Fast Boosting Trees for Classification, Pose Detection, and Boundary Detection on a GPU

Neil Birkbeck† Michal Sofka‡ S.Kevin Zhou‡
†Department of Computing Science
University of Alberta, Canada
‡ Siemens Corporate Research
775 College Road East, Princeton NJ, USA

Abstract

Discriminative classifiers are often the computational bottleneck in medical imaging applications such as foreground/background classification, 3D pose detection, and boundary delineation. To overcome this bottleneck, we propose a fast technique based on boosting tree classifiers adapted for GPU computation. Unlike standard tree-based algorithms, our method does not have any recursive calls which makes it GPU-friendly. The algorithm is integrated into an optimized Hierarchical Detection Network (HDN) for 3D pose detection and boundary detection in 3D medical images. On desktop GPUs, we demonstrate an $80\times$ *speedup in simple classification of Liver in MRI volumes, and* $30\times$ *speedup in multi-object localization of fetal head structures in ultrasound images, and* $10\times$ *speedup on 2.49 mm accurate Liver boundary detection in MRI.*

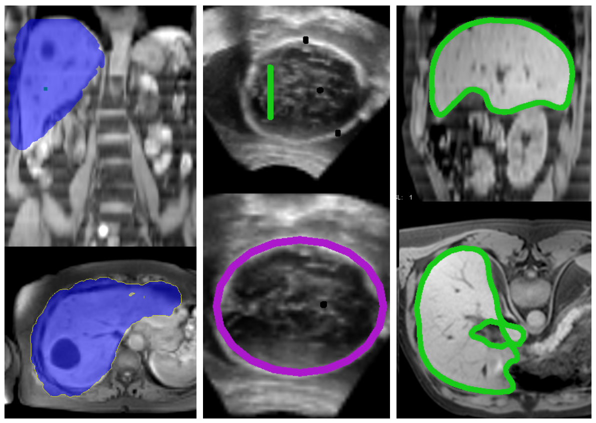
Figure 1. Discriminative boosting tree classifiers are applicable to a range of applications: naive per-pixel classification of Liver in MRI (left), pose detection of fetal head structures in ultrasound (middle), to accurate liver boundary detection in MRI (right).

1. Introduction

Efficiency of algorithms for automatic detection and segmentation is of central importance in many medical imaging applications, especially when these algorithms are mapped to embedded and mobile devices with fewer computational resources. Several fast and robust algorithms for detection and segmentation are based on machine learning techniques such as random forests [11], cascades of Adaboost classifiers [14], and boosting trees [13]. These algorithms have been accelerated by hierarchical methods [12] and by reducing the set of hypotheses the classifier needs to test [16]. In addition to these optimizations, adapting these algorithms for a GPU computation is becoming more attractive since the graphics hardware on desktop computers (e.g., nVidia 480 GTX), embedded (e.g., Radeon E4690 MXM), and mobile devices (e.g., nVidia Tegra 3) is becoming more general purpose. Yet, mapping algorithms to the GPU is often nontrivial because of the challenge to distribute the processing into many parallel tasks to achieve the highest speedups.

In this work, we accelerate detection and segmentation algorithms based on Probabilistic Boosting Trees (PBT) [13] by taking advantage of GPU processing power. Boosting trees are binary discriminative classifiers trained using a large annotated database of images. Each node of a boosting tree is a strong classifier (AdaBoost) and the whole tree tests each pixel for presence of an object (in detection or pixelwise segmentation) or for presence of an object boundary (in object segmentation). See Fig. 1 for examples. The decision at each node is probabilistic, i.e., both tree branches are descended when the strong classifier confidence is low. Traditionally, tree-based algorithms are implemented with recursive calls that are not possible on a GPU. We propose a stack-based technique to enable a GPU implementation. Features used by the classifier at each node are computed on the GPU, ensuring data transfer to the GPU is minimized.

The parallel power of the GPU has already been exploited for machine learning algorithms, including boosting [1, 3] and random forests [11]. Many of the implementations are application specific, with applications ranging from 2D object detection [5, 15] and motion gesture recognition [1] to classification [11]. The speedups depend on application, generation of graphics card, and also reference CPU implementation, with reported speedups being lower for already optimized cascaded detection [5].

