers picked from $\{0, \dots, \lambda\}$. Formally, this composition is defined as

$$\text{comp}(\lambda, d) := \binom{\lambda + d - 1}{d - 1}, \quad (2)$$

where the offset of -1 in the binomial coefficient stems from the fact that we allow 0 as an element in the composition. Similar levels use the same shade in Figure 8.

The algorithms presented in Section 3.1 can be generalized by nesting **while** loops for each added dimension. The k^{th} such loop scans the k^{th} component of the d -dimensional address.

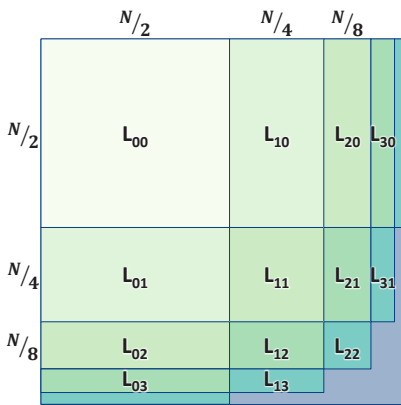


Fig. 8. 2d storage layout. Level L_{00} stores $(N/2) \times (N/2)$ single data values, L_{10} $N/2 \times N/4$ sums of 2×1 data values, and, in general, level L_{ij} stores $(2^{i-1} N) \times (2^{j-1} N)$ elements, each of which is a sum of $2^i \times 2^j$ data values.

3.3 Asymptotic Analysis

For this asymptotic analysis, we will assume an input data set of sufficiently large, finite size N , in which each element is stored using b bits.

Memory Complexity. The finest level L_0 will store $N/2$ values each at exactly b bits. Each of these values is in the range $[0, 2^b - 1]$. The next level will store $N/4$ sums of two such b -bit elements, resulting in a range of $[0, 2^{b+1} - 2]$, and so forth, with the $N/2^{\ell+1}$ elements of the ℓ^{th} level occupying a range of $[0, 2^{b+\ell} - 2^\ell]$. If fractional bits can be stored efficiently, each data element at level ℓ therefore requires $\log_2(2^{b+\ell} - 2^\ell + 1)$ bits, resulting in the following equation for the total amount of bits per element.

$$M_{\text{theo}}(b) = \sum_{\ell=0}^{\infty} \frac{\log_2(2^{b+\ell} - 2^\ell + 1)}{2^{\ell+1}} \quad (3)$$

Practical implementations may choose to not store fractional bits and instead round to the nearest full bit. This results in a storage cost of

$$M_{\text{prac}}(b) = \sum_{\ell=0}^{\infty} \frac{b + \ell}{2^{\ell+1}} = b + 1. \quad (4)$$

To the best of our knowledge, Eq. 3 has no closed form solution and has to be solved numerically.

For higher dimensions d , we use the *sum-of-levels* concept introduced in Subsection 3.2. Each sum-of-levels λ instance contains $N/2^{d+\lambda}$ data elements. Each data element stores a value in the range $[0, 2^\lambda(2^b - 1)]$. In total, there are $\text{comp}(\lambda, d)$ instances of sum-of-levels λ blocks. Combined, this amounts to

$$M_{\text{theo}}(b, d) = \sum_{\lambda=0}^{\infty} \text{comp}(\lambda, d) \frac{\log_2(2^{\lambda+b} - 2^\lambda + 1)}{2^{d+\lambda}}. \quad (5)$$

Practical implementations that round up fractional bits in Eq. 5 will achieve

$$M_{\text{prac}}(b, d) = \sum_{\lambda=0}^{\infty} \text{comp}(\lambda, d) \frac{\lambda + b + 1}{2^{d+\lambda}} = b + d \quad (6)$$

bits per element. Again, to the best of our knowledge, Eq. 5 has to be solved numerically.

