Memory Efficient 3D Integral Volumes

Martin Urschler, Alexander Bornik
Ludwig Boltzmann Institute for
Clinical Forensic Imaging, Graz, Austria
martin.urschler@cfi.lbg.ac.at

Michael Donoser
Institute for Computer Graphics and Vision
Graz, University of Technology, Austria
donoser@icg.tugraz.at

Abstract

Integral image data structures are very useful in computer vision applications that involve machine learning approaches based on ensembles of weak learners. The weak learners often are simply several regional sums of intensities subtracted from each other. In this work we present a memory efficient integral volume data structure, that allows reduction of required RAM storage size in such a supervised learning framework using 3D training data. We evaluate our proposed data structure in terms of the trade-off between computational effort and storage, and show an application for 3D object detection of liver CT data.

1. Introduction

The potentially huge size of volumetric 3D data sets from Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) often is a source of problems when adapting sophisticated computer vision algorithms to 3D medical image analysis. Especially machine learning approaches fall into this category. Supervised detection and segmentation methods for 3D anatomical structures have been adapted and extensively investigated. Reasons why machine learning is favored compared to hand-crafted algorithms are inherent difficulties of medical data due to the high variability of anatomical structures, low SNR, and the chance of pathologies. By letting a machine learning method figure out spatial and intensity-based relationships describing the structures of interest, one can make sure that available information from the data is used exhaustively.

Many sophisticated methods like Random Ferns [15] or Random Forests [6] have been developed with 2D computer vision applications in mind. While the extension to 3D data is simple from an algorithmic point of view, the practical implementation of a 3D random forest detection or segmentation algorithm requires a lot of careful design decisions in terms of memory consumption and computational efficiency to deal e.g. with large volumetric CT data sets. Of course one could argue, that the memory- and computation-intense training phase requires a dedicated compute server farm with lots of parallel processing and tens to hundreds GB of RAM. However, not all research groups have easy access to such a computing server infrastructure, and a reduction in memory consumption is certainly useful from a computational point of view and to be able to work in environments with memory and bandwidth restrictions like embedded devices.

The Random Forest framework is often used in a machine learning work-flow like the following. First a number of training data sets is collected and the object(s) of interest are annotated by an expert. For automated anatomical structure detection in 3D medical image analysis this might be a bounding box around the organ of interest, or for segmentation a manual or semi-automatic voxel-wise delineation of the same organ. Then the training data sets and respective annotations are used as the input for training the supervised machine learning approach. Often the training data, which in case of CT volumes are 12 bit intensity values in Hounsfield Units, is preprocessed to speed up voxel region access using an integral image data structure [20]. From this preprocessed data, the Random Forest framework trains a forest of decision trees, where each node of each tree tests a set of randomly chosen feature/threshold pairs (i.e. weak learners) with respect to their ability of discriminating the given training/annotation set according to a measure describing the information gain of the data split. The benefits of the random forest framework are the excellent generalization behaviour, the simple implementation of the greedy training scheme compared to other machine learning approaches [2] like neural networks or support vector machines, and the ability to train the trees of a forest efficiently in parallel, since trees are independent of each other.

In this work we propose a memory efficient 3D integral volume data structure, which helps in keeping low the memory consumption of machine learning approaches, when using simple weak learners based on rectangular voxel features like Adaptive Boosting [17] or Random Forests [3]. The data structure keeps the most important benefit of the integral image concept, the computation of regional inten-

sity features of arbitrary size in constant time. We focus on the Random Forest framework and describe how it can be used to perform 3D object detection, very similar to the recently presented approach of Criminisi et al. [5], but carefully optimized for memory efficiency with the help of the novel integral volume.

2. Related Work

Two relevant areas of related work are described, machine learning approaches using ensembles of weak learners, and the state of the art in 2D and 3D integral image data structures.

Object detection and segmentation have recently seen an increasing use of machine learning approaches built on simple features which are used as weak learners. By combining weak learners to ensembles powerful discriminative classification and regression algorithms became possible. A seminal example is the AdaBoost based face detector of Viola and Jones [20], where Haar like rectangular features are boosted [17] to form a strong classifier. Since these weak learners are evaluated billions of times over the training data, an efficient data structure, that allows the lookup of single intensities, as well as mean intensities over rectangular regions, was proposed in the form of the integral image. Later research has started to focus on random forests in computer vision [4, 18], where weak learners are organized in tree structures and a high degree of randomness in choosing weak learners for the nodes of the tree is injected into the training. Very recently this lead to the popular Hough Forest approach of Gall et al. [9], who use the Generalized Hough Transform for voting of object locations in a random forest framework. The need for efficient weak learner evaluation remained relevant for all these approaches.

