Implementing Decision Trees and Forests on a GPU

Toby Sharp

Microsoft Research, Cambridge, UK
toby.sharp@microsoft.com

Abstract. We describe a method for implementing the evaluation and training of decision trees and forests entirely on a GPU, and show how this method can be used in the context of object recognition.

Our strategy for evaluation involves mapping the data structure describing a decision forest to a 2D texture array. We navigate through the forest for each point of the input data in parallel using an efficient, non-branching pixel shader. For training, we compute the responses of the training data to a set of candidate features, and scatter the responses into a suitable histogram using a vertex shader. The histograms thus computed can be used in conjunction with a broad range of tree learning algorithms.

We demonstrate results for object recognition which are identical to those obtained on a CPU, obtained in about 1% of the time.

To our knowledge, this is the first time a method has been proposed which is capable of evaluating or training decision trees on a GPU. Our method leverages the full parallelism of the GPU.

Although we use features common to computer vision to demonstrate object recognition, our framework can accommodate other kinds of features for more general utility within computer science.

1 Introduction

1.1 Previous Work

Since their introduction, randomized decision forests (or random forests) have generated considerable interest in the machine learning community as new tools for efficient discriminative classification [1,2]. Their introduction in the computer vision community was mostly due to the work of Lepetit et al in [3,4]. This gave rise to a number of papers using random forests for: object class recognition and segmentation [5,6], bilayer video segmentation [7], image classification [8] and person identification [9].

Random forests naturally enable a wide variety of visual cues (e.g. colour, texture, shape, depth etc.). They yield a probabilistic output, and can be made computationally efficient. Because of these benefits, random forests are being established as efficient and general-purpose vision tools. Therefore an optimized implementation of both their training and testing algorithms is desirable.

D. Forsyth, P. Torr, and A. Zisserman (Eds.): ECCV 2008, Part IV, LNCS 5305, pp. 595–608, 2008.

This work is complementary to that of Shotton et al [6] in which the authors demonstrate a fast recognition system using forests. Although they demonstrate real-time CPU performance, they evaluate trees sparsely at 1% of pixels and still achieve only 8 frames per second, whereas our evaluations are dense and considerably quicker.

At the time of writing, the premium desktop CPU available is the Intel Core 2 Extreme QX9775 3.2 GHz quad-core. This chip has a theoretical peak performance of 51.2 Gflops using SSE instructions (12.8 Gflops without SSE). With DDR3 SDRAM at 200 MHz, system memory bandwidth peaks at 12.8 GB/s. In contrast, the premium desktop GPU is the nVidia GeForce GTX 280. With its 240 stream processors it has a theoretical peak of 933 Gflops and a memory bandwidth of 141 GB/s.

In [10], the authors demonstrate a simple but effective method for performing plane-sweep stereo on a GPU and achieve real-time performance. In [11], the authors present belief propagation with a chequerboard schedule based on [12]. We follow in similar fashion, presenting no new theory but a method for realizing the GPU's computational power for decision trees and forests.

To our knowledge, this is the first time a method has been proposed which is capable of evaluating or training decision trees on a GPU. In [13], the authors explored the implementation of neural networks for machine learning on a GPU, but did not explore decision trees.

1.2 Outline

Algorithm 1 describes how a binary decision tree is conceptually evaluated on input data. In computer vision techniques, the input data typically correspond to feature values at pixel locations. Each parent node in the tree stores a binary function. For each data point, the binary function at the root node is evaluated on the data. The function value determines which child node is visited next. This continues until reaching a leaf node, which determines the output of the procedure. A forest is a collection of trees that are evaluated independently.