Table 1 compares theoretical and practical memory consumption per element for common combinations of data dimensionality d and input bit rate b . In contrast to summed area tables, which essentially require $\mathcal{O}(\log_2^d(N))$ bits per data element, Fenwick trees correspond to substantial savings in memory requirements—a $\mathcal{O}(1)$ memory consumption is basically offset by one additional bit per dimension. Table 2 lists the relative difference in percent between practical and theoretical memory consumption per element. As can be seen, practical implementations that chose to round up fractional bit rates will lose at most about 21%.

Table 1. Bitrate as a function of data dimensionality d and input bit rate b . The first number corresponds to the theoretical bitrate $M_{\text{theo}}(b, d)$ (Eq. 5), rounded to three digits, whereas the number in parentheses corresponds to the precise bit rate of a practical implementation without fractional bit allocation $M_{\text{prac}}(b, d)$ (Eq. 6).

	$d = 1$	$d = 2$	$d = 3$	$d = 4$
$b = 1$	1.701 (2)	2.487 (3)	3.335 (4)	4.219 (5)
$b = 2$	2.867 (3)	3.776 (4)	4.713 (5)	5.659 (6)
$b = 4$	4.969 (4)	5.948 (6)	6.933 (7)	7.910 (8)
$b = 8$	8.998 (9)	9.997 (10)	10.99 (11)	11.972 (12)
$b = 16$	16.999 (17)	17.999 (18)	18.995 (19)	19.977 (20)

Run-time complexity. Using the lifting concept borrowed from wavelet theory, construction of d -dimensional Fenwick trees is in $\mathcal{O}(N^d)$. Updates and prefixes are in $\mathcal{O}(2^d \log_2^d(N))$ in the average and worst-case, as observed by [18]. Point queries, however, are in $\mathcal{O}(1)$

Table 2. Relative difference between $M_{\text{theo}}(b, d)$ and $M_{\text{prac}}(b, d)$ in percent and rounded to two digits (also known as *slack*). As can be seen, for the most common combinations of dimensionality d and input bit rate b , the slack is at most about 21%.

	$d = 1$	$d = 2$	$d = 3$	$d = 4$
$b = 1$	17.59%	20.61%	19.93%	18.50%
$b = 2$	4.62%	5.92%	6.10%	6.03%
$b = 4$	0.62%	0.87%	0.97%	1.13%
$b = 8$	0.02%	0.03%	0.06%	0.19%
$b = 16$	<0.01%	0.02%	0.02%	0.12%

on average. To see this, we would like to refer the reader back to Algorithm 3 and Figure 5. The 1d algorithm essentially scans the address from the least significant bit to the most significant bit. It terminates at the least significant 0-bit. For data in which each data dimension is a power-of-two 2^L , the L least-significant bits of a random address are uniformly distributed. Therefore, there is a chance of 0.5 for the algorithm to terminate with a single memory fetch, a chance of 0.5^2 to terminate after two memory fetches and so forth. In the limit, we therefore obtain

$$R_{\text{point}} = \lim_{L \rightarrow \infty} \sum_{i=0}^L \frac{i+1}{2^{i+1}} = 2 \in \mathcal{O}(1). \quad (7)$$

For non-power-of-two data dimensions, a 0 bit is generally more likely to occur in a valid address, maintaining the $\mathcal{O}(1)$ complexity of point queries even for those domains. For higher dimensions d , the address vector is scanned in the same fashion for each dimension, resulting in a recursion of the form

$$R_{\text{point}}(1) = 2$$

$$R_{\text{point}}(d) = \lim_{L_d \rightarrow \infty} \sum_{i=0}^{L_d} \left[\frac{i+1}{2^{i+1}} R_{\text{point}}(d-1) \right] = 2^d \in \mathcal{O}(2^d), \quad (8)$$

which is constant in the amount of data N .

The worst-case complexity is still in $\mathcal{O}(2^d \log_2^d(N))$, but as we will show in Section 6, constant-time point-queries can be observed in practice even under the GPU's massively parallel computation paradigm in which threads executing faster queries may have to wait on those performing slower queries.

Since fast point queries can be fused with the logarithmic prefix computations, Fenwick trees are well-suited for unconstrained access on binary occupancy data—empty cells or bricks can be determined in constant time, allowing for, e.g., fast brick-based empty space skipping, whereas retrieving the data for a *occupied* cell or brick can still be achieved in logarithmic time.

