Eurographics Conference on Visualization (EuroVis) 2021
R. Borgo, G. E. Marai, and T. von Landesberger
(Guest Editors)

Volume 40 (2021), Number 3

Parameterized Splitting of Summed Volume Tables

Christian Reinbold and Rüdiger Westermann

Computer Graphics & Visualization Group, Technische Universität München, Garching, Germany

Abstract

Summed Volume Tables (SVTs) allow one to compute integrals over the data values in any cubical area of a three-dimensional orthogonal grid in constant time, and they are especially interesting for building spatial search structures for sparse volumes. However, SVTs become extremely memory consuming due to the large values they need to store; for a dataset of n values an SVT requires $\mathcal{O}(n \log n)$ bits. The 3D Fenwick tree allows recovering the integral values in $\mathcal{O}(\log^3 n)$ time, at a memory consumption of $\mathcal{O}(n)$ bits. We propose an algorithm that generates SVT representations that can flexibly trade speed for memory: From similar characteristics as SVTs, over equal memory consumption as 3D Fenwick trees at significantly lower computational complexity, to even further reduced memory consumption at the cost of raising computational complexity. For a $641 \times 9601 \times 9601$ binary dataset, the algorithm can generate an SVT representation that requires 27.0GB and $46 \cdot 8$ data fetch operations to retrieve an integral value, compared to 27.5GB and $1521 \cdot 8$ fetches by 3D Fenwick trees, a decrease in fetches of 97%. A full SVT requires 247.6GB and 8 fetches per integral value. We present a novel hierarchical approach to compute and store intermediate prefix sums of SVTs, so that any prescribed memory consumption between $\mathcal{O}(n)$ bits and $\mathcal{O}(n \log n)$ bits is achieved. We evaluate the performance of the proposed algorithm in a number of examples considering large volume data, and we perform comparisons to existing alternatives.

CCS Concepts

• ***Information systems*** → ***Data structures;*** • ***Human-centered computing*** → *Scientific visualization;*

1. Introduction

Summed Area Tables (SATs) are a versatile data structure which has initially been introduced to enable high-quality mipmapping [Cro84]. SATs store the integrals over the data values in quadratic areas of a two-dimensional orthogonal grid that start at the grid's origin. The entries in a SAT can be considered prefix sums, as they are computed via column- and row-wise one-dimensional prefix sums. With four values from a SAT the integral over any quadratic region can be obtained in constant time. The three-dimensional (3D) variant of SATs is termed Summed Volume Tables (SVTs). They are of special interest in visualization, since they can be used to efficiently build adaptive spatial search structures for sparse volumes. In particular, construction methods for kD-trees and Bounding Volume Hierarchies (BVHs) [VMD08, HHS06] can exploit SVTs to efficiently find the planes in 3D space where the space should be subdivided.

Fig. 1 shows a temperature snapshot in Rayleigh-Bénard convection flow of size $641 \times 9691 \times 9601$. To efficiently render this dataset via direct volume rendering algorithms, some form of adaptive spatial subdivision needs to be used to effectively skip empty space. However, the SVT from which such an acceleration structure can be computed requires 247.6GB of memory, so that only on computers with large memory resources all data can be stored in main memory. While the input field is only of size $\mathcal{O}(n)$, the memory consumption of a SVT is of $\mathcal{O}(n \log n)$.

Alternative SVT representations such as 3D Fenwick trees [Fen94, Mis13, SR17] offer a memory-efficient intermediate data structure from which an adaptive space partition can be constructed. 3D Fenwick trees have a memory consumption of $\mathcal{O}(n)$ bits, yet recovering the integral values requires a number of $\mathcal{O}(\log^3 n)$ data fetch operations. For the example given in Fig. 1, a 3D Fenwick tree requires only 27.5GB of memory, but to obtain an integral value for a given volume $1512 \cdot 8$ fetches need to be performed.

For labelled datasets, SVTs can be used as an alternative to hierarchical label representations like the mixture-graph [ATAS21], to efficiently determine the number of labels contained in a selected sub-volume. Furthermore, SVTs can effectively support a statistical analysis of the data values in arbitrary spatial and temporal sub-domains. As another application of SVTs, we briefly sketch meteorological data analysis in Sec. 7. This includes the time- or memory-efficient computation of moving averages over selected sub-regions and time intervals.

1.1. Contribution

We propose an algorithm to generate SVT representations that can flexibly trade speed for memory. These representations build upon

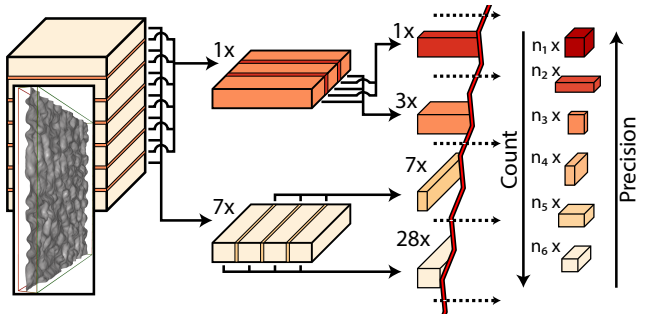

Figure 1: *Schematic operation principle of splitting SVTs. A 3D array is hierarchically split into multiple high to low precision arrays (decreasing color saturation indicates decreasing precision), to obtain a data structure from which sums over axis-aligned subarrays can be computed. For the* $641 \times 9601 \times 9601$ *input volume obtained by a supercomputer simulation of a Rayleigh–Bénard convection, the SVT requires 247.6GB. Our algorithm generates SVT representations at 27.0GB or 71.2GB, requiring respectively 46 or 8 data fetch operations per prefix sum.*

a recursive uni-axial partitioning of the domain and corresponding partial prefix sums, in combination with a hierarchical representation that progressively encodes this information. Since partial prefix sums require less bits to encode their values, the overall memory consumption can be controlled by the number and position of the performed domain splitting operations. By using different partitioning strategies, any prescribed memory consumption between $\mathcal{O}(n)$ bits and $\mathcal{O}(n \log n)$ bits can be achieved.

In principle, the algorithm proceeds in two phases: Firstly, for every possible SVT representation of a given volume an abstract *parameter tree* is constructed. This tree encodes the uni-axial split operations in a hierarchical manner, and it allows estimating both the memory consumption of the resulting SVT representation and the required data fetch operations for computing an integral value. Secondly, the tree is translated into the concrete SVT representation, by traversing the tree and performing the encoded operations.

To find a SVT representation according to a prescribed memory consumption or number of fetches, we propose a heuristic that generates a parameter tree which adheres to a given resource budget. This heuristic provides a close-to-optimal parameter tree for arbitrary budgets, over the entire spectrum ranging from memory-efficient yet compute-intense SVT representations to standard memory-exhaustive SVT representations with constant reconstruction time.

Our algorithm generates SVT representations with equal memory consumption as 3D Fenwick trees at significantly lower computational complexity. For the example in Fig. 1, we can construct a SVT representation that requires 27.0GB but requires only as few as $46 \cdot 8$ data fetch operations to retrieve an integral value, a decrease in data fetch operations of 97% compared to Fenwick trees. Our specific contributions are

- an abstract parameter tree representation that translates directly into a SVT representation,
- a capacity estimator for the memory and compute requirements of a given parameter tree,
- a heuristic that automatically provides a parameter tree that matches a prescribed capacity.

We analyze our proposed approach with respect to memory consumption and data fetch operations, and compare the results to those of alternative SVT representations. By using differently sized datasets, we demonstrate lower capacity requirements and improved scalability of our approach compared to others.

The paper is structured as follows: We first discuss approaches related to ours in the light of memory consumption and computational issues. After a brief introduction to the concept of SATs, we introduce the versatile data structure our approach builds upon. We demonstrate in particular the parameterization of this data structure to enable trading memory consumption for computational access efficiency. In Sec. 6, we then describe how to realize a concrete SVT representation that adheres to a user defined performance or memory budget. We evaluate our design choices and compare the obtained representations to alternative approaches in Sec. 7. We conclude our work with ideas for future work.