Medical image analysis duplicated the recent trend towards decision tree- and forest-based algorithms for detection and segmentation of 3D anatomical structures. Early work comes from Tu et al. [19] who describe application of a probabilistic boosting tree for 3D colon segmentation from CT. Zheng et al. introduced the concept of marginal space learning for 3D object segmentation [21]. Later Montillo et al. showed a method for simultaneous segmentation of multiple structures in highly varying CT scans, based on entangled decision forests [13]. Very recently Donner et al. presented an object detection method [8] borrowing ideas from Random Forests and Hough Forests. It can be used for localizing 3D anatomical structures from keypoint annotations. A very promising approach was shown by Criminisi et al. [5]. It uses a multi-class random regression forest for localization of multiple objects from a huge database of CT scans. Their work underlines the need for memory efficient data structures when training machine learning algorithms from large sets of 3D volumetric data. They state that training is performed on server farms and they restrict themselves to use only down-sampled versions of the input data, a necessity they share with many of the other mentioned approaches. None of these works provide a detailed investigation on the memory consumption of their algorithms.

The integral image representation goes back to the computer graphics literature, with F. Crow describing the use of a summed-area table (SAT) for providing different resolution levels of 2D textures [7]. He showed the benefits of this data structure in efficiently accessing texture elements, and discussed the increased storage size of the SAT. As an idea to reduce size he proposed to partition an image into blocks to save bits. A. Glassner extended the idea of summed-area tables to multi-dimensional sum tables [10], which introduced 3D "summed-volume tables".

Later Viola and Jones used the idea of summed-area tables (under the name of integral images) for feature computation in their seminal paper on face detection by Boosting Haar-like features [20]. In the computer graphics community the need for highly efficient SAT generation was recently discussed. Hensley et al. describe fast SAT computation for rendering glossy reflections to simulate depth of field [12]. They build a summed-area table directly on the GPU by the technique of recursive doubling. They also discuss problems with numerical precision of summed-area table entries when represented as floating point values. One modification they propose is the centering around image pixel values, where they show the benefits in terms of precision. Another GPU based implementation of summed-area tables is described in [14]. It is based on parallel prefix sums implemented in the NVidia CUDA [1] environment.

H. Belt described methods to reduce the storage size of integral images [1]. He identified the large memory footprint of 2D integral images as a drawback, especially for embedded device implementations in mobile phones or in robotics, where memory is a scarce resource. So on a conceptual level he dealt with similar problems for 2D integral images like we aim for in this work for 3D integral volumes. His strategies for reducing the word length of integral image values are the use of complement coded arithmetics and computing through the overflow, which increases the effective range of values representable by a few bits, and a quantization of original pixel values prior to integral image calculation, which leads to approximative results.

From the large interest in supervised object detection/segmentation algorithms based on weak learners we can clearly see the need for highly efficient weak learner computation. The literature about integral volumes does not go into much detail about memory efficient implementations, however, due to the cubic scaling of the values of the integral volume, this is an important aspect. For 2D integral images some ideas to reduce memory costs have been described, especially to deal with embedded device limita-

[1] http://www.nvidia.org

tions in memory and bandwidth, but we are not satisfied with introducing inaccuracies into the data representation by rounding. Up to our knowledge no concise work on reducing memory consumption of 3D integral volumes has been presented that significantly reduces storage costs.

3. Integral Volumes

A precomputed integral volume resembles a representation of the intensities of an input volume, where for each location the sum of all the intensities up to the location are stored. It allows very efficient calculation of voxel regions of any size from a given input volume i of width W, height H, and depth D. Elements of i (the voxel intensity values) are stored in a three-dimensional array with indices $x \in [0, W-1]$, $y \in [0, H-1]$, and $z \in [0, D-1]$. Only volumes with non-negative elements are regarded, i.e. $i(x, y, z) \geq 0$. Note that CT volumes have a 12 bit intensity resolution, stored in 16 bit integer representations. Also CT uses Hounsfield Units, which range between -1024 and 3071, and one has to shift this range to a non-negative one in order to compute the integral volume.

The integral volume ii contains at each location (x, y, z) the sum of the original intensity values i from the origin $(0, 0, 0)$ up to and including (x, y, z) according to

$$ii(x, y, z) = \sum_{x'=0}^{x} \sum_{y'=0}^{y} \sum_{z'=0}^{z} i(x', y', z').$$

It can be computed using the recursion

$$\begin{aligned} ii(x,y,z) = & i(x,y,z) + ii(x-1,y,z) + ii(x,y-1,z) \\ & + ii(x,y,z-1) - ii(x-1,y-1,z) \\ & - ii(x-1,y,z-1) - ii(x,y-1,z-1) \\ & + ii(x-1,y-1,z-1), \end{aligned}$$

with the borders of the volume equal to 0 in order to deal with index locations $x, y, z = -1$.