In §2 we describe the features we use in our application which are useful for object class recognition. In §3, we show how to map the evaluation of a decision

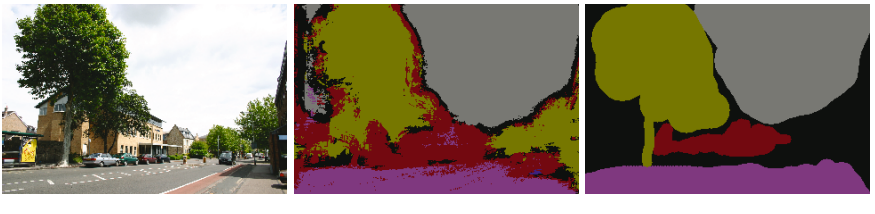


Fig. 1. *Left:* A 320 × 213 image from the Microsoft Research recognition database [14] which consists of 23 labeled object classes. *Centre:* The mode of the pixelwise distribution given by a forest of 8 trees, each with 256 leaf nodes, trained on a subset of the database. This corresponds to the `ArgMax` output option (§3.3). This result was generated in 7 ms. *Right:* The ground truth labelling for the same image.

Algorithm 1. Evaluate the binary decision tree with root node N on input x

```
1. while N has valid children do
2.   if TestFeature(N, x) = true then
3.     N ← N.RightChild
4.   else
5.     N ← N.LeftChild
6.   end if
7. end while
8. return data associated with N
```

forest to a GPU. The decision forest data structure is mapped to a *forest texture* which can be stored in graphics memory. GPUs are highly data parallel machines and their performance is sensitive to flow control operations. We show how to evaluate trees with a *non-branching* pixel shader. Finally, the training of decision trees involves the construction of histograms – a *scatter* operation that is not possible in a pixel shader. In §4, we show how new GPU hardware features allow these histograms to be computed with a combination of pixel shaders and vertex shaders. In §5 we show results with speed gains of 100 times over a CPU implementation.

Our framework allows clients to use any features which can be computed in a pixel shader on multi-channel input. Our method is therefore applicable to more general classification tasks within computer science, such as multi-dimensional approximate nearest neighbour classification. We present no new theory but concentrate on the highly parallel implementation of decision forests. Our method yields very significant performance increases over a standard CPU version, which we present in §5.

We have chosen Microsoft's Direct3D SDK and High Level Shader Language (HLSL) to code our system, compiling for Shader Model 3.

2 Visual Features

2.1 Choice of Features

To demonstrate our method, we have adopted visual features that generalize those used by many previous works for detection and recognition, including [15,16,3,17,7]. Given a single-channel input image I and a rectangle R, let σ represent the sum $\sigma(I, R) = \sum_{\mathbf{x} \in R} I(\mathbf{x})$.

The features we use are differences of two such sums over rectangles R_0, R_1 in channels c_0, c_1 of the input data. The response of a multi-channel image I to a feature $F = \{R_0, c_0, R_1, c_1\}$ is then $\rho(I, F) = \sigma(I[c_0], R_0) - \sigma(I[c_1], R_1)$. The Boolean test at a tree node is given by the threshold function $\theta_0 \leq \rho(I, F) < \theta_1$. This formulation generalizes the Haar-like features of [15], the summed rectangular features of [16] and the pixel difference features of [3]. The generalization of features is important because it allows us to execute the same code for all the nodes in a decision tree, varying only the values of the parameters. This will enable us to write a non-branching decision evaluation loop.

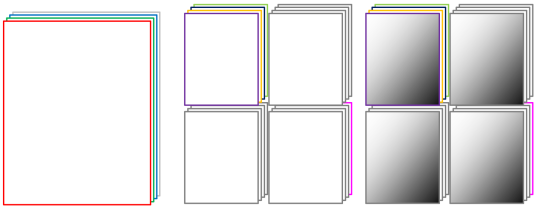


Fig. 2. Processing images for feature computation. Each group of four rectangles represents a four-component (`ARGB`) texture, and each outline in the group represents a single component (channel) of the texture. *(a)* An original sRGB image. *(b)* The image is convolved with 16 filters to produce 16 data channels in 4 four-component textures (§2.3). *(c)* The filtered textures are then integrated (§2.4).