4 PROPERTIES OF THE DATA STRUCTURE

In this section, we will discuss the most important properties of this data structure.

Fill-Agnostic. We are mostly concerned with data for which $K \ll N^d$ since in these cases the data can be stored more efficiently than in a dense array. However, we also show that our data structure is *fill-agnostic* and provides good results for any fill density ρ . The property *fill-agnostic* is beneficial for practical applications since it makes the memory requirements for the spatial indexing data structure predictable. Other index data structures (most noteworthy spatial hashing) are typically not *fill-agnostic*. Although it is more likely for most data structures to become small in memory for small ρ , it is not guaranteed at all. Many data structures rely on an uneven distribution of data values (dense clusters in an otherwise mostly empty space) to successfully scale. Construction of these data structures on the GPU therefore might fail for data sets that happen to have unsuitable properties. One of the biggest problems with these data structures is the unpredictable

memory consumption. The only way to find out if a non-fill-agnostic data structure can be constructed for a specific data set is to actually attempt the construction.

A *fill-agnostic* data structure can guarantee its memory requirements beforehand and the success of the construction is therefore predictable.

Unconstrained Access. Constrained access means that the result of a point query at an empty position is undefined.

In contrast unconstrained access means that point queries at arbitrary positions return the correct result. In general any data structure with constrained access can trivially be extended by adding 1 bit of occupancy data per index. However, this results in additional storage and a runtime overhead. Our data structure allows unconstrained access without additional occupancy storage.

Constant Number of Elements. Our data structure is constant in the number of elements, which means that the resulting tree indexes the same number of elements as the input sequence had non-zero elements.

Compactable. Since each level of our data structure results in a predictable number of bits the bit stream can be compacted. In contrast data structures with unpredictable sizes of elements or bit count cannot be further compacted, which sometimes results in *slack* or waste of storage.

Payload-Agnostic. The data structure does (like most spatial indexing data structures) support any kind of payload. One of the advantages of a payload-agnostic data structure is its versatility and therefore the reduced implementation effort when multiple kinds of data types need to be supported.

5 GPU-BASED IMPLEMENTATION

We have implemented our data structure on the GPU using OpenGL Compute Shaders using the 4.50 core profile.

Construction. The key observation is that sub-trees of the Fenwick tree can be built in parallel, resulting in a truncated tree. We can therefore generate an arbitrary amount of the levels of the tree in parallel, synchronize, and continue building the remainder of the tree in the next pass. This is important when 1-bit occupancy data (block or cell *occupy* or *empty*) are concerned, since we would like the output bit stream to be packed as tightly as possible to keep the memory footprint small. Any parallel implementation has thus to ensure that concurrent threads access only mutually exclusive parts of the stream, or they have to resort to expensive synchronization operations.

Considering the aforementioned constraints, we chose to split the total work in two asymmetric compute shaders, which we call *reduce-by-256* and *reduce-by-8*. The first one takes tightly packed bits in an unsigned integer Shader Storage Buffer Object (SSBO) as input, and it generates the first eight levels of the tree as an output. Starting with 1-bit values in the nodes at level 0 of the tree, entries at level 1 range from $0 \dots 2$, at level 2 from $0 \dots 4$, and, generally, at level L from $0 \dots 2^L$. While these values could be packed tighter using fractional bits, for practical reasons we allocate $(L+1)$ bits to each of them and store them tightly packed as $(L+1)$ -bit values into an output SSBO. In contrast, each level is aligned to the next unsigned integer. Each invocation of this shader consumes 4096 bits of the input stream (128 bytes), with 32 invocations forming a local work group. To store intermediate results, each local work group uses 16KB = 32×512 bytes of shared local memory, which provides each invocation with 512 bytes (128 unsigned integers) for exclusive use. Each invocation thus outputs $L_0 : 2048 \times 1\text{bit}$, $L_1 : 1024 \times 2\text{bits}$, \dots , $L_7 : 16 \times 8\text{bits}$ output bits, which are all 32-bit aligned. The unprocessed result is stored at a 32-bit alignment in a *carry* SSBO comprising $1/8^{\text{th}}$ of the

size of the original ($N/256 \times 32\text{bits}$, where N is the amount of input bits) and it is used as input for subsequent passes.