From eight values of the integral volume, the sum of the original intensities in an arbitrarily sized region can be computed, i.e. a box filter can be calculated in constant time. So the mean μ_R of voxel values i in a region R is

$$\begin{aligned} \mu_R = & \frac{1}{N} \sum_{x=x_0}^{x_1} \sum_{y=y_0}^{y_1} \sum_{z=z_0}^{z_1} i(x,y,z) \\ = & \frac{1}{N} (ii(x_1,y_1,z_1) - ii(x_0-1,y_1,z_1) - ii(x_1,y_0-1,z_1) \\ & - ii(x_1,y_1,z_0-1) + ii(x_0-1,y_0-1,z_1) \\ & + ii(x_0-1,y_1,z_0-1) + ii(x_1,y_0-1,z_0-1) \\ & - ii(x_0-1,y_0-1,z_0-1)). \end{aligned}$$

3.1. Memory Consumption

Contrary to other works we do not use a floating point represention for the integral volume. Due to summing up values, which start from low numbers and monotonically increase to very large numbers, it can not be prevented, that a floating point representation loses precision. Instead we use integers, where a naive implementation of the integral volume requires a 64 bit word length per voxel, since the maximal number that needs to be stored is the product of the maximum extents in the three dimensions and the maximum intensity of a voxel. Due to the monotonically increasing structure of the integral volume, such a naive representation wastes a lot of memory. As an example consider a CT volume of size $512 \times 512 \times 512$ with a 12 bit intensity resolution, giving a maximally possible summed value of $549.755.813.888$ for the integral volume. One needs 39 bits to represent this number, in practice this would be stored in a 64 bit number. To compute a required bit length L_{ii} of an integral volume we use

$$L_{ii} = \lceil log_2((2^{L_i} - 1)WHD + 1) \rceil$$

with L_i the bit length of the input volume and $\lceil \cdot \rceil$ the ceil operator.

3.2. A Memory Efficient Variant

We propose a memory efficient integral volume (MEIV) by combining several techniques. We divide the input volume regularily into 3D blocks (see Fig. 1). One absolute offset is computed per block and this offset is stored in an additional data structure of smaller size. Now for each block we compute the difference to a simple one-parameter prediction model for the integral volume values in this block. Finally the differences to the model prediction are stored voxel wise using a bitset with dynamic per block word length. Of course with this representation we trade-off memory consumption for computation, since evaluating the prediction model and bitset access require additional computational effort when compared to the naive variant. However, evaluating the integral volume for arbitrarily sized region sums is still performed in constant time, while most of the additional computational effort goes into the initial creation of the MEIV. Since integral volume setup is a preprocessing step that needs to be done only once per input volume, this is a worthwhile compromise. The full MEIV construction is given in Algorithm 2.

3.2.1 Dividing into Blocks

We subdivide the integral volume with an intensity range between $[0, R]$ and of size $W \times H \times D$ into blocks of equal block size B. Thus, we have $M = \lceil \frac{W}{B} \rceil$, $N = \lceil \frac{H}{B} \rceil$, $P = \lceil \frac{D}{B} \rceil$ blocks in the three dimensions, respectively. See Fig. 1 for

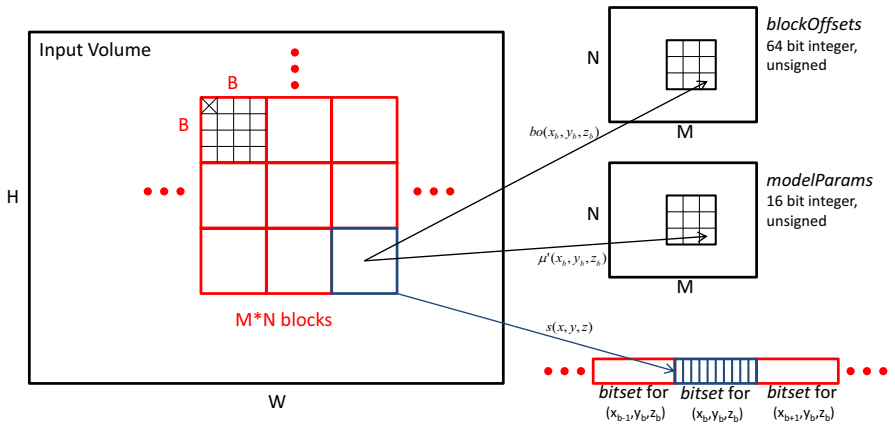


Figure 1. Geometry of memory efficient integral volumes in a 2D illustration. The input volume is split into blocks of size B (red). Block offsets and model parameters are computed and stored in arrays of a size equal to the number of blocks. Differences of the integral volume sums per voxel and the model predictions are stored in the list of bitsets of differing per-block word length.