2. Related Work

SATs have been introduced by Crow [Cro84], as a data structure to quickly obtain integral values over arbitrary rectangular regions in 2D data arrays. Since then, SATs have found use in many computer vision and signal processing tasks such as object detection [BTVG06, VJ04, GGB06, Por05], block matching [FLML14], optical character recognition [SKB08] and region filtering [HPD08, Hec86, BSB10].

In computer graphics, SVTs are used to realize gloss effects [HSC*05], and in particular to accelerate the creation of spatial search structures for sparse scene or data representations. Havran et al. [HHS06] build a BVH / SKD-Tree hybrid acceleration structure for mesh data by discretizing the 3D domain and finding kd-splits in expected $\mathcal{O}(\log \log n)$. A SVT over the discretized domain is then used to evaluate the split cost function in constant time. Similarly, Vidal et al. [VMD08] propose to use SVTs to speed up cost function evaluations in a BVH construction process for voxelized volume datasets. In their work, the cost function requires the computation of bounding volumes over binary occupancy data. By running binary search on a SVT, this task can be solved in $\mathcal{O}(\log n)$ instead of $\mathcal{O}(n^3)$, where n is the side length of the volume. Ganter & Manzke [GM19] propose to use SVTs to cluster bounding volumes of small size before assembling them bottom-up into a BVH for Direct Volume Rendering (DVR). SVTs also allow to compute statistical quantities for arbitrarily large axis-aligned regions in constant time [PSCL12]. Thus, they facilitate interactive exploratory tasks in large scale volume datasets. The major drawback of SVTs is their memory consumption. Since prefix sums may span up to n elements, where n is the number of entries in a d-dimensional array, SVT entries require up to $\mathcal{O}(\log n)$ bits precision. This yields a total memory consumption of $\mathcal{O}(n \log n)$, where the original array is only of size $\mathcal{O}(n)$.

In Computer Graphics, classical Mip Mapping [Wil83] (or Rip Mapping in the anisotropic case) has been used for decades to ap-

proximately compute partial means (or equivalently sums) of textures in constant time and $\mathcal{O}(n)$ memory. Partial means of boxes with power of two side length are precomputed and then interpolated to approximate boxes with arbitrary side length. Belt [Bel08] proposes to apply rounding by value truncation to the input array before computing its corresponding SVT. By reducing the precision of the input, the SVT requires less bits of precision as well. To compensate for rounding error accumulation, the rounding routine is improved by considering introduced rounding errors of neighboring SVT entries. Clearly, this scheme cannot be used to reduce memory requirements for arrays of binary data. Although approximate schemes may suffice for imaging or computer vision tasks where small errors usually are compensated for, they are inappropriate in situations where hard guarantees are required (e.g. the support of a BVH has to cover all non-empty regions).

Exact techniques such as computation through the overflow [Bel08] or the blocking approach by Zellmann et al. [ZSL18] enforce a maximal precision bound per SVT entry and, thus, avoid the logarithmic increase in memory. The former approach simply drops all except for ℓ least significant bits and hence stores prefix sums modulo 2^ℓ. As long as queried boxes are small enough such that partial sums greater or equal than 2^ℓ are omitted, this method is exact. This approach excels in filtering tasks with kernels of prescribed box size. However, it is not applicable in situations where box sizes are either large or not known in advance.

Zellmann et al., on the other hand, brick the input array into 32^3 bricks and compute a SVT for each brick separately, hence they dub their method *Partial SVT*. As each prefix sum cannot sum over more than 32^3 elements, the amount of additional bits per entry is limited to 15 bits. However, when computing sums along boxes which do not fit into a single 32^3 brick, one value has to be fetched from memory for each brick intersecting the queried box. Since there still are $\mathcal{O}(n/(32^3)) = \mathcal{O}(n)$ many bricks, this approach scales equally poorly as summing up all values by iterating over the input array directly. More generally, any attempt to store intermediate sums with a fixed precision PREC scales poorly. In order to recover the "largest" possible sum of all array elements, one would have to add up at least $\mathcal{O}(n)/2^{\text{PREC}}$ factors of maximal size 2^{PREC}. To circumvent this issue, the authors propose to build a hierarchical representation of partial SVTs where the largest entries of each partial SVT form a new array that again is bricked and summed up partially. However, the authors admit that all bricks that overlap only partially with the queried box have to be processed at their current hierarchy level. Hence, the number of touched bricks reduces to the size of the box boundary. In the best case of cubes as boxes, the complexity still is $\mathcal{O}(n^{2/3})$—which is impractical for $n \geq 1024^3$.

Ehsan et al. [ECRMM15] propose a technique that allows to compute arbitrary prefix sums by fetching a constant number of four values only through replacing each third row and column of a 2D array by their corresponding high precision SAT entries. By either adding an array entry to preceding SAT entries, or subtracting it from subsequent ones, the full SAT can be recovered. Their approach directly generalizes to 3D, requiring eight values instead. As a consequence, they only have to store 19 out of 27 SVT entries in high $\mathcal{O}(\log n)$ precision, reducing memory consumption by up to 30%. However, total memory consumption still is in $\mathcal{O}(n \log n)$ and does not scale well.

3D Fenwick Trees as introduced by Mishra [Mis13] have beneficial properties with regard to both memory consumption and access. As shown by Schneider & Rautek [SR17], they require $\mathcal{O}(n)$ memory while allowing to compute prefix sums by summing up $\mathcal{O}(\log^3 n)$ values. In the 1D case [Fen94] this is achieved by recursively processing subsequent pairs of numbers such that the first number is stored verbatim; and the second one is summed up with the first one and then passed to the next recursion level. In the 3D case, this process generalizes to processing 2^3-shaped blocks where one corner is stored verbatim and the other corners are processed recursively. Note that a complexity of $\mathcal{O}(\log^3 n)$ still yields numbers in the thousands for $n \geq 1024^3$ and above. Our approach improves significantly in this regard.

Memory Efficient Integral Volume (MEIV) by Urschler et al. [UBD13] first computes the full SVT and then partitions it into bricks of small size (brick sizes of 3^3 up to 12^3 were investigated). For each brick, MEIV stores its smallest prefix sum *bo* together with a parameter μ describing a one-parameter model for the brick entries subtracted by *bo*. The value of μ is determined by an optimization step performing binary search. Since the model cannot fit all block entries perfectly, the remaining error per entry is stored in a dynamic word length storage with smallest as possible precision. As a result, MEIV is able to decrease memory consumption exceptionally well with regard to the minimal overhead of fetching only two values from memory, namely some brick information and a value from the dynamic word length storage. Its clear downside is the increased construction time by fitting μ. In the authors' experiments, constructing the MEIV representation with optimal block size for the *largeRandomVolume* dataset takes 75 times longer than for the regular SVT. Further, MEIV cannot give any memory guarantees as the final memory consumption is sensitive to the chosen brick size as well as the actual dataset. Compared to MEIV, we obtain the same savings in memory by allowing 6 instead of 2 fetches from memory, and—more importantly—memory requirements of our approach are known before actual encoding. Further, our approach is able to flexibly adapt memory requirements in the full range of $\mathcal{O}(n)$ to $\mathcal{O}(n \log n)$ respecting the user's need, and thus still can be used whereas other approaches (especially Ehsan and MEIV) run out of memory.

3. Summed Volume Tables

We now briefly describe the basic concept underlying SVTs. For the sake of clarity, we do so on the example of a SAT, the 2D counterpart of a SVT, before we extend the concept to an arbitrary number of dimensions. Note that we use **1-based indexing** whenever accessing arrays during the course of the paper.