The *reduce-by-8* shader reads, per invocation, 256 unsigned integers from the carry buffer and outputs tightly packed k -bit values, where k is the current level of the tree plus one. It emits 128, 64, 32 values, and, therefore, maintains 32-bit alignment as well. Again, 128 unsigned integers of shared local memory are used by each invocation, resulting in 16KB for each local work group. In this local memory, values are stored as 32-bit aligned unsigned integers to avoid the costly bit arithmetics of the *reduce-by-256* shader. This shader is repeated until the full tree is constructed. SSBO barriers are used between any of the passes to ensure coherent state of the input and output buffers.

The particular choice of reducing the data 256:1 in a first pass and then continuing with 8:1 reductions are motivated as follows. Starting with the full-size occupancy volume, memory footprint considerations are of utmost importance. Using 16K of local memory (and 48K of local memory as cache), the *reduce-by-256* pass starts with as much work as fits into the limited local memory and performs as much reduction as possible without the need to either synchronize adjacent workgroups or pad to 32bit alignment. The *reduce-by-8* passes then use a 32bit aligned data values, and we reduce by as many levels as we can without using $32\times$ parallelism in a workgroup using the limited local memory. Future architectures may choose the size of these reduction passes differently, but we found that our approach benefits of the larger cache, as opposed to 48K local memory and 16K cache.

Computing the memory optimal instance. We search for the most memory efficient instance of our data structure by computing the memory requirements for different brick sizes. We first construct an binary occupancy volume storing a value of 1 if a voxel is occupied and a value of 0 if a voxel is empty. We then brick this volume using a set of candidate brick sizes by performing a reduction with a logical OR operator inside each brick. The brick size candidates are power-of-two numbers, for which 1 voxel is used as padding in either dimension. For instance a brick size of 4^3 results in bricks that carry a payload of 3^3 voxels. The memory requirement for the Fenwick tree follows trivially from the brick size. The size of the payload (i.e., the contents of the non-empty bricks) is computed separately for each brick size using the reduced occupancy volume. Generally, smaller brick sizes lead to a better approximation of the empty space and allow for the rejection of more bricks. However, the index overhead gets larger for smaller bricks, since more nodes of the Fenwick tree are required. Further, the overhead for the padding of bricks becomes larger relative to small brick sizes.

Reconstruction. The GPU based implementation of the reconstruction and the prefix sum make use of hardware accelerated bit-operations. The $\text{flip}_0(\cdot)$ and $\text{flip}_1(\cdot)$ operation are implemented using the optimized GLSL function *findLSB()*.

6 RESULTS

We tested our implementation on a workstation equipped with a dual Xeon E5-2680 clocked at 2.70GHz and running Windows 7. The system was equipped with 64GB of RAM and a single NVIDIA GeForce Titan Black.

Figure 9 provides a stacked bar plot of the time in ms taken to construct full Fenwick trees of various sizes. We show the time taken by the first *reduce-by-256* pass, as well as subsequent *reduce-by-8* passes. The *reduce-by-256* pass processes input bits in a packed alignment, thereby minimizing the memory footprint. While it is reasonably optimized for speed, it performs involved bit arithmetic to compute the first 8 levels of the tree. It is therefore slower than the following *reduce-by-8* pass, which operates on k -bit values aligned to 32bit unsigned integers, but maintaining compact storage has high priority in this stage. The expense of the luxury of 32 bit alignment is that we re-

quire a temporary *carry* buffer comprising $1/8^{\text{th}}$ the size of the input buffer. The input sizes were chosen to be power-of-two fractions and multiples of 11.25Mbit, which corresponds to the amount of work that the 2,880 cores of our GPU can perform in parallel. For this experiment, our input is an array consisting of 1bit occupancy values.