Unlike trees in random forests [11, 4], the Probabilistic Boosting Tree is more general with the following characteristics: (1) boosting on the decision nodes, (2) support for cascading, and (3) fuzzy decision at each node. In contrast to many existing approaches, our focus is on 3D images which is harder due to larger data sets that need to be transferred to the graphics card. Furthermore, the existing approaches typically present only a single feature type (e.g., one of ZNCC [3], LRD [7], and Haar-like features [11]). Our framework supports multiple feature types, the ability to add new features, and is integrated into an optimized hierarchical 3D detection pipeline.

Our technique is built on a stack-based transformation of the PBT that is suitable for GPU implementations (Section 2). The boosting tree data structure is mapped to a texture for efficient access that allows for re-use of tree evaluation code with different features and makes a shader-based implementation feasible (Section 3). The GPU-PBT implementation is integrated into a multi-structure hierarchical detection framework where results are sorted on the GPU to reduce data transfer. On desktop GPUs, we demonstrate $30\times$ speedups for automatic fetal head structure detection, and we obtain $10\times$ speedup for accurate (2.49 mm) Liver MRI boundary detection. Also, in simpler classification tasks, speedups of $78\times$ are possible (Section 4).

2. Discriminative classifiers in Medical Imaging Applications

Many medical imaging algorithms are intimately coupled to the output of a discriminative model. For example, the PBT, has been successfully deployed as the classifier in a variety of medical imaging applications, including automatic measurement of fetal brain structures [12] and automatic organ detection and segmentation (e.g., liver [9] and heart [16]). The following list gives a few examples of how these discriminative models are used in such imaging applications (Fig. 1):

Classification: In its most direct use, discriminative classifiers can be trained to detect anatomical structures on a per-pixel basis (e.g., as in [8]).

Single object localization: Object localization can be efficiently performed by decomposing the 9-dimensional search space of similarity transformations into a series of smaller search spaces [16]. During detection, the discriminative classifier is evaluated on pixels in the volume to determine a small set (e.g., 100-1000) of candidate positions. Orientation is detected by evaluating each of these positions with a set of hypothesis orientations and similarly for scale. The final list of candidates is aggregated to obtain a single pose estimate.

Coupled object localization: Joint multi-object detection is decomposed into sequential single-object detections with spatial priors used for prediction of dependent structures [12]. Thus composed Hierarchical Detection Network (HDN) consists of a network of nodes for detecting position, orientation, and scale, for each structure; spatial dependencies are represented as arcs.

Boundary detection: Given an initialization of a triangulated mesh (e.g., a mean mesh placed at a detected location), a discriminative model can be trained to identify the boundary of the structure [9]. The mesh surface is iteratively deformed along the normal in order to maximize the value of the classifier evaluated at the surface points.

Each of these problems are computationally dependent on the discriminative classifier. Although the features used in the algorithms may be different, each application evaluates the same classifier on several pixel locations (in classification), several candidate poses (in pose detection), or several mesh displacements (in boundary detection). These data parallel problems can be efficiently evaluated on the GPU.

2.1. Probabilistic Boosting Tree Classifier

The Probabilistic Boosting Tree (PBT) models the posterior distribution of a data set [13], enabling its uses as a discriminative model for all of the above listed applications. A PBT is a binary decision tree with a fuzzy decision taken at each internal node of the tree depending on the output of the node's strong classifier. Each node, N, contains a strong classifier, $\hat{q}_N(y|x)$, and the empirical distribution of its leaf nodes $q_N(y)$, where $y \in \{-1, +1\}$, and x is an input point. The posterior value of a node is determined by recursively combining the posterior values of its children nodes; the weight used in the combination comes from evaluating the node's strong classifier, $\hat{q}_N(y|x)$.

The strong classifier can be any classifier that uses any problem specific feature. We use an AdaBoost classifier for the strong classifier, which combines several binary weak classifiers to produce a strong estimate.

When evaluating a PBT node, the value of the strong classifier, $\hat{q}_N(+1|x)$, determines which children of the node will be descended. If the strong classifier is fairly certain (e.g., $\hat{q}_N(+1|x) \geq 1-\epsilon_1$ or $\hat{q}_N(-1|x) = (1-\hat{q}_N(+1|x)) \leq \epsilon_1$), then only one child is descended. However, if the value is close to 0.5, then recursive calls are made to both children. The algorithm is highlighted in the pseudo-code of Algorithm 1, where we assume that the context necessary to evaluate the posterior is available to the classifiers. The constants are set as $e_1 < 1^{-6}$ and $e_2 = 0.1$.