the volume geometry in a two-dimensional illustration. Our strategy is to loop over all blocks (x_b, y_b, z_b) sequentially, computing the integral volume values $ii(x, y, z)$ solely for the block (using previously computed neighbor blocks), and storing ii into an intermediate 64 bit unsigned integer array of size $B\times B\times B$. Next we treat the first element of the intermediate array (an example is marked by a cross in Fig. 1) as a block offset which is stored in the *blockOffsets* data structure using a 64 bit unsigned integer. All other values of the block may now be represented relatively to this block offset bo by subtracting $bo(x_b, y_b, z_b)$ from the value $ii(x, y, z)$. The block offset is our first mechanism for memory reduction by decreasing the range of possible values in a block. Storing the block offset in a lower resolution $(M\times N\times P)$ array is a form of hierarchical compression of the integral volume, however, by itself it is not enough to reduce the memory consumption sufficiently. This is due to large differences that still occur in late blocks of the integral volume, where still very high numbers of voxels have to be summed up over rows, columns and planes.

3.2.2 One-parameter Model Predictions

After computing bo we continue investigating the current block (x_b, y_b, z_b) by calculating a model estimate predicting ii for the voxels in the block. Since each element of the integral volume is a sum over a given number of input voxels, a very simple one-parameter model is to assume that $ii(x, y, z)$ is approximately $m(x, y, z) = (x+1) * (y+1) * (z+1) * \mu$, where μ is a mean value estimate over the summed voxels. With such a one-parameter model, we can expect that the difference between m and ii is low. To find an optimal μ for a block, we perform an optimization over the possible values of μ, namely the integers in the range $[0, R]$. The optimization performs a binary search (see Algorithm 1) in the range $[0, R]$, with an algorithm that - starting from the full range - divides the range recursively, looking for the mean estimate that minimizes the bit length to store the differences of m and ii. Note that this cost function is V-shaped and contains a single local minimum in the range $[0, R]$. The binary search gives us the optimal μ' which is stored in the *modelParams* array as 16 bit unsigned integer (corresponding to the intensity range of the input volume). We finally combine the block offset bo from the previous step and the mean estimate μ' into a single model computation according to

$$m(x, y, z) = bo(x, y, z) + \mu' * ((x+1)(y+1)(z+1) - (x_b+1)(y_b+1)(z_b+1)).$$

This gives us a model estimate for the integral volume at (x, y, z) that counts the number of additionally required voxels relative to the $(0, 0, 0)$ location of the block and multiplies it with the mean estimate. The value that we finally store in our dynamic word length storage is the difference between the model estimate and the integral volume value $s(x, y, z) = m(x, y, z) - ii(x, y, z)$.

3.2.3 Dynamic Word Length Storage

Our last step to reduce memory consumption is to store the differences $s = m - ii$ from the previous step with differing word lengths per block. This is intuitive, since early blocks in the computation require smaller bit lengths due to much lower sums than blocks near the end of the integral volume. From the smallest stored value range svr (see Algorithm 1) we can compute this bit length, L_B, and compactly store the values of a block in a bitset of size $B^3 L_B$. In our C++ implementation we use the Boost [2] dynamic bitset for this purpose. Its low memory overhead and efficient bit access

[2] http://www.boost.org

Algorithm 1 Binary Search Model Estimates

Require: intermediateBlock
Ensure: μ' is optimal model parameter
  push μ range $[0, R]$ onto queue Q
  while Q not empty **do**
    μ range $[l, h]$ = Q.pop()
    compute min & max values for $m(l) - s$ in block
    stored value range $svr_L := max - min$
    compute min & max values for $m(h) - s$ in block
    stored value range $svr_H := max - min$
    if $svr_L < svr_H$ **then**
      $\mu_{nxt} := l + \lfloor \left(0.5 + \frac{svr_L}{svr_L + svr_H} (h - l)\right) \rfloor$
    else
      $\mu_{nxt} := l + \lfloor \left(0.5 + \left(1 - \frac{svr_H}{svr_L + svr_H}\right) (h - l)\right) \rfloor$
    compute stored value range svr_N of μ_{nxt}
    compute stored value range svr_{N-1} of $\mu_{nxt} - 1$
    compute stored value range svr_{N+1} of $\mu_{nxt} + 1$
    if $svr_N \leq svr_{N-1}$ and $svr_N \leq svr_{N+1}$ **then**
      converged $\mu' := \mu_{nxt}$
    else
      if $svr_{N-1} > svr_N$ and $svr_N > svr_{N+1}$ **then**
        push μ range $[\mu_{nxt}, h]$ onto Q
      if $svr_{N-1} < svr_N$ and $svr_N < svr_{N+1}$ **then**
        push μ range $[l, \mu_{nxt}]$ onto Q

has been shown in [16] in comparison with other bitset libraries, therefore it is very well suited for our purpose.