Given a two-dimensional array F of scalar values, an entry (x,y) of its corresponding SAT is computed by summing up all values contained in the rectangular subarray that is spanned by the indices $(1,1)$ and (x,y), that is

$$\text{SAT}_F[x,y] := \sum_{x' \leq x,\, y' \leq y} F[x',y'].$$

If the values of a SAT are precomputed, partial sums of F for arbitrary rectangular subarrays can be computed in constant time, by making use of the inclusion-exclusion principle. Instead of reading and adding up values of F along the whole region spanned by (x_1+1, y_1+1) and (x_2, y_2), it suffices to read the SAT-values at the corners of the selected subarray. It holds that

$$\sum_{x_1<x'\le x_2,\ y_1<y'\le y_2} F[x',y'] = \mathrm{SAT}_F[x_1,y_1] + \mathrm{SAT}_F[x_2,y_2] - \mathrm{SAT}_F[x_1,y_2] - \mathrm{SAT}_F[x_2,y_1]$$

with $\mathrm{SAT}_F(x,y)$ set to zero if $x=0$ or $y=0$. Thus, SATs reduce the summation of $(x_2-x_1)\cdot(y_2-y_1)$ values to a summation of four values. The concept of SATs extends to any number of dimensions. Given a d-dimensional array F, the corresponding d-dimensional SVT is realised by

$$\mathrm{SVT}_F[v_1, v_2, \ldots, v_d] := \sum_{v'_i \le v_i} F[v'_1, v'_2, \ldots v'_d].$$

Partial sums of hyperboxes (line segments in 1D, rectangles in 2D, cuboids in 3D, and so on) can be computed by evaluating a SVT at the 2^d corner points of that hyperbox. In the special case of $d=1$, a SVT stores prefix sums of a 1D array. The entries in a SVT can be interpreted as d-dimensional prefix sums regarding axis-aligned volumes.

4. Hierarchical SVT data structure

We now introduce a hierarchical approach that allows identifying the intermediate representation to store a SVT so that a prefix sum can be computed with as little as possible additional compute under a prescribed memory budget. Here we assume that the input array F stores non-negative integral numbers. Thus, negative entries need to be eliminated by appropriate shifting, and floating point values need to be rescaled. While shifting does not affect accuracy, since the sign bit can be reused, floating point numbers cannot be rescaled to integers in a reasonable way if their value range is too large. In this case, however, computing partial sums even with the plain SVT is likely to fail due to numerical errors introduced by adding and subtracting potentially large prefix sums.

We require the following notation: An array F is said to have shape $n \in \mathbb{N}^d$ if and only if it is d-dimensional of shape $n_1 \times n_2 \times \cdots \times n_d$. We define the size of n by $|n| := \prod_{i=1}^{d} n_i$. In particular, an array of shape n has $|n|$ elements. Further, for any multi index $v \in \mathbb{N}^d$ and integers $k \in \{1,\ldots,d\}$, $i \in \mathbb{N}$, we denote by $v|_{k=i}$ the multi index that is obtained by replacing the k-th component of v by i. When accessing array elements via a multi index, $F[v]$ is a shorthand notation for $F[v_1,\ldots,v_d]$.

Our proposed intermediate representation (i.e, a data structure) evolves around the concept of splitting the input array F of shape $n \in \mathbb{N}^d$ into a small array F_a of precomputed, high precision aggregates and a set $\{F_{s_0}, F_{s_1}, \ldots\}$ of low precision subarrays such that any prefix sum can be efficiently computed from one prefix sum of the aggregate array and one prefix sum of a single subarray. Fig. 2 illustrates an exemplary split of a 3D input array. The aggregates are obtained by summing values of F along bands following one of the d dimensions, let us say the k-th one, which is called the *split dimension*. Each band is dissected into multiple segments by

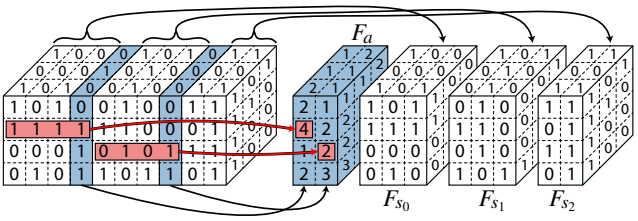

Figure 2: *Splitting a $10\times4\times3$ block along the largest dimension into the aggregate array F_a and three subarrays $F_{s_0}, F_{s_1}, F_{s_2}$. Cut position are marked in blue. Red blocks and arrows indicate how two entries of F_a are formed by summation.*

cutting at positions $c_1 < c_2 < \cdots < c_\ell$, called *split positions*, along the split dimension. For each segment, its values of F are summed to form an aggregate. All aggregates are arranged in the aggregate array F_a of shape $n|_{k=\ell}$. More precisely, it is

$$F_a[v] = \sum_{i=c_{v_k-1}+1}^{c_{v_k}} F[v|_{k=i}],$$

where $v \in \mathbb{N}^d$ and $c_0 := 0$. The prefix sums of F_a correspond to prefix sums of F ending at corners $v \in \mathbb{N}^d$ with $v_k \in \{c_1,\ldots,c_\ell\}$.

To enable the computation of any prefix sum of F, additional $\ell+1$ subarrays $F_{s_i} \subseteq F$ of shape $n|_{i=c_{i+1}-c_i-1}$ with $F_{s_i}[v] = F[v|_{k=c_i+v_k}]$ are defined. Here, i ranges from 0 to ℓ with $c_{\ell+1} := n_k+1$. Any prefix sum of F up to the corner $v \in \mathbb{N}^d$ now can be computed as follows: The index of the last split position not succeeding v, i.e., $i = \max(\{m \mid c_m \le v_k\} \cup \{0\})$, and the subarray offset $j = v_k - c_i$ are determined. Then it is

$$\mathrm{SVT}_F[v] = \mathrm{SVT}_{F_a}[v\,|_{k=i}] + \mathrm{SVT}_{F_{s_i}}[v\,|_{k=j}]. \tag{1}$$

Note that due to $\ell + \sum_{i=0}^{\ell}(c_{i+1}-c_i-1) = c_{\ell+1}-c_0-1 = n_k$, the number of values stored in F equals the numbers of values stored in F_a and all subarrays. Since values in SVTs of subarrays arise as sums over only a fraction of values of F, they require less bits of precision than values in the SVT of F. Thus, storing SVTs after applying the splitting operation comes with memory savings at the cost of one additional memory fetch and addition per prefix sum query on F. The memory savings can be reinforced at the cost of more fetches by recursively applying the split process to the newly acquired arrays F_a and all F_{s_i} until a certain termination condition holds. Then, each terminal array of small shape is stored by encoding either its entries (verbatim), or the entries of its SVT in fixed precision. The result is a split hierarchy of which an example is shown in Fig. 3. To compute a prefix sum from this representation, Eq. (1) is applied recursively up to the point where a prefix sum can be derived from a terminal array stored in memory.

4.1. The parameter tree

We describe a specific split hierarchy by means of a *parameter tree*. When splitting the input array F, we encode split dimension as well as split positions into the root node of the parameter tree. If a newly acquired array (aggregate array or subarray) is split further, a

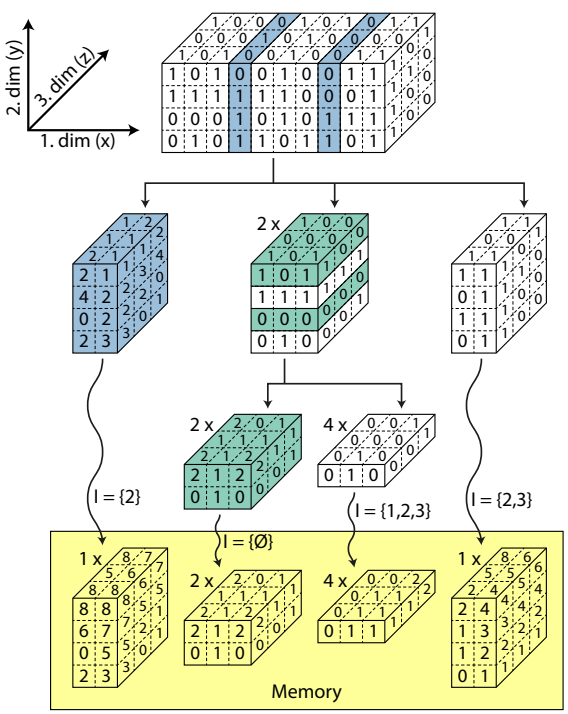

Figure 3: *Sample split hierarchy of a* $10 \times 4 \times 3$ *binary input array. We show aggregate arrays and subarrays generated during the recursive split process. Whenever two or more subarrays of similar shape form in a split operation, a multiplier in the top left corner of a block indicates how many arrays of its shape arise. The block's filling with numbers matches the first (i.e. leftmost/undermost) of its associated subarrays. The other subarrays of similar shape may contain different numbers. As a final step indicated by wavy arrows, each terminal array is processed by computing cumulative sums according to the leaf parameter I (see Sec. 4.1). The result is stored in memory.*

parameter subtree representing its subsequent split process is built and attached to the root node. If a newly acquired array is terminal, a leaf node describing its memory layout is created and attached instead.