Non-recursive Probabilistic Boosting Tree: As the GPU does not support recursive calls, we need to remove the recursion from Algorithm 1. If this were a simple decision

Algorithm 1: PbtPosterior

```
Data: $N$ node of the tree
Data: $L$ = left($N$), $R$ = right($N$)
Result: $p_N(+1|\mathbf{x})$ the posterior prob. for subtree $N$
if Leaf($N$) then return $q_N(+1)$
$p = \hat{q}_N(+1|\mathbf{x})$
if $p > (1 - e_1)$ then return PbtPosterior($R$)
else if $p < e_1$ then return PbtPosterior($L$)
else if $p > 0.5 + e_2$ then
    return $(1 - p)q_L(+1) + p$PbtPosterior($R$)
else if $p < 0.5 - e_2$ then
    return $(1 - p)$PbtPosterior($L$)+$p\ q_R(+1)$
else
    return $(1 - p)$PbtPosterior($L$)+$p$ PbtPosterior($R$)
```

Algorithm 2: PbtPosteriorStackBased

```
Data: $N$ node of the tree
Result: $p(+1|\mathbf{x})$ the posterior of the tree
S = $\{\langle root, 1.0\rangle\}$
total $\leftarrow$ 0
while $\| S \| \neq 0$ do
    $\langle N, w\rangle$ = pop(S)
    if isLeaf($N$) then
        total $\leftarrow$ total + $w * q_N(+1)$
        continue;
    p $\leftarrow \hat{q}_N(+1, x)$
    if $p > (1 - e_1)$ then
        S.push($\langle$ right(N), $w\rangle$)
    else if $p < e_1$ then
        S.push($\langle$ left(N), $w\rangle$)
    else if $p > 0.5 + e_2$ then
        $L$ = left($N$)
        total = total + $w * (1 - p)q_L(+1)$
        S.push( $\langle$right($N$), $w * p\rangle$)
    else if $p < 0.5 - e_2$ then
        $R$ = right($N$)
        total = total + $w * p * q_R(+1)$
        s.push($\langle$left($N$), $w * (1 - p)\rangle$)
    else
        // Descend down both nodes
        S.push( $\langle$right($N$), $w * p\rangle$ )
        S.push( $\langle$left($N$), $w * (1 - p)\rangle$ )
return total;
```

tree evaluation with one recursive call, then the single recursive call could be replaced with an iterative descent.

In the PBT, the base case of the recursion simply returns the empirical distribution of the node, $q_N(+1)$, and the results of recursive calls are combined with a convex combination of the strong classifier value. As such, the final result is just a sum of weighted empirical distributions of the nodes. The total weight given to any node's empirical distribution is the product of the weights associated with the path from the root to the node. Therefore, a non-recursive implementation uses a stack that holds a list of to-be traversed nodes and their respective weights. When a leaf node is visited, it simply adds its empirical distribution with the weight to the accumulator (Algorithm 2).

3. Cuda PBT implementation

In the Cuda programming model [10], a parallel problem is decomposed into a grid of thread blocks, with each block containing many threads. Blocks are assigned to the GPU's multi-processors, which breaks down the block and schedules it in groups of 32 threads. In detection or classification, the grid of thread blocks overlay the input (e.g., all pixels in the volume), and each thread evaluates the classifier for a different voxel or a different orientation/scale hypothesis.

The memory architecture includes a 16kb low-latency shared memory (accessible from threads within the same block), high-latency local memory (per-thread) and global memories (accessible by all threads), and cached global accesses through texture memory. Use of these memory types appropriately is key to efficiency.

3.1. Memory Layout

On a GPU implementation, several threads will be descending the tree in parallel. Neighboring threads may access the tree data structure in different regions, meaning using global memory to store the tree would cause slow uncoalesced memory access. The tree data structure is too large to fit in the 16kb shared memory; therefore, to guarantee efficient/cached access to the tree data structure, we arrange the tree data structure in a texture image (see Fig. 3). Node pointers are replaced with 2D location indices that reference the node's 2D region in the texture image as in [11].