The rectangular sums are computed by appropriately sampling an integral image [15]. Thus the input data consists of multiple channels of integral images. The integration is also performed on the GPU (§2.4).

2.2 Input Data Channels

Prior to integration, we pre-filter sRGB images by applying the bank of separable 2D convolution filters introduced in [14] to produce a 16-channel result. This over-complete representation incorporates local texture information at each pixel. The convolution is also performed on the GPU (§2.3). The pipeline for preparing input textures is shown in Figure 2.

2.3 Convolution with the Filter Bank

For object class recognition, we pre-filter images by convolving them with the 17-filter bank introduced in [14] to model local texture information. Whereas the authors of that work apply their filters in the CIE Lab colour space, we have found it sufficient to apply ours only to the non-linear R, G, B and Y channels. The Gaussians are applied to the RGB channels, and the derivative and Laplacian filters to the luma.

To perform the separable convolution on the GPU, we use the two-pass technique of [18].

Since the pixel shader operates on the four texture components in parallel, up to four filters can be applied in one convolution operation. All 17 filters can therefore be applied in 5 convolutions. In practice we prefer to omit the largest scale Laplacian, applying 16 filters in 4 convolutions.

2.4 Image Integration

The sums over rectangular regions are computed using integral images [15]. Integral images are usually computed on the CPU using an intrinsically serial method, but they can be computed on the GPU using prefix sums [19]. This algorithm is also known as parallel scan or recursive doubling. For details on how this can be implemented on the GPU, see [20].

```
bool TestFeature(sampler2D Input, float2 TexCoord, Parameters Params)
{   // Evaluate the given Boolean feature test for the current input pixel
    float4 Sum1 = AreaSum(Input, TexCoord, Params.Rect1);
    float4 Sum2 = AreaSum(Input, TexCoord, Params.Rect2);
    float Response = dot(Sum1, Params.Channel1) - dot(Sum2, Params.Channel2);
    return Params.Thresholds.x <= Response && Response < Params.Thresholds.y;
}
```

Fig. 3. HLSL code which represents the features used to demonstrate our system. These features are suitable for a wide range of detection and recognition tasks.

2.5 Computation of Features

Figure 3 shows the HLSL code which is used to specify our choice of features (§2.1). The variables for the feature are encoded in the `Parameters` structure. The Boolean test for a given node and pixel is defined by the `TestFeature` method, which will be called by the evaluation and training procedures as necessary.

We would like to stress that, although we have adopted these features to demonstrate our implementation and show results, there is nothing in our framework which requires us to use a particular feature set. We could in practice use any features that can be computed in a pixel shader independently at each input data point, e.g. pixel differences, dot products for BSP trees or multi-level forests as in [6].

3 Evaluation

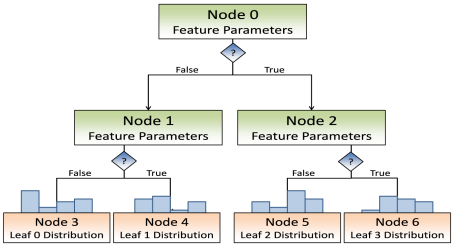
Once the input data textures have been prepared, they can be supplied to a pixel shader which performs the evaluation of the decision forest at each pixel in parallel.

3.1 Forest Textures

Our strategy for the evaluation of a decision forest on the GPU is to transform the forest's data structure from a list of binary trees to a 2D texture (Figure 4). We lay out the data associated with a tree in a four-component `float` texture, with each node's data on a separate row in breadth-first order.

In the first horizontal position of each row we store the texture coordinate of the corresponding node's left child. Note that we do not need to store the right child's position as it always occupies the row after the left child. We also store all the feature parameters necessary to evaluate the Boolean test for the node. For each leaf node, we store a unique index for the leaf and the required output – a distribution over class labels learned during training.