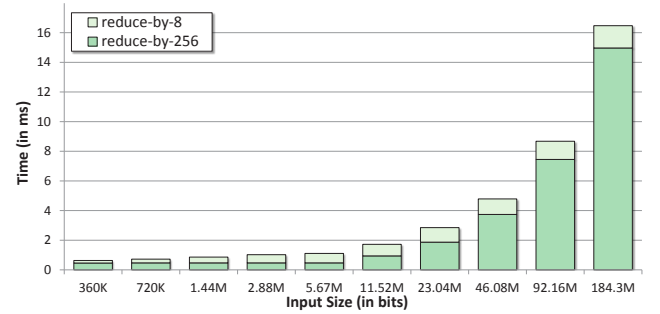


Fig. 9. Time to build full Fenwick trees of different input sizes on the GPU. We show both the time in ms for the first *reduce-by-256* pass (dark) and the subsequent *reduce-by-8* passes stacked on top.

We also measured the time required to reconstruct the full binary input occupancy volume from Fenwick trees of various sizes. For this, we chose a size of 256 invocations per local work group. Figure 10 provides a plot of the amount of point queries that can be performed per second as a function of the input data size. The average-case asymptotic cost of each point query is in $\mathcal{O}(1)$, but the worst-case complexity is in $\mathcal{O}(\log_2 N)$, where N is the input data size. Despite the GPU's computation model, which may force invocations which terminate early to wait on invocations with higher computational load, the benefits of our average $\mathcal{O}(1)$ point query complexity are obvious for reasonably large input data sets, for which we achieve up to 22.4 billion point queries per second. Also shown is a naïve $\mathcal{O}(\log_2(N))$ implementation that computes the difference of two prefixes. Its performance decreases as expected when data size increases—1.38 billion point queries per second for 2.88Mbit of data ($\log_2(N) = 22$) and only 1.18 billion per second for 184.3Mbit of data ($\log_2(N) = 28$).

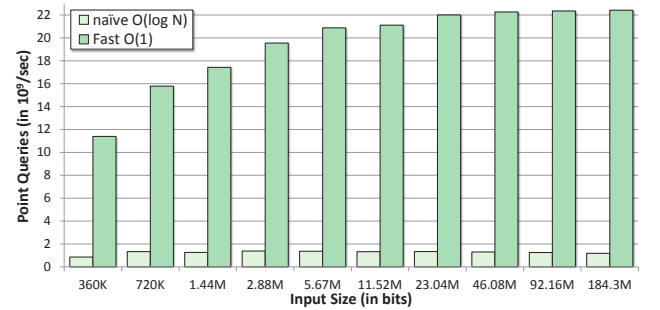


Fig. 10. Time to reconstruct a full binary occupancy volume from the Fenwick tree on the GPU. We show billion point queries over the input data size. Dark green: our $\mathcal{O}(1)$ point query algorithm. Light green: naïve $\mathcal{O}(\log_2(N))$.

Finally, we evaluate the performance of a bricked volume renderer equipped with our data structure. We first classify noise in the volume by thresholding. Then, we build a full resolution, binary occupancy volume and subsample this occupancy volume to match a sequence of pre-selected brick sizes. Subsampling in this context is performed using a logical OR operation. Our renderer uses a payload of $2^k - 1$ for each brick, plus one voxel padding to facilitate proper trilinear interpolation. Each brick thus comprises $2^k \times 2^k \times 2^k$ voxels. We compute the storage requirements for each brick choice $k = 0, \dots, 8$, and select the brick size minimizing our storage requirements. Each brick is then stored in a 3D texture atlas. A GPU-based Fenwick