In our case, the node data contains the AdaBoost classifiers, which are made up of several weak classifiers. Each of which contains either a simple threshold classifier or a histogram classifier whose decision is made on a single feature. Therefore, the PBT node data in the texture image needs to store the sum of alpha values (for AdaBoost), the number of weak classifiers, and the data for each of the classifiers. These weak classifiers are stored in adjacent columns of the texture (Fig. 3) and each column contains the associated weight (α_i). The relevant data for the weak classifiers depends on the features used by the PBT (§3.2).

The transformation from Algorithm 2 to Cuda is then straightforward (Fig. 2). The stack used to replace the recursion is local to each thread, and each thread has 16kb of local memory (in Cuda 1.1-1.3). Using `float4` for ele-

```
template
  <float feature_func(float3 pos, float nx, float ny)>
__device__ float
pbt_evaluate_tree(const float3& pos) {
  float4 stack[kMaxQueueSize];
  int        numInStack = 1;
  // Insert root (@ 0,0) with weight 1.
  stack[0] = float4_make(0, 0, 0, 1.0);
  while (numInStack > 0) {
    --numInQueue;
    float4 node = stack[numInQueue];
    float prob =
      pbt_eval_classifier
        <feature_func>(pos, node.x, node.y);
    if (prob > 1.0 - e1)
    ... // Analogous to Algorithm 2; Nodes are ↩
          inserted by position
  }
}
```

Figure 2. The Cuda implementation of the stack-based Algorithm 2. Templating the PBT evaluation function on the feature evaluation function allows re-use of the higher level PBT code by simply re-writing a new `feature_func`.

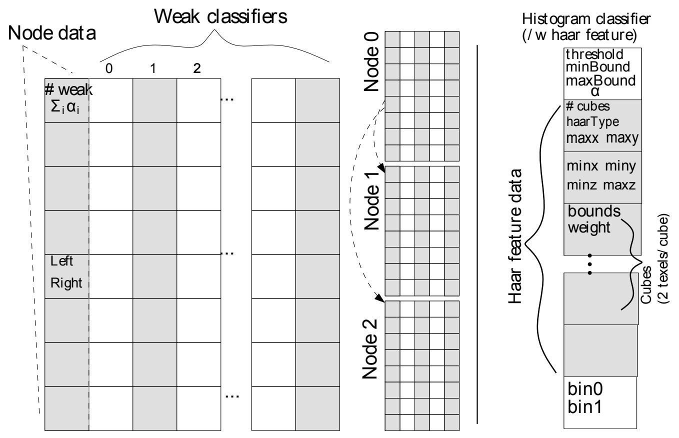

Figure 3. The PBT data structure is stored in a texture. Left: weak classifiers are arranged along the columns, and the PBT nodes store 2D texture indices to their children. Right: layout for Haar features with a histogram classifier. The 64-bin histogram is stored in the 64-bits of two float components.

ments of the stack implies a maximum of 1024 elements in the stack. However, the traversal of the tree is depth-first, so the stack only needs to be as large as the depth of the tree. We limit $kMaxQueueSize = 32$, which is more than enough for our shallow trees (e.g., depth 10 or less).

The Cuda PBT evaluation function is templated on a `feature_func`, allowing new feature types to be added. Both the 3D sample box and the 2D texture location of the feature data are passed to the feature evaluation function.

3.2. Features

During detection or classification the PBT tree is evaluated with the context surrounding a specific location (and pose). Position detection use Haar features; orientation/scale, and boundary detection use steerable features [16].

$$I, \sqrt{I}, I^{0.333}, I^2, I^3, \max(0.001, log(I)), I_x, I_y, I_z$$
$$|\nabla I|^2$$
$$f = \langle|\nabla I|, \mathbf{d}\rangle, |f|, \sqrt{|f|}, f^2, |f|^3, \log(\max(10^{-3}, |f|))$$
$$\sqrt{|\nabla I|^2 - f^2}$$
$$\theta = \cos^{-1}(\langle\frac{\nabla I}{|\nabla I|}, \mathbf{d}\rangle), \theta^2, \theta^3, \log(\max(10^{-3}, \theta))$$

Table 1. Steerable features used in our system: image intensity, I, $\nabla I = [I_x, I_y, I_z]$, and $\mathbf{d}$ is an input sample direction.