To navigate through the tree during evaluation, we write a pixel shader that uses a local 2D *node coordinate* variable in place of a pointer to the current node (Figure 5). Starting with the first row (root node) we read the feature parameters



Tree 0	LeftChild / LeafNode	Feature Parameters: Rect1	Rect2	Channel1	Channel2	Thresholds	Distribution
Node 0	1 -1	X X X X	X X X X	X X X X	X X X X	X X	
Node 1	3 -1	X X X X	X X X X	X X X X	X X X X	X X	
Node 2	5 -1	X X X X	X X X X	X X X X	X X X X	X X	
Node 3	-1 0						X X X X
Node 4	-1 1						X X X X
Node 5	-1 2						X X X X
Node 6	-1 3						X X X X

Fig. 4. *Left*: A decision tree structure containing parameters used in a Boolean test at each parent node, and output data at each leaf node. *Right*: A 7×5 *forest texture* built from the tree. Empty spaces denote unused values.

```
float4 Evaluate(uniform sampler2D Forest, uniform sampler2D Input,
         uniform float2 PixSize, in float2 TexCoord : TEXCOORD0) : COLOR0
{
    float2 NodeCoord = PixSize * 0.5f;
    // Iterate over the levels of the tree, from the root down...
    [unroll] for (int nLevel = 1; nLevel < MAX_DEPTH; nLevel++)
    {
        float LeftChild = tex2D(Forest, NodeCoord).x;
        // Read the feature parameters for this node...
        Parameters Params = ReadParams(Forest, NodeCoord, PixSize);
        // Perform the user-supplied Boolean test for this node...
        bool TestResult = TestFeature(Input, TexCoord, Params);
        // Move the node coordinate according to the result of the test...
        NodeCoord.y = LeftChild + TestResult * PixSize.y;
    }
    // Read the output distribution associated with this leaf node...
    return Distribution(Forest, NodeCoord);
}
```

Fig. 5. An HLSL pixel shader which evaluates a decision tree on each input point in parallel without branching. Here we have omitted evaluation on multiple and unbalanced trees for clarity.

and evaluate the Boolean test on the input data using texture-dependent reads. We then update the vertical component of our node coordinate based on the result of the test and the value stored in the child position field. This has the effect of walking down the tree according to the computed features. We continue this procedure until we reach a row that represents a leaf node in the tree, where we return the output data associated with the leaf.

For a forest consisting of multiple trees, we tile the tree textures horizontally. An outer loop then iterates over the trees in the forest; we use the horizontal component of the node coordinate to address the correct tree, and the vertical component to address the correct node within the tree. The output distribution for the forest is the mean of the distributions for each tree.

This method allows our pixel shader to be *non-branching* (i.e. it does not contain flow control statements) which is crucial for optimal execution performance.

3.2 Geometry and Texture Set-Up

In order to evaluate the forest at every data point, we render a rectangle that covers the size of the input data. The first texture stage is assigned the forest texture described above.

There is a slight difficulty with presenting the input data to the shader. According to our discussion of features (§2) we have 16-17 input channels and for each tree node we wish to choose the correct channel on which to evaluate the feature response. Unfortunately there is no legal way within the pixel shader to index an array of textures based on a read operation. Neither is it possible to create a 3D texture from the input channels and use the required channel to address the third dimension. To overcome this difficulty, we tile each group of four input channels into one large four-component texture, and bind the latter to the second texture stage.

When we create the forest texture, we must modify appropriately the coordinates of the feature rectangles so that they address the correct channel group within the tiled texture as well as the correct spatial location. We also set unit vectors (`Channel0, Channel1` in Figure 3, 4) which are used in dot products to select the correct scalar output.