In a naive implementation, the parameter quickly becomes unmanageable since it is branching with a factor that scales with the number of subarrays per split. By recursively splitting all subarrays of similar shape in the same way, one can collapse all of their corresponding subtrees to a single one and thus reduce the branching factor to the number of different subarray shapes occurring in a split, plus one for the aggregate array. Hence, we constrain the branching factor by requiring as few as possible different subarray shapes. This can be achieved by specifying a fixed subarray size z along the split dimension and placing split positions accordingly with equal spacing. However, we have to resolve alignment issues if $z+1$ is not a divisor of the length n_k of the split dimension minus one. Overall, we experimented with three different alignment strategies:

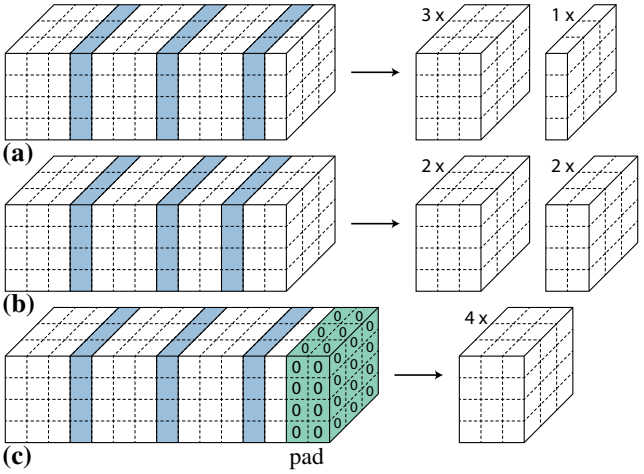

Figure 4: *Splitting a* $13 \times 4 \times 3$ *array along the longest dimension into subarrays of size* 3 *with* ***(a)*** *at_end,* ***(b)*** *distributed and* ***(c)*** *pad alignment. Split positions are highlighted on the left. The number and shape of resulting subarrays is shown on the right.*

at_end: The last subarray remains "incomplete" and thus has a size smaller than z along the split dimension.

distributed: A slice is removed from as many subarrays as one would have to pad in order to expand the last subarray to size z along the split dimension.

pad: The last subarray is padded with zeros until size z is reached.

Figure 4 depicts the results of all alignment strategies when splitting a $13 \times 3 \times 3$ field with fixed subarray size of 3 along the first dimension. During our experiments we noticed no difference in quality when generating SVT representations with either *at_end* or *distributed* aligned splits. Further, both yield subarrays of at most two different shapes, thus restricting the branching factor of the recursion to 3. *Pad* alignment incurs an additional memory overhead of up to 10%. In return it guarantees a unique subarray shape, reducing the branching factor to 2. This property may be favourable when engineering massively parallel en- & decoding schemes in future work. In the scope of this paper we decided to utilize *distributed* aligned splits.

In summary, a split now is defined by the split dimension k and subarray size z that allows to infer the split positions according to the *distributed* alignment strategy. Both values are encoded into an internal tree node representing the split. A leaf node, on the other hand, contains a set of dimension indices $I \subseteq \{1,\dots,d\}$ which describe along which dimensions array values are cumulated before finally storing each cumulated value in memory. The special cases of storing verbatim or SVT entries are represented by $I = \emptyset$ and $I = \{1,\dots,d\}$, respectively. Fig. 5 shows the parameter tree describing the split hierarchy of Fig. 3. Note that a tree node may represent more than one array by collapsing subtrees of similarly shaped subarrays. For instance, two arrays of shape $3 \times 4 \times 3$ are represented by the "k= 2, z= 1" internal node, and the memory layout of the four terminal arrays of shape $3 \times 1 \times 3$ is given by the "I = {1,2,3}" node.

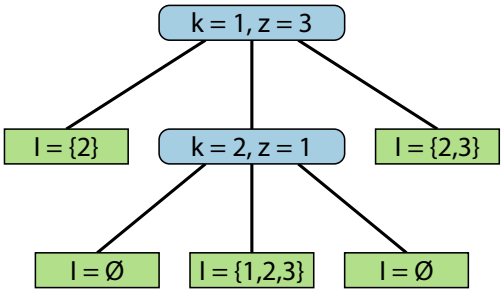

Figure 5: *A parameter tree of depth* 2 *describing the split hierarchy of Fig. 3. It has two internal (blue) and five leaf nodes (green). The leftmost subtree of each internal node describes the split hierarchy of the aggregate array whereas the remaining subtrees describe the split hierarchy of subarrays for two different subarray shapes. Text in nodes indicate encoded parameters. Note that the lower right leaf node can be chosen arbitrarily since for the 3D shape given in Fig. 3 there is only one unique subarray shape arising from the lower split.*

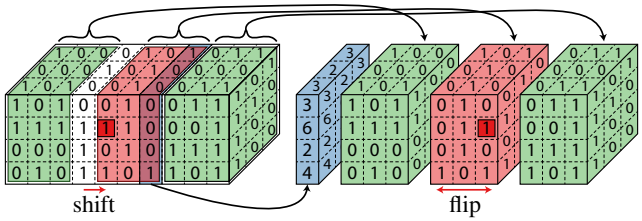

Figure 6: *Splitting a* $10 \times 4 \times 3$ *array by permitting subtraction. Compared to Fig. 2, the first split position is introduced after two subarrays instead of one, and the second (red) subarray is conjugated by first shifting by one and then flipping. Note how the boxed* 1 *changes position. Subarrays in green are not conjugated.*

4.2. The conjugate trick

To reduce the numbers of split positions and thus the number of high precision aggregates by one half (without changing the subarray size), we are generalizing the technique described by Ehsan et al. [ECRMM15]. Instead of adding up a subarray prefix sum and the aggregate prefix sum at the preceding split position as in Eq. (1), one can obtain the same result by subtracting the prefix sum of a flipped subarray from the aggregate prefix sum at the subsequent split position. If F_{s_i} is the $(i+1)$-th subarray with entries $F_{s_i}[v] = F[v|_{k=c_i+v_k}]$, we denote with $F^*_{s_i}$ its *conjugate* that is obtained by shifting by one and mirroring along the split dimension, i.e. $F^*_{s_i}[v] = F[v|_{k=c_{i+1}+1-v_k}]$. Then we have

$$\mathrm{SVT}_F[v] = \mathrm{SVT}_{F_a}[v\,|_{k=i}] - \mathrm{SVT}_{F^*_{s_{i-1}}}[v\,|_{k=j}], \tag{2}$$

where $i = \min\{m \mid c_m \geq v_k\}$ and $j = c_i - v_k$.

By replacing every second subarray by its conjugated version, one out of two split positions become superfluous. An exemplary split resulting from this process is shown in Fig. 6. Whenever a corner v is located in a subarray with odd index, Eq. (2) is used to compute the prefix sum, and Eq. (1) otherwise. If the last subarray of a split is a conjugate one, a split position at n_k (the last possible position) is added to ensure that a subsequent split position always exists. Allowing for both addition and subtraction and thus halving the size of F_a generally improves the final SVT representation by a more shallow split hierarchy and/or smaller block sizes at leaf levels. This is advantageous as smaller SVT leafs require less precision per entry, and smaller verbatim leafs require less additions to obtain a prefix sum.

5. Analysis of the parameter tree

If the shape of the input array F as well as its largest possible entry (usually of the form $2^{\#\text{bits}} - 1$) is known in advance, all relevant properties of a SVT representation of F can be derived via a top-down-top traversal of the corresponding parameter tree describing the split hierarchy. When descending the tree, information about shapes and largest possible entries is propagated according to the split parameters k, z. Note that whereas the largest possible entry for subarrays can be taken over from the array being split, for aggregate arrays, an additional factor of $2z+1$ has to be applied. It matches the number of values that are summed up to compute a single entry of the aggregate array when utilizing the conjugate trick (see Sec. 4.2).