tree is computed from the occupancy volume corresponding to the optimal brick size. After uploading the data thus obtained to the GPU, we measured times to render a wide range of volumetric data sets, including medical (CT and MRI) and industrial CT data sets of various sizes. Table 3 summarizes the data sets used in this paper and our findings. For each data set, we measured the time to render (a) Phong-lit isosurfaces and (b) semi-transparent structures from our bricked representation. For both cases, we rendered to a 1024×768 view port, and we used a raymarching step size of 0.25 voxel diagonals. For semi-transparent renderings, we did not use any early ray termination, but brick-based empty-space skipping was used. We then repeated the experiment using a naïve volume renderer with the same parameters. As can be seen from Table 3, the compression ratio for some data sets can be significant, ranging from 1.39:1 up to 40.25:1. These figures include the slack introduced by the brick padding. While using our brick-based volume renderer that stores brick indices in a Fenwick tree comes at a cost, we would like to note that any bricked volume renderer necessarily has a performance overhead over naïve implementations, including the need to perform exact ray-brick intersections and retrieving the bricks from the texture atlas. It is generally accepted that this cost is more than compensated for by the benefits, such as the ability to scale to larger volume sizes.

For isosurface extraction, our method was faster than a naïve approach on the majority of the data sets. The reason is that, in these data sets, bricks are fairly large and the isosurface is spatially coherent. The overhead of our data structure is therefore more than compensated for by the ability to efficiently skip empty regions in the data. For semi-transparent rendering, the traversal cost of our data structure becomes more pronounced, but we still achieve a better performance than a naïve implementation for half of the data sets in our repository. We would like to particularly note the case of the Vessel data set, which due to its massive amount of small and intricate structures poses a challenge for less memory efficient bricking data structures. For this data set, we observe very good performance for semi-transparent rendering when compared to naïve approaches. However, isosurface reconstruction is a lot worse. We believe this is caused by severe loss of ray coherence, which results in GPU threads of the same workgroup performing drastically different work loads.

6.1 Future Work

GPU-based Fenwick trees are a powerful and versatile data structure, in part due to the hitherto undocumented constant run-time of the point query operation and their close relation to the (unnormalized) Haar wavelet basis, which seems to have gone unnoticed in the past. In this work, we exploited this relation by constructing Fenwick trees using a lifting scheme, which avoids the majority of the implementation hassle otherwise arising from non-power-of-two dimensions.

In the future, we would like to generalize this—and other—related data structures by using a simple algebraic framework. In particular, we can consider pre-computation of summed area tables, Fenwick trees, orthogonal wavelets and potentially other data structures in terms of a linear transform

$$\mathbf{y} := \mathcal{A} \mathbf{x}, \quad (9)$$

where $\mathbf{x} \in \mathbb{R}^n$ is an input data vector, and $\mathcal{A} \in \mathbb{R}^{n \times n}$ is a non-singular matrix. Computing a prefix then becomes equivalent to evaluating the following equation during run-time.

$$\text{prefix}(\mathbf{x}, k) = \underbrace{(1, \dots, 1, 0, \dots, 0)}_{k \times 1} \mathcal{A}^{-1} \mathbf{y} := \mathbf{p}_k \mathcal{A}^{-1} \mathbf{y}. \quad (10)$$

Clearly, for summed area tables $\mathbf{p}_k \mathcal{A}^{-1} = \hat{\mathbf{e}}_k$ (the k^{th} standard unit vector), offering optimal asymptotic prefix computation speed, whereas for Fenwick trees, each column of \mathcal{A}^{-1} has a logarithmic amount of non-zero entries on average. Using this framework, we will explore operations more complex than prefixes that are of practical

relevance in our community. In particular, if an orthogonal, discrete wavelet basis with m vanishing moments is used for the reconstruction matrix \mathcal{A}^{-1} , any vector \mathbf{v} whose components sample a polynomial of degree m or less should have an efficient run-time evaluation $\mathbf{v} \cdot \mathcal{A}^{-1}$. This would allow for generalizations of summed area tables. Summed area tables allow for integration over rectangular domains, whereas, using wavelets, it would become possible to integrate the product of data and a polynomial kernel over a rectangular domain. This could have applications in texture filtering and selective blurring as it arises, e.g., in rendering depth of field.