Haar features are weighted combinations of the sums of cubic regions of an image. These sums are efficiently calculated using the integral image. Our features use no more than 4 cubes. The gamut of features possible for a given location consists of various translated and scaled versions of these boxes. Each possible feature can be described by the size, weights, and locations of these cubes (relative to a testing point). Fig. 3 illustrates how this information is encoded into a column of the texture for a histogram classifier.

Evaluation on the GPU is then straightforward: we simply look up the cubes for the feature, evaluate the integral image, and combine the results. Our 3D computation of the integral image is based on a previous technique for computing 2D integral images [6].

Steerable features consist of 24 basic types (Table 1) [16]. For a given location, the features are transformations on either the gray value, the gradient, or the projection of the gradient onto an input direction. During pose detection, the input direction comes from the x-axis of the pose, and for boundary detection it is the normal of the mesh point. The possible feature locations lie on a grid of discrete 3D locations around a sample point, and each feature can be calculated on one of several image resolutions of an image pyramid. Each feature is completely described by its position in the 3D sampling pattern (3 integers), its type (1 integer), and the discrete image scale it is computed at (1 integer). By packing the type and scale into a single value (e.g., scale$\times$32 + type), this data can be packed into as few as 1 RGBA pixels of a texture image.

During evaluation, first, the initial datum to be modified (either I, I_x,∇I, etc.) is extracted. This requires a group of 5 conditionals corresponding to the rows of Table 1. Then subsequent modifiers (a power, an absolute value, or a logarithm) are applied. The type of data and the subsequent modifiers are stored as flags in a table of length 24 (similarly for the powers). The multiple scales of the image are stored into a single texture, and an offset table stores the lower left corner of each of the resolutions in this texture.

	Accuracy	CPU1	CPU8	GPU	$\frac{CPU1}{GPU}$	$\frac{CPU8}{GPU}$
PBT	6.4±2.2 mm	24.4s	4.53s	0.31s	78.4×	14.5×
Tree	8.0±2.9 mm	24.9s	4.55s	0.22s	114.9×	21.0×

Table 2. Accuracy and timing results for per-pixel classification in Liver MRI volumes. The *Tree* row gives timings for the non-probabilistic implementation of the boosting tree.

4. Experiments

We have validated our implementation on the applications outlined in Section 2. For evaluation, we used a 1.5GB nVidia 480 GTX on a Intel Core quad with 16GB of RAM. All timing results compare the GPU version to a CPU only version with either a single thread or an 8 thread OpenMP implementation[1] and do not include time to load the volumes from disk. Load times for 1mm resolution volumes, including building the 4mm and 2mm pyramid, takes 0.15-0.35s (resp. 0.05-0.18s) for the single-threaded (resp. multi-threaded) implementation.

4.1. Classification

For classification evaluation, we use Liver annotations on 59 MRI volumes of 3 mm resolution, with sizes of $107 \times 80 \times 60$ to $140 \times 101 \times 144$ voxels. Using three-fold cross-validation, we trained PBT classifiers to depth 6 with Haar features to recognize the inside of human labeled meshes.

During detection, we evaluate the classifier at each pixel, giving $p(x, y, z)$. The resulting probabilities are then regularized to produce binary labels, $l(x, y, z)$, by minimizing: $\int\int\int_V |p(x, y, z) - l(x, y, z)| dV + \int\int_A dA$. The optimization is performed with a discrete graph-cut approximation to the continuous problem [2]. The labels are then converted into a mesh for comparison to ground truth.

Table 2 shows the accuracy compared to ground truth, the average timings for the CPU (single- and 8-threaded) and GPU implementations, and the speedups (see left Fig. 1 for qualitative result). The GPU version achieves a speedup of 78× over the single-threaded CPU implementation.

We also evaluated the non-probabilistic version of the tree ($e_1 = 0.5$ in Algorithm 2) on both the CPU and GPU. This change hardly affects the CPU version, but causes a dramatic change in timing of the GPU implementation. It is possible to achieve a higher speedup in this case (115× over 1-thread CPU), as the Cuda program does not need to maintain the stack. However, this results in worse accuracy.

4.2. Hierarchical PBT detection

The classification results illustrate the potential benefit of using a GPU accelerated PBT evaluation. In such cases there is enough work for the GPU, and the same program is being run on all voxels in an image. We now evaluate the PBT in the context of an already optimized hierarchical detection system [12]. The system is applied to obtain automatic measurements of brain structures in fetal ultrasound. We focus here on detecting three structures: Cerebellum (CER), Cisterna Magna (CM), and Lateral Ventricles (LV). See Figure 4 for a qualitative example. We used 990 volumes with 1 mm resolution and dimensions $143 \times 90 \times 110$ to $231 \times 161 \times 208$ for training. The network encodes spatial relationships between structures: CER is detected first and is used to predict initializations for LV and CM [12]. Detection accuracy for these structures is 2.2 mm.