5.1. Memory requirements

Let T be the parameter tree. If T consists of a single, terminal node with dimension indices I, the largest possible entry that will be stored in memory is given by $m \cdot \prod_{i \in I} n_i$, where n is the array shape at the terminal node and m is the array's largest possible entry. Consequently, we have

$$\mathrm{MEM}(T) = |n| \cdot \lceil \log_2(1 + m \cdot \prod_{i \in I} n_i) \rceil.$$

If the root node of T is an internal node, let T_a be the subtree describing the representation of the aggregate array F_a, and let T_{s_1} and T_{s_2} be the two subtrees describing representations for the two distinct subarray shapes. Then, the memory consumption can be computed as

$$\mathrm{MEM}(T) = \mathrm{MEM}(T_a) + \lambda_1 \cdot \mathrm{MEM}(T_{s_1}) + \lambda_2 \cdot \mathrm{MEM}(T_{s_2}),$$

where λ_i is the number of subarrays with shape represented by T_{s_i}.

The memory required for storing the parameter tree itself is negligible. Even for large datasets of GB-scale, the whole parameter tree is of KB-size. If the parameter tree is fixed beforehand and baked into the encoding & decoding algorithm, it does not need to be stored at all.

5.2. Estimation of fetch operations

The parameter tree T exposes an upper bound for the number of fetch operations required to compute a prefix sum. We call this bound the *fetch estimate* for T and denote it by $\mathrm{FETCH}(T)$. If the root node of T is terminal with dimension indices I, we require

$$\mathrm{FETCH}(T) = \prod_{i \in \{1,\ldots,d\} \setminus I} n_i$$

fetches to compute a prefix sum, with n being the array shape at the terminal node. If the root node of T is internal, we have

$$\begin{aligned}\mathrm{FETCH}(T) = &\mathrm{FETCH}(T_a) + \\ &\max(\mathrm{FETCH}(T_{s_1}), \mathrm{FETCH}(T_{s_2}))\end{aligned} \tag{3}$$

with the same notation as in Sec. 5.1. Since computing a prefix sum according to Eq. (1, 2) is performed by querying a prefix sum of the aggregate array and one subarray, the fetch estimate is an upper bound for the number of fetch operations—however, not necessarily the minimal one. In the appendix, we present a method for computing a tighter bound that in our experiments is lower by 3% on average and 24% at most. All fetch counts presented in this paper are computed with the tighter bound instead of the fetch estimate.

A very coarse upper bound for the number of compute operations per prefix sum can be given by four times the number of fetches. Since memory access is of magnitudes slower than simple arithmetic operations, we conclude that computing prefix sums is memory-bound. Hence, we use the number of fetch operations as performance indicator.

5.3. Update costs

Whenever a value of F is modified, a single aggregate of F_a as well as one value of the subarray containing the modified value have to be updated. Hence, the cost $\mathrm{UPDATE}(T)$ for updating the SVT representation can be described by the same recursion formula (3) as for the fetch estimate. At terminal nodes however, update costs are computed by

$$\mathrm{UPDATE}(T) = \prod_{i \in I} n_i$$

since an entry of a terminal array of shape n can be part of up to $\prod_{i \in I} n_i$ sums stored in memory.

5.4. Construction costs

In the supplement, we show that each value stored in memory by our SVT representation is a certain partial sum of the input array F. Hence, all stored values can be efficiently determined by computing the classical SVT of F for instance via GPU computing, and then sampling a partial sum from the SVT for each stored value. Since our representation stores n values at leaf nodes, and SVTs can be computed in $\mathcal{O}(n)$ and sampled in $\mathcal{O}(1)$, the overall runtime-complexity of the construction algorithm is $\mathcal{O}(n)$. Due to the fundamental assumption that $\mathcal{O}(n \log n)$ of main memory is not available, we propose to realize the classical SVT via bricking strategies falling back to larger, but slower memory (e.g. persistent storage). After construction, partial sums can be queried by fetching data from the constructed SVT representation that is stored in (fast) main memory.

6. Identification of optimal parameter trees

Our split hierarchy design opens up a high-dimensional search space for SVT representations, with the parameters trees being elements in the space of parameters defining the hierarchy. Ideally,

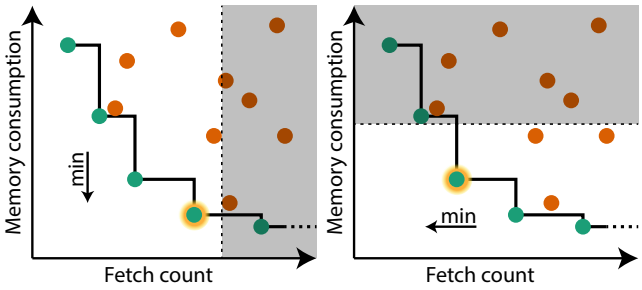

Figure 7: *Schematic illustration of parameter tree search spaces. Each dot represents a parameter tree of certain fetch count and memory consumption. Optimal parameter trees are colored in green and define the memory-performance trade-off curve shown in black. By restricting the search space in one quantity and optimizing for the other, the optimum is uniquely determined (highlighted dot).*

querying this search space for SVT representations yields a parameter tree instance that minimizes both the fetch count and the memory consumption with respect to input arrays of fixed shape and precision. However, representations with low fetch count have a high memory consumption and vice versa. To obtain a well defined optimization problem, we restrict the search space to representations that do not exceed a prescribed budget of *either* the number of fetch operations *or* the memory consumption, and then ask for the SVT representation that minimizes the respective other quantity. The resulting parameter trees follow a memory-performance trade-off curve as shown in Fig. 7.

The parameters which define a parameter tree are all discrete quantities, so that we are facing a combinatorial optimization problem. Even though we do not give a formal proof here, we believe that this problem is NP-hard. Thus, we propose a heuristic $\mathcal{H}$ that receives the shape n of the input array as well as a control parameter λ and returns a parameter tree $\mathcal{H}(\lambda, n)$ that is close to the memory-performance trade-off curve. Increasing [decreasing] λ typically results in parameter trees with higher [lower] fetch count and lower [higher] memory consumption. In particular, this design allows finding a beneficial parameter tree for a prescribed budget B of either fetches or—more interestingly—memory. It is achieved by defining the function

$$f(\lambda) = \begin{cases} \mathrm{FETCH}(\mathcal{H}(\lambda, n)) - B & \text{if fetch budget} \\ \mathrm{MEM}(\mathcal{H}(\lambda, n)) - B & \text{if memory budget} \end{cases}$$

and applying a root-finding method such as the bisection method to find λ with $f(\lambda) \leq 0$ being close to zero. Then, $\mathcal{H}(\lambda, n)$ is a parameter tree that, on the one hand, minimizes the unconstrained quantity, and exhausts the given budget on the other hand.

In the algorithmic formulation of the heuristic, λ represents a threshold for the fetch estimate described in Sec. 5.2. It is guaranteed that the fetch estimate of $\mathcal{H}(\lambda, n)$ does not exceed λ. We achieve this by the following procedure: If λ equals one, the heuristic returns the parameter tree representing a classical SVT; that is a single leaf node with $I = \{1, \ldots, d\}$. If, on the other hand, λ is at least as large as the shape product $|n|$, it returns a single leaf

node with setting $I = \emptyset$—that is the verbatim representation. For $1 < \lambda < |n|$, the heuristic builds an internal node. The split dimension k is chosen such that $n_k = \max(n_1, \ldots, n_d)$. The subarray size z is derived from λ according to an interpolation function that is manually defined to match the structure of optimal parameter trees for small array shapes. These were computed once by a Branch-and-Bound strategy that performs an exhaustive search.