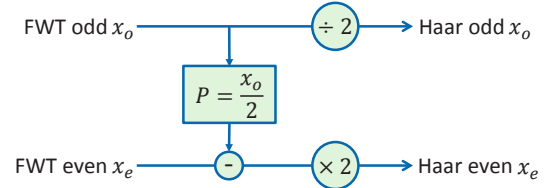


Fig. 11. Transformation between Fenwick tree coefficients and a (unnormalized) Haar basis.

Fenwick trees can also be transformed into a (unnormalized) Haar wavelet basis by the simple prediction and scaling depicted in Fig. 11. This could enable even more efficient storage on disk by performing run-length encoding of the sparse wavelet-domain representation of the input data.

Another venue for future research is to analyze the application of Fenwick trees to compute prefix scans of floating point data, which were beyond the scope of this paper due to the intricacies rounding errors introduce into the analysis.

Finally, we would like to note that our brick-sparse volume representation could be easily altered to allow for construction of data being streamed onto the GPU, thus avoiding the need to store the full occupancy volume and/or data volume on the GPU at any time.

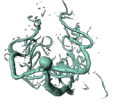
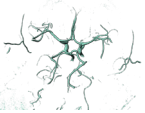
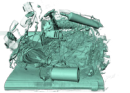





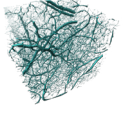
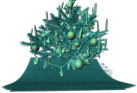
7 CONCLUSION

In this paper we presented a versatile GPU data structure based on Fenwick trees. We have provided an extensive theoretical analysis of the proposed data structure, combined with notes on its practical implementation and a performance evaluation. We have furthermore outlined the proposed data structures use for bricked sparse representation of volumetric data sets, which we believe has the potential to have high impact.

ACKNOWLEDGMENTS

The authors' work is funded by the Visual Computing Center (VCC) at King Abdullah University of Science and Technology (KAUST). Volume data sets were obtained from VolVis.org, TU Wien, and DigiMorph. The vessel data set was provided by John Keyser, TAMU.

Table 3. Data sets used in this paper. For each data set, the first column provides a representative picture plus the log-histogram. We then provide name, resolution, and range R for each data set. Next listed is the threshold τ used to classify noise in the data (noise is colored green in the histogram, and is defined as data values $x \leq \tau$), followed by the fill ratio ρ of the input volume (top) and the fill after bricking (bottom). Next are the optimal block size for reducing the size of the volume in GPU memory as much as possible, followed by the size of the Fenwick tree and the overall compression ratio achieved by our approach. We finally list average rendering performance in milliseconds for our approach and a naïve unbricked volume renderer, both rendering to a 1024×768 view port and both using a step size of 0.25 voxel diagonals (top: extraction of a single, Phong-lit isosurface at τ , bottom: semi-transparent rendering without opacity-based early ray termination). Note that the compression ratio includes overhead due to a 1-voxel padding for each block and that exact brick-intersections are included in the timings for our method.