Algorithm 2 MEIV Computation

Require: input volume $i(x, y, z)$
Ensure: $ii(x, y, z)$ is the integral volume of i
  for all blocks (x_b, y_b, z_b) of i **do**
    compute intermediate block $ii_B(x_b, y_b, z_b)$ of size B^3
    store $bo := ii_B(0, 0, 0)$ in *blockOffsets*
    determine best model mean estimate μ' (Algo. 1)
    for each voxel xx, yy, zz in ii_B **do**
      $s(xx, yy, zz) := m(xx, yy, zz; \mu') - ii(xx, yy, zz)$
      store s in dynamic bitset for current block

The total memory requirement of the MEIV is $O(MNP{*}B^3{*}L_B)$, thus it is highly dependent on the block size B. We can see that MEIV access for region evaluations is still possible in constant time, with a slight overhead for reading block offset and model parameter, accessing the stored value from the bitset, computing the model, and finally subtracting the stored value from the model computation result.

4. Random Forest Object Detection

We apply the MEIV data structure of Section 3.2 for Random Forest based 3D object detection, suited to locate anatomical structures from medical volumetric data. The Random Forest is very similar to the work in Criminisi et al. [5], that proposes a regression model on the distance of voxels to the annotated bounding box of an object of interest. We reproduce the main steps of their algorithm for self-containedness. Also we stress some design decisions of our implementation, that lead to a low memory consumption.

4.1. Forest Training

The forest training starts with loading all input volumes and computing the MEIV. A forest consists of T decision trees. For each $t \in T$ we apply the tree training, which is a depth-first procedure over the nodes of the decision tree. Tree training commences at the root node by randomly creating a pool of ρ features, specified as one rectangular region sum or the subtraction of two region sums of random size(s) and located at random offset(s). The decision if one or two regions are used depends on a coin flip. For each feature a set of thresholds is created, again in a random fashion. Each combination of a feature with one of its thresholds will be denoted as F. Now we compute the features on the whole training data set, i.e. all voxels of the training set, which have reached the current tree node (for the root node these are all available voxels). Feature computation makes heavy use of the MEIV representation for computing the region sums, a procedure that is repeated billions of time during training. Rather than storing the feature response for one computed feature for all voxels, we store the comparison of the feature response with all possible thresholds in a bitset, which has the same size as the number of training voxels. For each tested threshold we need a different bitset, but for small numbers of thresholds this is a lot more memory efficient than having to store each feature response, and significantly faster than recomputing the feature response several over and over again. Next we go over the set of thresholds and create the two subsets of the training voxels for left and right child nodes (the potential split). The two subsets are used to compute the information gain of the regression model on the distances to the six faces of the annotated object from the training data set. The threshold that gives the largest information gain over the current node's entropy is kept for the currently regarded feature. The feature-threshold pair that maximizes the information gain over the feature pool is kept as the final result of the regarded node. The subsets corresponding to this feature-threshold pair are used to split the current node and provide the input for the recursive node training procedure. Recursion stops as soon as a maximum tree depth T_D has been reached, the size of a subset of training voxels for a node goes below 25 voxels, or the information gain of a split is zero. Nodes where the recursion has stopped are treated differently from split nodes. In split nodes we store the feature-threshold pair F, while the leaf nodes store the mean and the

variance of the location of the six faces of the bounding box the node is voting for. Note that in our implementation we use a queue to implement the recursion. This way we have better control of the memory consumption during training.

4.2. Forest Evaluation

After integral volume computation for the test volume i_t, the forest evaluation loops over all voxels of i_t and evaluates each tree independently. The nodes are followed by applying the computation of the feature stored in the node, comparing with the stored threshold and continuing with left or right child node depending on the comparison. Feature computation again makes use of the MEIV to evaluate the features in constant time. For each tree the nodes are followed until a leaf node is reached. The mean distance and its variance to each of the six faces of a bounding box are used to create a vote for the object bounding box. Votes are averaged over the different trees, thus generating a vote for a final object location.