3.3 Output Modes

To give clients sufficient control, we have implemented several different output modes for forest evaluation:

- `Distribution`: Outputs the evaluated distribution over L labels into L output channels
- `ArgMax`: Outputs the index of the label with the greatest probability in the evaluated distribution
- `ForestLeaves`: Outputs the index of the leaf node reached in each of T trees into T output channels
- `TreeLeaf`: Outputs the index of the leaf node reached in the first tree

Due to limitations of Direct3D 9, a maximum of 16 output channels can be used for the `Distribution` and `ForestLeaves` modes. Where more than 16 label posteriors are required, they can be computed efficiently in multiple passes from one or more *leaf images* generated by the `ForestLeaves` option. If additionally the number of trees exceeds 16, the forest can be split into groups of trees which are evaluated in succession. Thus any number of trees and class labels can be handled by our system.

4 Training

Training of randomized trees is achieved iteratively, growing a tree by one level each *training round*. For each training round, a pool of candidate features is

sampled, and these are then evaluated on all the training data to assess their discriminative ability. Joint histograms over ground truth labels and feature responses are created, and these histograms may be used in conjunction with various learning algorithms, e.g. ID3 [21] or C4.5 [22], to choose features for new tree nodes. Thus learning trees can be a highly compute-intensive task. We adopt a general approach for efficient training on the GPU, suitable for any learning algorithm.

A training database consists of training examples together with ground truth class labels. Given a training database, a pool of candidate features and a decision tree, we compute and return to our client a histogram that can be used to extend the tree in accordance with a learning algorithm. For generality, our histogram is 4D and its four axes are: the leaf node index, ground truth class label, candidate feature index and quantized feature response. Armed with this histogram, clients can add two new children to each leaf of the current tree, selecting the most discriminative candidate feature as the new test.

In one sweep over the training database we visit each labeled data point and evaluate its response to each candidate feature. We also determine the active leaf node in the tree and increment the appropriate histogram bin. Thus for each *training round* we evaluate the discriminative ability of all candidate features at all leaf nodes of the current tree.

4.1 Histogram Initialization

We store the 4D histogram in a single 2D texture by tiling 2D slices through the histogram. The tiling strategy is chosen dynamically at runtime to avoid exceeding the maximum texture width and height of 4096. Note that the size of the histogram is independent of the size of the training database. Although we have not found it necessary in practice, it would be possible to split very large histograms across multiple textures. We use a 32-bit `float` texture for convenience and to avoid overflow.

4.2 Training Data

We now iterate through the training database, evaluating all the feature responses and accumulating them into the histogram. To access the training data as required we use a callback interface supplied by the client.

In order to evaluate the feature discrimination for every leaf node of the tree simultaneously, we need to determine which leaf of the current tree would be activated for each training data point. We achieve this by the method of §3 using the `TreeLeaf` output option to generate a *leaf image* that represents the active leaf nodes of the current tree for the current training image.

4.3 Feature Evaluation

We request four training examples at a time from the database because this will allow us to take full advantage of the GPU's SIMD capabilities by operating on four texture components in parallel (Figure 6a).

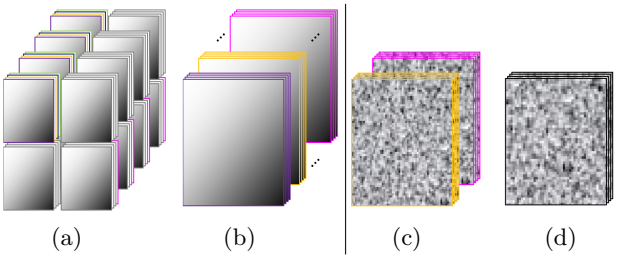


Fig. 6. Processing training data. *(a)* A set of four training examples, already pre-processed for feature computation. *(b)* The same data, rearranged so that each texture contains corresponding channels from all four training examples. *(c)* The appropriate textures are selected for a given feature and the box filters applied. *(d)* The final feature response is the difference of the two box filtered textures.

Since we are pre-filtering our sRGB image data (§2), we can either perform all the pre-processing to the training database in advance, or we can apply the pre-processing as each image is fetched from the database. After the pre-filtering (Figure 6a) we re-arrange the texture channels so that each texture contains one filtered component from each of the four training examples (Figure 6b). The input textures are thus prepared for evaluating our features efficiently.