The split parameters k and z define the shapes $n^{(a)}$, $n^{(s_1)}$, $n^{(s_2)}$ of the aggregate array and the subarrays. Thus, we can use the heuristic recursively to compute the subtrees of the internal node. To do so, we define control parameters λ_a, λ_s and set the aggregate subtree to $\mathcal{H}(\lambda_a, n^{(a)})$ and both subarray subtrees to $\mathcal{H}(\lambda_s, n^{(s_1)})$ and $\mathcal{H}(\lambda_s, n^{(s_2)})$ respectively. By requiring $\lambda_a, \lambda_s \geq 1$ and $\lambda_a + \lambda_s \leq \lambda$, we assure that the recursion terminates and that the fetch estimate of the final parameter tree does not exceed λ. This is due to Eq. (3) and the assumption that the heuristic already satisfies this guarantee for recursive calls. In order to find a reasonable choice for λ_a and λ_s, we again analyzed optimal parameter trees and noticed that the ratio of the fetch estimate $\text{FETCH}(T_a)$ of the aggregate subtree to the fetch estimate $\text{FETCH}(T)$ of the whole tree correlates to the ratio of the number of split positions $(n^{(a)})_k$ to the length n_k of the split dimension. While the latter ratio can be computed from the split parameters, the first is unknown yet. However, by assuming that the observed correlation applies for arbitrary array shapes, we can derive an estimate for the first ratio. Now, the exact value for λ_a is obtained by matching λ_a/λ to the estimate of the first ratio. Then, λ_s is computed by subtracting λ_a from λ. Pseudo code for the heuristic is given in the appendix.

7. Evaluation

The following evaluation sheds light on a) the quality of the heuristic to find an optimal parameter tree (Sec. 6) and b) on the properties of our derived SVT representations compared to alternative approaches, such as MEIV by Urschler et al. [UBD13], 3D Fenwick Trees by Mishra [Mis13], Partial SVTs by Zellmann et al. [ZSL18], and the approach of Ehsan et al. [ECRMM15].

All our experiments were run on a server architecture with 4x Intel Xeon Gold 6140 CPUs with 18 cores @ 2.30GHz each. Although we do not exploit any degree of parallelism, the generation of parameter trees with the proposed heuristic requires between 8.5s seconds for the smallest and 45.4s for the largest dataset (see Table 1). Timings reflect convergence speed of the bisection method used during parameter tree search as well as the cost for evaluating the heuristic in each bisection step. Parameter tree generation for single digit fetch counts is fast since the runtime of the heuristic correlates with the size of the parameter tree—and trees are very shallow in that scenario.

As the parameter tree has to be created only once for a fixed array shape and precision, we consider the runtime of the heuristic as negligible. Finding globally optimal parameter trees, however, is not tractable even for MB-scale datasets. The Branch-and-Bound strategy for precomputing optimal parameter trees already takes 12.5 hours for a 64^3 dataset, clearly necessitating the use of a heuristic.

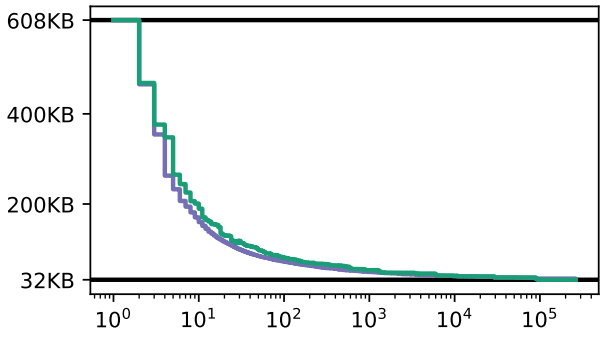

Figure 8: *Memory-performance curves for a binary 64^3 volume, when **(blue)** solving the global combinatorial optimization problem of parameter trees, and when **(green)** utilizing parameter trees returned by our heuristic. The x-axis shows number of fetch operations. The y-axis shows memory consumption. Black lines indicate lower and upper bounds for the memory consumption of SVT representations.*

7.1. Quality of the heuristic

We now evaluate how closely the parameter trees returned by the heuristic match the optimal parameter trees. The optimal memory-performance curve for a binary 64^3 volume is shown by the blue curve in Fig. 8. A point (x, y) on the curve indicates that there exists an optimal parameter tree with x fetches and y bits of memory consumption, i.e., if memory is constrained to y bits, our hierarchical data structure principally allows to reduce the number of required fetches to x, and vice versa. The green curve indicates the characteristic of our proposed heuristic. By measuring the shift in x-direction, one can determine the fetch operation overhead of the heuristic for a fixed memory threshold. Compared to the optimal solution, roughly 1.5 times the optimal amount of fetches are required. Note that due to the logarithmic scale of the x-axes, a shift translates to a factor instead of an offset. Vice versa, measuring the shift in y-direction yields the memory overhead of the heuristic, assuming a fixed budget of fetches. Here we can clearly see that the heuristic performs quite well except for the number of 4 fetches, where memory consumption is increased by 25%.

Note that although the memory-performance curve given by the heuristic is not guaranteed to be monotone, it shows a clear falling trend. Thus, when using the bisection method w.r.t. λ to achieve a certain memory threshold, close to perfect results can be expected. We are also confident that the heuristic has not been manually overfitted to the validation scenario. In the design phase, optimal parameter trees for various 1D to 4D arrays with at most 9.000 elements were investigated, while the volume used in the evaluation contains 262.144 elements. On the other hand, due to this we cannot ensure that the results of the evaluation generalize to GB-scale arrays.

7.2. Comparative study

In this study, we compare the SVT representations found by our heuristic to the approaches proposed by Ehsan et al. [ECRMM15],

Table 1: *Performance statistics for various SVT representations. Each group of three rows contains results (memory consumption, fetch count) for different approaches using the same volume. For each volume, theoretical lower and upper bounds for SVT representations are given. The **Reference** column shows results for the reference approaches by others. The **Ours** columns show results and timings for parameter tree computation for our SVT representations under varying constraints. Either memory or fetch count is constrained according to the reference in the same row. Note that the non-constrained quantities (in **bold**) are significantly lower than the corresponding reference quantities.*

Volume	Reference			Ours (Memory $\leq$ Ref. Memory)			Ours (Fetch $\leq$ Ref. Fetch)		
	Name	Memory	Fetch	Memory	Fetch	Timing	Memory	Fetch	Timing
$256 \times 256 \times 256$	Ehsan	36.6MB	8	35.1MB	**3**	20ms	**14.9MB**	8	41ms
Size: 2MB	Part. SVT	32MB	512	27.0MB	**4**	25ms	**3.9MB**	476	8.0s
SVT: 50MB	Fen. Tree	8.0MB	512	8.0MB	**38**	8.5s	**3.9MB**	476	8.1s
$1024 \times 1024 \times 1024$	Ehsan	2.8GB	8	2.7GB	**3**	23ms	**1.2GB**	8	52ms
Size: 128MB	Part. SVT	2GB	32K	1.5GB	**5**	147ms	**170.1MB**	27.5K	31.4s
SVT: 3.9GB	Fen. Tree	511.6MB	1000	511.6MB	**40**	6.9s	**252.7MB**	952	20.5s
$2048 \times 2048 \times 2048$	Ehsan	24.3GB	8	24.2GB	**3**	26ms	**10.3GB**	8	56ms
Size: 1GB	Part. SVT	16GB	256K	12.7GB	**5**	161ms	**1.2GB**	255.8K	33.5s
SVT: 34GB	Fen. Tree	4.0GB	1331	4.0GB	**39**	9.7s	**1.9GB**	1294	17.3s
$641 \times 9601 \times 9601$	Ehsan	176.6GB	8	168.1GB	**3**	30ms	**71.2GB**	8	51ms
Size: 6.9GB	Part. SVT	110.1GB	1.8M	86.7GB	**5**	157ms	**7.5GB**	1.7M	40.4s
SVT: 247.6GB	Fen. Tree	27.5GB	1521	27.0GB	**46**	11.8s	**11.8GB**	1468	34.6s
$8192 \times 8192 \times 8192$	Ehsan	1.8TB	8	1.8TB	**3**	51ms	**725.5GB**	8	40ms
Size: 64GB	Part. SVT	1TB	16M	892.8GB	**5**	100ms	**67.9GB**	10.6M	39.1s
SVT: 2.5TB	Fen. Tree	256.0GB	2197	252.7GB	**46**	10.7s	**114.4GB**	2188	45.4s

Mishra [Mis13], Zellmann et al. [ZSL18] and Urschler et al. [UBD13]. Table 2 summarizes the qualitative features supported by the varying approaches. For a quantitative analysis, we manually compute the memory consumption as well as the number of fetch operations for all reference methods except for Urschler et al., which will be covered later. We use binary volumes of shape 256^3 and $1K^3$, to reproduce the results of the alternatives from other works. To demonstrate the scalability of our approach, additional results using large scale binary datasets from $2K^3$ to $8K^3$ are presented. The results of these experiments are given in Table 1. They generalize to arrays with elements of arbitrary precision p, by adding an offset of $(p-1) \cdot n$ to all memory footprints of both our and reference methods.