Table 3 shows the timings and speedups for detection of these 3 structures averaged over 50 volumes. Most of the execution is in orientation and scale detection. The speedups are slightly lower than for classification, which is due to the hierarchical detection algorithm having regions that are inherently serial and must be done on the host CPU (e.g., results are read back, candidates are pruned, dependent structures are predicted). Furthermore, some of the phases of detection evaluate the PBT on as few as 1000s of elements, meaning the GPU is not fully utilized.

[1]The Intel Core quad has hyper-threading, making 8 threads the default.

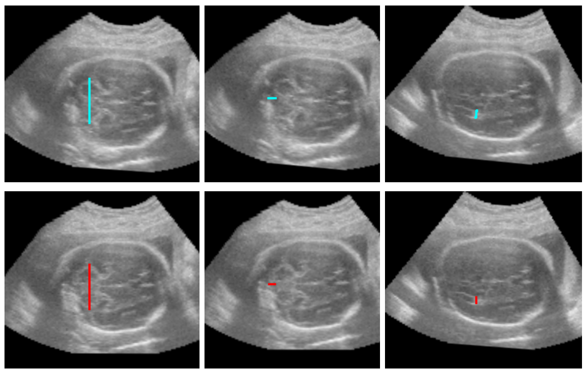
Figure 4. Qualitative illustration of detected results (top) and ground truth (bottom) for fetal head structures in 3D ultrasound. Structures from left: CER, CM, LV.

	CPU1	CPU8	GPU	$\frac{CPU1}{GPU}$	$\frac{CPU8}{GPU}$
CER	12.8s	4.77s	0.77s	16.6×	6.2×
LV	31.1s	7.63s	0.62s	49.9×	12.2×
CM	6.0s	2.60s	0.19s	31.2×	13.3×
Total	49.9s	14.90s	1.60s	31.4×	9.4×

Table 3. Timing for detection of fetal head structures.

	CPU1	CPU8	GPU	$\frac{CPU1}{GPU}$	$\frac{CPU8}{GPU}$
3 mm Pose	4.95s	0.92s	0.43s	11.39×	4.03×
3 mm BndyLow	1.71s	0.26s	0.12s	14.22×	2.11×
3 mm BndyMid	3.41s	0.50s	0.17s	19.79×	2.91×
1 mm BndyHi	10.24s	1.76s	1.01s	10.15×	1.71×
Total	20.41s	3.47s	1.81s	11.30×	2.19×

Table 4. Timing results for MRI Liver boundary detection.

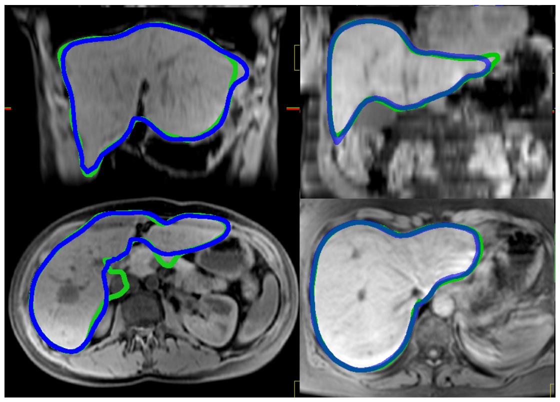
Figure 5. Axial and coronal views of two boundary detection results (blue) overtop the ground truth (green).

4.3. Boundary Detection

After the initial pose detection, the boundary is detected by iterative refinement and smoothing [9]. The data is the same 3 mm MR volumes used in the classification experiment, plus the corresponding 1 mm resolution volumes. With 3-fold cross validation, the boundary detection average mesh-to-mesh accuracy was 2.49 ± 0.85 mm (Fig. 5).