5. Experiments & Results

We perform two kinds of experiments, firstly the novel integral volume is investigated in terms of memory consumption and computational effort. Secondly the MEIV is used in a random forest object detection framework on a data set of liver CT volumes. All our experiments are performed on a work-station with 4 hyper-threaded Intel Core I7 930 (2.8 GHz) and 12 GB of RAM. The operating system is Linux, implementations are done in C++ using the gcc compiler and -O3 optimizations.

5.1. Integral Volume Performance

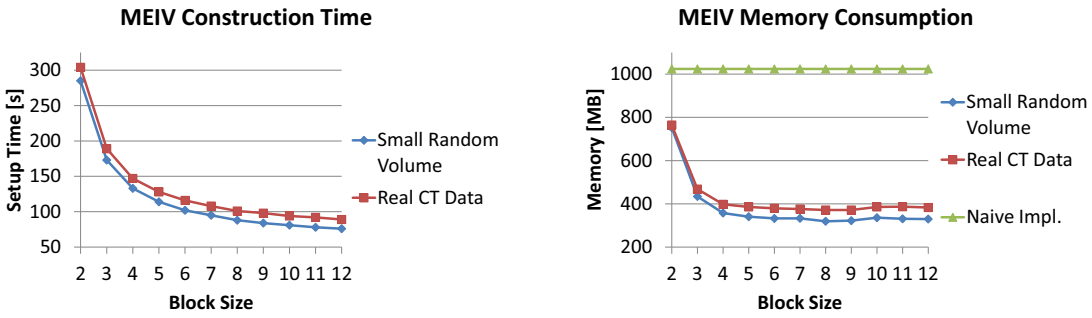
To demonstrate the performance of the proposed MEIV data structure, we perform experiments on a 3D volume of size $512\times512\times512$ with uniformly distributed random values from the range $[0, 1023]$ (*smallRandomVolume*), a real-world CT image of the thorax of size $512\times512\times476$, where the intensity range was rescaled to $[0, 1023]$ (*realCTVolume*), and a 3D volume of size $1024 \times 1024 \times 1024$, with random values in the range $[0, 512]$ (*largeRandomVolume*). The last data set serves as an upper limit for current practically used volumetric data. Our experiments compare the naive integral volume implementation with MEIV by computing both representations (in case of the random valued data, integral volume setup is repeated ten times and measurements are averaged), and afterwards computing 10000 randomly drawn region evaluations, that may range from a size of one voxel to the full size of the volume. Further, we vary the block size B of the MEIV in the range $[2, 12]$ to determine a good compromise between setup time, memory consumption and region evaluation time. The results of the setup time and memory consumption for *smallRandomVolume* and *realCTVolume* can be found in Fig. 2. For a block size $B=8$ the *smallRandomVolume* requires 319 MB, which is 31% of the naive implementation. This is consistent with the theoretical memory storage of the MEIV, given by $O(MNP*B^3*L_B)$, with $L_B = 19.5$ in this case. Note that in addition to the size of the bitset per block $B^3 L_B$, we need to store bo, μ' and some index variables per block as well, these are not visible in the big-O-notation result. The same evaluation for *largeRandomVolume* can be found in Table 1, the naive integral volume implementation for *largeRandomVolume* requires 8192 MB RAM in this case. For comparison the baseline setup time of the naive integral volume implementation is 9.2 seconds for *largeRandomVolume* and around 1.1 seconds for the two smaller data sets, respectively. After testing the integral volume access, it could be seen that the mean of the region evaluation times is nearly constant over varying block sizes, these results are shown in Table 2.

Access Time	*smallRndm*	*CTVolume*	*largeRndm*
naive [ms]	0.0017	0.0012	0.0017
MEIV [ms]	0.0057	0.0055	0.0068
no IV [ms]	2.1	2	15.7

Table 2. Mean region evaluation times in milliseconds from 10000 randomly drawn regions per volume.