Dataset	Description	Fenwick Tree parameters					Rendering Performance	
		τ	ρ	bricks	size	ratio	with FWT	w/o FWT
	Aneurysm $512 \times 512 \times 512$ $R = [0 \dots 3000]$	704	0.43%	3^3	1.19MB	40.25:1	31.05ms	25.15ms
			↓ 0.85%				30.12ms	21.83ms
	Angiography $416 \times 512 \times 112$ $R = [0 \dots 685]$	160	0.49%	3^3	220.5KB	16.73:1	8.83ms	13.98ms
			↓ 2.27%				9.74ms	10.33ms
	Backpack $416 \times 512 \times 373$ $R = [0 \dots 4071]$	50	21.7%	7^3	72.23KB	1.39:1	5.09ms	7.18ms
			↓ 46.4%				18.18ms	9.73ms
	Bonsai $256 \times 256 \times 256$ $R = [0 \dots 255]$	37	10.4%	7^3	12.39KB	3.30:1	4.09ms	2.79ms
			↓ 19.4%				4.52ms	2.71ms
	King Snake $1024 \times 1024 \times 795$ $R = [0 \dots 65535]$	10150	43.7%	31^3	7.54KB	1.78:1	10.03ms	15.11ms
			↓ 47.5%				15.11ms	26.46ms
	Pawpawsaurus Campb. $958 \times 646 \times 1088$ $R = [0 \dots 65535]$	24000	10.6%	15^3	50.21KB	5.30:1	9.05ms	31.29ms
			↓ 15.1%				25.07ms	35.07ms
	Present $492 \times 492 \times 442$ $R = [0 \dots 4095]$	280	7.56%	7^3	78.80KB	3.75:1	6.74ms	7.17ms
			↓ 17.3%				14.87ms	10.18ms
	Stag Beetle $832 \times 832 \times 494$ $R = [0 \dots 4095]$	0	4.06%	7^3	245.5K	13.19:1	11.88ms	31.08ms
			↓ 5.01%				14.38ms	33.33ms
	Vessel $1024 \times 1024 \times 1024$ $R = [0 \dots 255]$	30	2.50%	3^3	9.54MB	4.72:1	970.86ms	943.39ms
			↓ 8.69%				77.58ms	387.60ms
	Xmas Tree $512 \times 499 \times 512$ $R = [0 \dots 4095]$	110	1.67%	3^3	1.16MB	9.30:1	21.10ms	27.24ms
			↓ 4.31%				27.38ms	24.58ms

REFERENCES

- [1] D. Benson and J. Davis. Octree textures. *ACM Transactions on Graphics*, 21(3):785–790, July 2002.
- [2] G. E. Brelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.
- [3] G. E. Brelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, 1990.
- [4] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proc. ACM Symp. Interactive 3D Graphics (I3D)*, pages 15–22, 2009.
- [5] F. C. Crow. Summed-area tables for texture mapping. *ACM Siggraph 1984, Computer Graphics*, 18(3):207–212, 1984.
- [6] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Journal on Fourier Analysis and Applications*, 4(3):247–269, 1998.
- [7] C. Dyken, G. Ziegler, C. Theobaldt, and H.-P. Seidel. High-speed marching cubes using histopyramids. *Computer Graphics Forum*, 27(8):2028–2039, 2008.
- [8] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 3(24):327–336, 1994.
- [9] I. García, S. Lefebvre, S. Hornus, and A. Lasram. Coherent parallel hashing. *ACM ToG/Siggraph Asia*, 30(6):A161, 2011.
- [10] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison-Wesley Professional, 2007.
- [11] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24(3):547–555, 2005.
- [12] M. Kraus and T. Ertl. Adaptive texture maps. In *Proc. ACM Siggraph/Eurographics Conf. on Graphics Hardware*, pages 7–15, 2002.
- [13] M. Labschütz, S. Bruckner, E. Gröller, M. Hadwiger, and P. Rautek. JiT-Tree: a just-in-time compiled sparse GPU volume data structure. *IEEE TVCG/Vis Week*, 22(1):1025–1034, 2016.
- [14] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, Oct. 1980.
- [15] R. Larson and M. Shah. Method for generating addresses to textured graphics primitives stored in RIP maps. US Patent 5,222,205, 06/22/1993.
- [16] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM ToG/Siggraph*, 25(3):579–588, 2006.
- [17] S. Lefebvre, S. Hornus, and F. Neyre. Octree textures on the gpu. In M. Pharr, editor, *GPU Gems 2*, chapter 37, pages 595–613. Addison-Wesley Professional, 2005.
- [18] P. Mishra. A new algorithm for updating and querying sub-arrays of multidimensional arrays. ArXiv:1311.6093 [cs.DS], 2013.
- [19] S. Sengupta, A. E. Lefohn, and J. D. Owens. A work-efficient prefix sum algorithm. In *Proc. Workshop on Edge Computing Using New Commodity Architectures*, pages D–26–27, 2006.
- [20] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1998.
- [21] G. Ziegler, A. Tevs, C. Theobaldt, and H.-P. Seidel. On-the-fly point clouds through histogram pyramids. In *Proc. Vision Modeling and Visualization*, pages 137–144, 2006.