Boundary detection is performed on a mesh hierarchy: the first two resolutions BndyLow (602 vertices) and BndyMid (1202 vert.) are refined on 3 mm volumes; the highest resolution (2402 vert.) is detected on 1 mm. The boundary refinement evaluates the PBT classifier, for each vertex, at several displacements along the normal. The best displacement is chosen for each vertex, and the resulting mesh is smoothed. This two step process is iterated several times. Currently, smoothing is performed on the CPU.

In this case the pose detection also estimates 3-PCA coefficients of the mesh before boundary detection. This part is not optimized and takes roughly 50ms, making the speedups slightly worse (Table 4). The speedup of the overall boundary detection is lower than for classifier evaluation, suggesting that frequent read-back and CPU mesh smooth operations limit the speedup. Specifically, on BndyHi, when evaluating the classifier for 32 displacement samples, the speedups are better: $50\times$ (resp. $7\times$) for a single (resp 8) thread implementation.

5. Conclusion

We have presented an efficient Cuda implementation of the PBT classifier that can be used in classification or detection. Speedups of up to $78\times$ were obtained in classification of Liver in MRI data, to $30\times$ in detection of automatic fetal head detection in ultrasound images, and $10\times$ in MRI Liver boundary detection. In the latter two cases, the optimized detection pipeline makes use of hierarchical pruning.

The use of such optimizations is critical to meet the efficiency demands of current medical imaging applications. Furthermore, the proposed stack-based transformation can be implemented in shader-based GPUs (e.g., mobile devices). With mobile GPU hardware slated to increase efficiency (up to $50\times$ in a few years), such parallel implementations become important for advanced mobile medical imaging solutions.

References

[1] M. Bayazit, A. Couture-Beil, and G. Mori. Real-time motion-based gesture recognition using the GPU. In *IAPR Conf. on Machine Vision Applications (MVA)*, 2009. 1

[2] Y. Boykov and V. Kolmogorov. Computing geodesics and minimal surfaces via graph cuts. In *ICCV*, pages 26–33, 2003. 5

[3] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng. Scalable learning for object detection with GPU hardware. In *IROS*, pages 4287–4293, 2009. 1, 2

[4] A. Criminisi, J. Shotton, and S. Bucciarelli. Decision forests with long-range spatial context for organ localization in ct volumes. In *MICCAI-PMMIA workshop*, 2009. 2

[5] H. Ghorayeb, B. Steux, and C. Laurgeau. Boosted algorithms for visual object detection on graphics processing units. In *ACCV 2006*, pages 254–263. 2006. 1, 2

[6] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, August 2007. 4

[7] A. Herout, R. Josth, P. Zemcik, and M. Hradis. GP-GPU implementation of the local rank differences image feature. In *Computer Vision and Graphics*, pages 380–390. 2009. 2

[8] V. Lempitsky, M. Verhoek, J. A. Noble, and A. Blake. Random forest classification for automatic delineation of myocardium in real-time 3D echocardiography. In *Functional Imaging of the Heart*, pages 447–456, 2009. 2

[9] H. Ling, S. K. Zhou, Y. Zheng, B. Georgescu, M. Suehling, and D. Comaniciu. Hierarchical, learning-based automatic liver segmentation. *CVPR*, 2008. 2, 6

[10] NVIDIA. *NVIDIA CUDA Programming Guide 2.3*. 2009. 3

[11] T. Sharp. Implementing decision trees and forests on a GPU. In *ECCV*, volume 5305, pages 595–608, 2008. 1, 2, 3

[12] M. Sofka, J. Zhang, S. Zhou, and D. Comaniciu. Multiple object detection by sequential Monte Carlo and hierarchical detection network. In *CVPR*, 13–18 June 2010. 1, 2, 5

[13] Z. Tu. Probabilistic boosting-tree: Learning discriminative models for classification, recognition, and clustering. In *ICCV*, pages 1589–1596, 2005. 1, 2

[14] P. Viola and M. J. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, pages 511–518, 2005. 1

[15] C. Wojek, G. Dorkó, A. Schulz, and B. Schiele. Sliding-windows for rapid object class localization: A parallel technique. In *DAGM sym. on Patt. Rec.*, pages 71–81, 2008. 1

[16] Y. Zheng, A. Barbu, B. Georgescu, M. Scheuering, and D. Comaniciu. Four-chamber heart modeling and automatic segmentation for 3-D cardiac CT volumes using marginal space learning and steerable features. *IEEE T. Med. Imaging*, 27(11):1668–1681, Nov. 2008. 1, 2, 4