5.2. CT Liver Detection

We use the MEIV with a block size $B = 8$ to train a random forest for object detection using a database of 30 abdominal CT volumes from the MICCAI Liver Segmentation Grand Challenge data set [11]. We use manually annotated axis aligned bounding boxes around the liver, and train a Random Forest according to Section 4. The 30 volumes have an in-plane resolution of 512×512 voxel, and the number of slices ranges between 64 and 502, with a mean of 214 slices. Physical resolution is around 0.6 mm in-plane and ranges between 0.7 and 5 mm slice thickness. The mean memory consumption of the 30 volumes represented as 8 byte integral volumes is 428 MB, leading to a total of 12840 MB for holding the full training data set in memory using a naive integral volume implementation. Note that even without the overhead of the Random Forest training, this does not fit into the RAM of our work-station,and constantly loading the integral volumes from a file system would be a bottleneck significantly slowing down training. The Random Forest is trained sequentially using $T = 10$ trees with a maximum depth of $D = 7$, and a feature pool of $\rho = 40$ random features per node that are compared against three thresholds. With this setup we require eight additional data structures in the form of bitsets to store node splits and feature-threshold comparisons. Each of these bitsets has a size equal to the full number of training data voxels resulting in 200 MB per bitset, and 1600 MB in total for train-



(a) (b)

Figure 2. MEIV evaluation over different block sizes. (a) Setup time is slightly higher for real CT data, for block sizes larger than 8 the time is below 100 seconds. (b) Memory consumption is smallest for a block size of 8, around 35% of the naive integral volume implementation's consumption. Again real CT data requires slightly more memory due to the non-Gaussian distribution of intensities.

block size	3	4	5	6	7	8	9	10	11	12
setup time [s]	1273	1022	888	797	743	**699**	667	642	629	608
memory [MB]	3042	2694	2599	2570	2574	**2544**	2561	2597	2636	2631

Table 1. *largeRandomVolume* results over varying block sizes. The naive integral volume implementation requires 8192 MB of RAM and has a setup time of 9.2 seconds. For a block size of eight, a 31% reduction in storage size can be achieved.

ing. We want to stress here, that we do not require a down-sampling of the training data sets. In practice it is necessary to perform the feature computation and threshold evaluation on a subset of the training data to reduce computation time, resembling an internal bagging step with a node depending subset size, i.e. smaller subsets near the root node. However, after the best feature for a node has been identified on the subset the feature calculation is repeated for the full training data reaching the node to create the node split.

We have performed a leave-one-out experiment using our training data set, where we have performed five rounds of training from 25 randomly drawn CT volumes from the training set, and have tested on the remaining five volumes for the overlap of the six faces of the liver bounding box annotations. The result of this experiment was a mean bounding box localization error of 24 mm with a standard deviation of 19 mm. Forest training using the described setup still takes a long time. Our sequential implementation requires around nine days to train a forest from 25 data sets using the full input volumes. This resembles a mean per-node computation time of one minute for the MEIV. The three times higher access time to evaluate the MEIV compared to the naive implementation (see Table 2) did not translate to a three times higher per node computation effort, in practice the difference was a factor of 1.2.

6. Discussion

From our experiments we can clearly see the benefits of our proposed MEIV data structure in terms of memory consumption. The main problem of the naive integral volume implementation, the cubic dependency of the computed sums on the volume size, is reduced, leaving the MEIV with less than a third of the required memory with the right choice of block size. This behaviour is consistent with the theoretical storage requirements given our implementation. We see in Fig. 2 that a block size $B = 8$ leads to a reasonable compromise between setup time (which is not very relevant, since this computation has to be performed only once in typical applications) and memory consumption. With the MEIV it is thus possible to increase the number of full resolution training data sets in a random forest object detection application by a factor of three given a certain computational setup with a limited amount of RAM. This property scales to larger volumetric data sets, and is also applicable for computing servers with much larger amounts of main memory. We again want to stress that we find it relevant to prevent down-sampling of input data, so that a weak learner based algorithm is able to use all of the available information in the training set.

The liver object detection results are worse than in Criminisi et al. [5], however, they have used a different, much larger database of CT volumes, so a direct comparison is hard. They had to use a compute server and have down-sampled their input volumes for training. It would be very interesting to test our framework on their data to investigate further the benefits of being able to train on the originally sized input volumes. We want to note here, that the focus of our proposed work is not on improving on existing object detection accuracy, but to provide a way to prevent down-

sampling in forest training using our memory efficient integral volume, to be able to use all available information from a training set.

7. Conclusion and Outlook

In this work we have proposed MEIV, a memory efficient integral volume implementation that allows a storage size reduction to less than a third of the naive variant. We have given a first glimpse on how this technique may be used in a weak learner ensemble based machine learning approach for 3D object detection, using 3D training sets that lead to a huge memory consumption. The use of MEIV is of course not restricted to Random Forest based object detection, but more generally applicable. Also it is trivial to implement the technique for 2D integral images, this would help in applications for embedded devices where memory is scarce, and therefore storage of the integral data structure of the test image has to be optimized in terms of size and bandwidth.

Future work will investigate the 3D object detection framework more deeply on a more comprehensive data set. Further, the application to a segmentation framework is even more interesting, although more demanding. Finally we will investigate how to speed up parts of the forest training using CUDA for highly parallelized feature computation on the GPU.