It can be seen that our proposed heuristic performs significantly better than any other approach relying on intermediate sum computation. In all cases, we can achieve an improvement of more than a factor of 2.5 in memory consumption or number of fetch operations while matching the budget regarding the respective other quantity. Notably, while it seems that the approach by Ehsan is in all scenarios only this factor behind us, it cannot reduce the memory consumption any further. Thus, where Ehsan requires 1.8TB, our SVT variant can go down as low as 64GB. In comparison to Partial SVTs, we can trade almost all fetch operations for the memory requirement of 67.9GB, and can reduce the memory requirement about more than 90% at the same number of fetch operations. Compared to the 3D Fenwick Tree, our SVT representation requires only 2% of the number of fetch operations at similar memory consumption.

It is fair to say, however, that the improvements over Partial SVTs with respect to memory requirement become less significant with increasing sparsity of the volume. Partial SVTs first split the 3D array into subarrays of size 32^3, so that empty subarrays can be pruned and do not need to be stored. Even though the so generated sparse structure requires a certain overhead to encode the sparsity information, it is likely that at extreme sparsity levels the Partial SVTs become competitive with respect to memory consumption. It is, on the other hand, not the case that the number of fetch operations decreases similarly, since indirect memory access operations are required to step along the sparse encoding.

Another comparison we perform is against MEIV of Urschler et al. [UBD13], by reusing the array shapes and precision of the datasets used in the authors' work. We set the lowest achieved memory consumption achieved by MEIV as fixed memory budget and compute the parameter tree according to Sec. 6. For the *smallRandomVolume* dataset of shape 512^3 and a maximal possible entry of 1023 we require 6 fetch operations and 311MB of memory (compared to 319MB by MEIV). For the *largeRandomVolume* dataset of shape $1K^3$ and a maximal possible entry of 512 we again require 6 fetch operations, but 2491MB of memory (compared to 2544MB by MEIV). In the case of the *realCTVolume* dataset with shape $512 \times 512 \times 476$ and a maximal possible entry of 1023, MEIV achieves roughly 350MB of memory consumption. We achieve 304MB while requiring 5 fetch operations. Clearly, we achieve results of equal quality compared to MEIV without having its limitations as described in Sec. 2.

7.3. Meteorological use case

In meteorological and climatological research, historical weather data such as the publicly available ERA5 data set [HBB*20] containing global, atmospheric reanalysis data is often analyzed using statistical measures like mean and variance over spatial subdomains and time intervals. These measures indicate trends and can

Table 2: *Properties of different SVT representations where n is the number of data values in the volume. Besides memory consumption, we show runtime-complexities for reconstructing an arbitrary partial sum, reconstructing an actual data value, single-threaded construction of the data structure, and updating the representation after a single data value is changed. Further, we indicate if memory consumption can be predicted before constructing the representation, if construction is straightforwardly parallelizable, and if representations can be modified to exploit sparsity in datasets. For our approach, we present lower and upper complexity bounds for all achievable representations. (*) Average runtime-complexity is shown. Worst Case complexity is $\mathcal{O}(\log^3 n)$. (**) Our approach reconstructs data values with the same amount of fetches as required for partial sum reconstruction by regarding data values as partial sums over hyperboxes of size 1. (***) Update performance heavily depends on the actual parameter tree (see Sec. 5.3).*

	Verbatim	**SVT**	**Ehsan**	**Part. SVT**	**Fen. Tree**	**MEIV**	**Ours**
Memory consumption	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$ large const.	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$ better in practice	$\mathcal{O}(n) - \mathcal{O}(n \log n)$
Read partial sum	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$ small const.	$\mathcal{O}(\log^3 n)$	$\mathcal{O}(1)$	$\mathcal{O}(1) - \mathcal{O}(n)$
Read data value	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$ *	$\mathcal{O}(1)$	$\mathcal{O}(1) - \mathcal{O}(n)$ **
Construction time	–	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$ large const.	$\mathcal{O}(n)$
Data value update	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^3 n)$	$\mathcal{O}(n)$	$\mathcal{O}(1) - \mathcal{O}(n)$ ***
Predictable memory consumption	✓	✓	✓	✓	✓	x	✓
Parallelizable construction	–	✓	✓	✓	✓	✓	✓
Can exploit sparsity	✓	x	x	✓	x	x	?

be used to reveal correlations between observed physical quantities at different sub-domains and times.

As a use case, we utilize the 2m temperature field of the ERA5 hourly data on single levels from 1979 to 2020 [HBB*18]. Data is rescaled to 8bit integers according to Sec. 4 and hourly data is aggregated per day, yielding an 8bit precision dataset of size $1440 \times 721 \times 15404$. We use our parameterized SVT representation to plot the mean temperature progressions over different interactively selected, spatial sub-domains of 150×50 grid points (roughly matching the extend of the Sahara) in a line plot. Assuming a horizontal resolution of 200 pixels, we partition the user chosen time range in 200 equally large intervals and compute one aggregate for each pixel. When viewing the whole temporal domain, each aggregate thus describes a sub-domain of $150 \times 50 \times 77$ grid points and requires 577.500 fetches from the verbatim dataset of size 14.9GB. A SVT requires only 8 fetch operations but 78.2GB of memory. In contrast, by employing our parameterized SVT representation using 22.8GB of memory, we can still perform the computation of mean values per any sub-domain using $31 \cdot 8$ fetch operations, facilitating an interactive visual analysis of arbitrary subregions.

8. Conclusion and future work

We have proposed a versatile data structure and heuristic for generating SVT representations that can flexibly trade speed for memory. Hence, SVT representations that are specifically built for a fixed memory or compute budget can be utilized. In a number of experiments on large scale datasets we have compared the resulting SVT representations to those by others, and we have demonstrated significantly reduced memory consumption at similar decoding performance, or vice versa.

In the future, we intend to address the following issues: Firstly, we will engineer cache-aware and/or GPU-accelerated encoding and decoding schemes, so that a) decoding can further benefit from massive parallelism and b) encoding can be realised in timings similar to state-of-the-art SAT encoding. [CWT*18, EFT*18, HSO07] Secondly, we will apply our approach to build spatial acceleration structures such as BVHs for large-scale mesh or volume datasets. Further, we plan to efficiently build implicit BVH structures for DVR that optimize for low variance in density per bounding volume. By using the technique described by Phan et al. [PSCL12], our technique allows to compute variances in constant time without running out of memory. Note that memory efficient SVT implementations are especially important in this regard, because the SVT that is used for computing second order moments is created from an input of double precision than the dataset. Third, we will investigate potential optimizations for sparse data. Here we will address how much memory used by our data structure can be saved by pruning empty subarrays, and whether the heuristic can be adapted to respect empty regions in the dataset.

Further, we plan to extend our approach to nominal data, that is, computing SVTs of histograms instead of scalar entries. Applications are in any research field processing segmented volumes such as neuroscience or material science. For instance, Al-Thelaya et al. [ATAS21] perform sub-volume queries over nominal data to enable real-time computation of local histograms over user selected regions. However, due to arranging histograms in a Mip Map structure, their approach requires an additional footprint assembly step that quickly becomes unfeasible if very large regions are selected. By replacing the Mip Map architecture with our SVT scheme, we believe it will be possible to build a sophisticated mixture graph that allows to skip the footprint assembly step entirely.