References

[1] H. J. W. Belt. Storage Size Reduction for the Integral Image. Technical Report TN-2007/00784, Philips Research Eindhoven, 2007. 2

[2] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007. 1

[3] L. Breiman. Random Forests. *Machine Learning*, 45:5–32, 2001. 1

[4] H. Chen, T. Liu, and C. Fuh. Segmenting highly articulated video objects with weak-prior random forests. In *Proc. European Conference Computer Vision (ECCV)*, 2006. 2

[5] A. Criminisi, D. Robertson, E. Konukoglu, J. Shotton, S. Pathak, S. White, and K. Siddiqui. Regression forests for efficient anatomy detection and localization in computed tomography scans. *Medical Image Analysis*, in press, 2013. 2, 5, 7

[6] A. Criminisi and J. Shotton, editors. *Descision Forests for Computer Vision and Medical Image Analysis*. Springer, 2013. 1

[7] F. C. Crow. Summed-area tables for texture mapping. *SIGGRAPH Comput. Graph.*, 18(3):207–212, 1984. 2

[8] R. Donner, B. H. Menze, H. Bischof, and G. Langs. Global localization of 3D anatomical structures by pre-filtered Hough Forests and discrete optimization. *Medical Image Analysis*, in press, 2013. 2

[9] J. Gall, A. Yao, N. Razavi, L. van Gool, and V. Lempitsky. Hough Forests for Object Detection, Tracking, and Action Recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(11):2188–2201, 2011. 2

[10] A. S. Glassner. *Graphics Gems*, chapter VI.7: Multidimensional Sum Tables, pages 376–381. Academic Press, 1990. 2

[11] T. Heimann, B. van Ginneken, M. Styner, Y. Arzhaeva, V. Aurich, C. Bauer, A. Beck, C. Becker, R. Beichel, G. Bekes, F. Bello, G. Binnig, H. Bischof, A. Bornik, M. M. Cashman, Y. Chi, A. Cordova, M. Dawant, M. Fidrich, D. Furst, D. Furukawa, L. Grenacher, J. Hornegger, D. Kainmuller, I. Kitney, H. Kobatake, H. Lamecker, T. Lange, J. Lee, B. Lennon, R. Li, S. Li, H. Meinzer, G. Nemeth, S. Raicu, A. Rau, M. van Rikxoort, M. Rousson, L. Rusko, A. Saddi, G. Schmidt, D. Seghers, A. Shimizu, P. Slagmolen, E. Sorantin, G. Soza, R. Susomboon, M. Waite, A. Wimmer, and I. Wolf. Comparison and Evaluation of Methods for Liver Segmentation From CT Datasets. *IEEE Transactions on Medical Imaging*, 28(8):1251–1265, 2009. 6

[12] 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. 2

[13] A. Montillo, J. Shotton, J. Winn, J. E. Iglesias, D. Metaxas, and A. Criminisi. Entangled Decision Forests and their Application for Semantic Segmentation of CT Images. In *Proc Information Processing in Medical Imaging (IPMI)*, 2011. 2

[14] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe. GPU-Efficient Recursive Filtering and Summed-Area Tables. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH Asia*, 30(6):176:1–176:12, 2011. 2

[15] M. Oezuysal, M. Calonder, V. Lepetit, and P. Fua. Fast Keypoint Recognition Using Random Ferns. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(3):448–461, 2010. 1

[16] V. Pieterse, D. G. Kourie, L. Cleophas, and B. W. Watson. Performance of C++ bit-vector implementations. In *Proc Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '10)*, pages 242–250. ACM, 2010. 5

[17] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *Annals of Statistics*, 26(5):1651–1686, 1998. 1, 2

[18] J. Shotton, M. Johnson, and R. Cipolla. Semantic texton forests for image categorization and segmentation. In *Proc. IEEE Conf Computer Vision Pattern Recognition (CVPR)*, 2008. 2

[19] Z. Tu, X. Zhou, D. Comaniciu, and L. Bogoni. A Learning Based Approach for 3D Segmentation and Colon Detagging. In *Proc. European Conference Computer Vision (ECCV)*, 2006. 2

[20] P. Viola and M. J. Jones. Robust Real-Time Face Detection. *International Journal of Computer Vision*, 57(2):137–154, 2004. 1, 2

[21] Y. Zheng, B. Georgescu, H. Ling, S. K. Zhou, M. Scheuring, and D. Comaniciu. Constrained marginal space learning for efficient 3D anatomical structure detection in medical images. In *IEEE Conf. Computer Vision and Pattern Recognition*, 2009. 2