References

[ATAS21] AL-THELAYA K., AGUS M., SCHNEIDER J.: The Mixture Graph-A Data Structure for Compressing, rendering, and querying segmentation histograms. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021), 645–655. doi:10.1109/TVCG.2020.3030451. 1, 10

[Bel08] BELT H. J. W.: Word Length Reduction for the Integral Image. In *15th IEEE International Conference on Image Processing* (2008), pp. 805–808. doi:10.1109/ICIP.2008.4711877. 3

[BSB10] BHATIA A., SNYDER W. E., BILBRO G.: Stacked Integral Image. In *IEEE International Conference on Robotics and Automation* (2010), pp. 1530–1535. doi:10.1109/ROBOT.2010.5509400. 2

[BTVG06] BAY H., TUYTELAARS T., VAN GOOL L.: SURF: Speeded Up Robust Features. In *Computer Vision – ECCV 2006* (Berlin, Heidelberg, 2006), Leonardis A., Bischof H., Pinz A., (Eds.), Springer Berlin Heidelberg, pp. 404–417. 2

[Cro84] CROW F. C.: Summed-Area Tables for Texture Mapping. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1984), SIGGRAPH '84, Association for Computing Machinery, p. 207–212. doi:10.1145/800031.808600. 1, 2

[CWT*18] CHEN P., WAHIB M., TAKIZAWA S., TAKANO R., MATSUOKA S.: Efficient Algorithms for the Summed Area Tables Primitive on GPUs. In *IEEE International Conference on Cluster Computing (CLUSTER)* (2018), pp. 482–493. doi:10.1109/CLUSTER.2018.00064. 10

[ECRMM15] EHSAN S., CLARK A. F., REHMAN N. U., MCDONALD-MAIER K. D.: Integral Images: Efficient Algorithms for Their Computation and Storage in Resource-Constrained Embedded Vision Systems. *Sensors 15*, 7 (2015), 16804–16830. doi:10.3390/s150716804. 3, 6, 8

[EFT*18] EMOTO Y., FUNASAKA S., TOKURA H., HONDA T., NAKANO K., ITO Y.: An Optimal Parallel Algorithm for Computing the Summed Area Table on the GPU. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2018), pp. 763–772. doi:10.1109/IPDPSW.2018.00123. 10

[Fen94] FENWICK P. M.: A New Data Structure for Cumulative Frequency Tables. *Software: Practice and Experience 24*, 3 (1994), 327–336. doi:10.1002/spe.4380240306. 1, 3

[FLML14] FACCIOLO G., LIMARE N., MEINHARDT-LLOPIS E.: Integral Images for Block Matching. *Image Processing On Line 4* (2014), 344–369. doi:10.5201/ipol.2014.57. 2

[GGB06] GRABNER M., GRABNER H., BISCHOF H.: Fast Approximated SIFT. In *Computer Vision – ACCV 2006* (Berlin, Heidelberg, 2006), Narayanan P. J., Nayar S. K., Shum H.-Y., (Eds.), Springer Berlin Heidelberg, pp. 918–927. 2

[GM19] GANTER D., MANZKE M.: An Analysis of Region Clustered BVH Volume Rendering on GPU. *Computer Graphics Forum 38*, 8 (2019), 13–21. doi:10.1111/cgf.13756. 2

[HBB*18] HERSBACH H., BELL B., BERRISFORD P., BIAVATI G., HORÁNYI A., MUÑOZ, SABATER J., NICOLAS J., PEUBEY C., RADU R., ROZUM I., SCHEPERS D., SIMMONS A., SOCI C., DEE D., THÉPAUT J.-N.: ERA5 hourly data on single levels from 1979 to present, 2018. Copernicus Climate Change Service (C3S) Climate Data Store (CDS). Accessed on 08-03-2021. doi:10.24381/cds.adbb2d47. 10

[HBB*20] HERSBACH H., BELL B., BERRISFORD P., HIRAHARA S., HORÁNYI A., MUÑOZ-SABATER J., NICOLAS J., PEUBEY C., RADU R., SCHEPERS D., SIMMONS A., SOCI C., ABDALLA S., ABELLAN X., BALSAMO G., BECHTOLD P., BIAVATI G., BIDLOT J., BONAVITA M., DE CHIARA G., DAHLGREN P., DEE D., DIAMANTAKIS M., DRAGANI R., FLEMMING J., FORBES R., FUENTES M., GEER A., HAIMBERGER L., HEALY S., HOGAN R. J., HÓLM E., JANISKOVÁ M., KEELEY S., LALOYAUX P., LOPEZ P., LUPU C., RADNOTI G., DE ROSNAY P., ROZUM I., VAMBORG F., VILLAUME S., THÉPAUT J.-N.: The era5 global reanalysis. *Quarterly Journal of the Royal Meteorological Society 146*, 730 (2020), 1999–2049. URL: https://rmets.onlinelibrary.wiley.com/doi/abs/10.1002/qj.3803, arXiv:https://rmets.onlinelibrary.wiley.com/doi/pdf/10.1002/qj.3803, doi:https://doi.org/10.1002/qj.3803. 9

[Hec86] HECKBERT P. S.: Filtering by Repeated Integration. *SIGGRAPH Comput. Graph. 20*, 4 (Aug. 1986), 315–321. doi:10.1145/15886.15921. 2

[HHS06] HAVRAN V., HERZOG R., SEIDEL H.: On the Fast Construction of Spatial Hierarchies for Ray Tracing. In *IEEE Symposium on Interactive Ray Tracing* (2006), pp. 71–80. doi:10.1109/RT.2006.280217. 1, 2

[HPD08] HUSSEIN M., PORIKLI F., DAVIS L.: Kernel Integral Images: A Framework for Fast Non-Uniform Filtering. In *IEEE Conference on Computer Vision and Pattern Recognition* (2008), pp. 1–8. doi:10.1109/CVPR.2008.4587641. 2

[HSC*05] HENSLEY J., SCHEUERMANN T., COOMBE G., SINGH M., LASTRA A.: Fast Summed-Area Table Generation and its Applications. *Computer Graphics Forum 24*, 3 (2005), 547–555. doi:10.1111/j.1467-8659.2005.00880.x. 2

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. *GPU gems 3*, 39 (2007), 851–876. 10

[Mis13] MISHRA P.: A New Algorithm for Updating and Querying Subarrays of Multidimensional Arrays. *arXiv* (2013). arXiv:1311.6093v6. 1, 3, 8, 9

[Por05] PORIKLI F.: Integral Histogram: A Fast Way to Extract Histograms in Cartesian Spaces. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* (2005), vol. 1, pp. 829–836 vol. 1. doi:10.1109/CVPR.2005.188. 2

[PSCL12] PHAN T., SOHONI S., CHANDLER D. M., LARSON E. C.: Performance-Analysis-Based Acceleration of Image Quality assessment. In *IEEE Southwest Symposium on Image Analysis and Interpretation* (2012), pp. 81–84. doi:10.1109/SSIAI.2012.6202458. 2, 10

[SKB08] SHAFAIT F., KEYSERS D., BREUEL T. M.: Efficient Implementation of Local Adaptive Thresholding Techniques Using Integral Images. In *Document Recognition and Retrieval XV* (2008), Yanikoglu B. A., Berkner K., (Eds.), vol. 6815, International Society for Optics and Photonics, SPIE, pp. 317 – 322. doi:10.1117/12.767755. 2

[SR17] SCHNEIDER J., RAUTEK P.: A Versatile and Efficient GPU Data Structure for Spatial Indexing. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 911–920. doi:10.1109/TVCG.2016.2599043. 1, 3

[UBD13] URSCHLER M., BORNIK A., DONOSER M.: Memory Efficient 3D Integral Volumes. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV) Workshops* (June 2013). 3, 8, 9

[VJ04] VIOLA P., JONES M. J.: Robust Real-Time Face Detection. *International Journal of Computer Vision 57*, 2 (2004), 137–154. 2

[VMD08] VIDAL V., MEI X., DECAUDIN P.: Simple Empty-Space Removal for Interactive Volume Rendering. *Journal of Graphics Tools 13*, 2 (2008), 21–36. doi:10.1080/2151237X.2008.10129258. 1, 2

[Wil83] WILLIAMS L.: Pyramidal Parametrics. In *Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1983), SIGGRAPH '83, Association for Computing Machinery, p. 1–11. doi:10.1145/800059.801126. 2

[ZSL18] ZELLMANN S., SCHULZE J. P., LANG U.: Rapid kd Tree Construction for Sparse Volume Data. In *Proceedings of the Symposium on Parallel Graphics and Visualization* (Goslar, DEU, 2018), EGPGV '18, Eurographics Association, p. 69–77. 3, 8, 9