We then iterate through the supplied set of candidate features, computing the response of the current training examples to each feature. For each feature we select two input textures according to the channels specified in the feature (Figure 6c). We compute each box filter convolution on four training images in parallel by passing the input texture to a pixel shader that performs the necessary look-ups on the integral image. In a third pass, we subtract the two box filter responses to recover the feature response (Figure 6d).

We ensure that our leaf image (§4.2) also comprises four components that correspond to the four current training examples.

4.4 Histogram Accumulation

The computed feature responses are then accumulated into the 4D histogram, using also the values of the leaf image and ground truth label at each pixel. Histogram accumulation is a *scatter* operation rather than a *gather* operation, so it cannot be implemented in a pixel shader. Instead we use a vertex shader, inspired by [23], to perform the scattering.

As our vertex buffer input we supply a list of all the 2D texture coordinates which cover the area of the feature response image. Our vertex shader (Figure 7) uses this input coordinate to read the feature response value, ground truth label and leaf index at one position from three supplied input textures. The other value required for the histogram operation is the feature index which is passed in as a constant. The shader then computes the 4D-to-2D mapping according to the active tiling scheme (§4.1). Finally the 2D histogram coordinate is output by the vertex shader.

```
float4 Scatter(uniform sampler2D Textures[3], uniform float4 select,
               in out float2 coord : TEXCOORD0, uniform float feature)
{
    float4 address = {coord.x, coord.y, 0.0f, 0.0f};
    float leaf = dot(tex2Dlod(Textures[0], address), select);
    float label = dot(tex2Dlod(Textures[1], address), select);
    float response = dot(tex2Dlod(Textures[2], address), select);
    float2 pos = Map4DTo2D(leaf, label, response, feature);
    return float4(pos.x, pos.y, 0.0f, 1.0f);
}
```

Fig. 7. An HLSL vertex shader that scatters feature response values to the appropriate position within a histogram

A simple pixel shader emits a constant value of 1 and, with additive blending enabled, the histogram values are incremented as desired.

We execute this pipeline four times for the four channels of data to be aggregated into the histogram. A shader constant allows us to select the required channel.

4.5 Quantization and Thresholds

In order to histogram the real-valued feature responses they must first be quantized. We require that the client provides the number of quantization bins to use for the training round. An interval of interest for response values is also provided for each feature in the set of candidates. In our `Scatter` vertex shader, we then linearly map the response interval to the histogram bins, clamping to end bins. We make the quantization explicit in this way because different learning algorithms may have different criteria for choosing the parameters used for the tree's Boolean test.

One approach would be to use 20-30 quantization levels during a training round and then to analyze the histogram, choosing a threshold value adaptively to optimize the resulting child distributions. For example, the threshold could be chosen to minimize the number of misclassified data points or to maximize the KL-divergence. Although this method reduces training error, it may lead to over-fitting. Another approach would be to use a very coarse quantization (only 2 or 3 bins) with randomized response intervals. This method is less prone to over-fitting but may require more training rounds to become sufficiently discriminative.

We have tested both of the above approaches and found them effective. We currently favour the latter approach, which we use with the ID3 algorithm [21] to select for every leaf node the feature with the best information gain. Thus we double the number of leaf nodes in a tree after each training round.

We create flexibility by not requiring any particular learning algorithm. Instead, by focusing on computation of the histogram, we enable clients to adopt their preferred learning algorithm efficiently.

5 Results

Our test system consists of a dual-core Intel Core 2 Duo 2.66 GHz and an nVidia GeForce GTX 280. (Timings on a GeForce 8800 Ultra were similar.) We have coded our GPU version using Direct3D 9 with HLSL shaders, and a CPU version using C++ for comparison only. We have not developed an SSE version of the CPU implementation which we believe may improve the CPU results somewhat (except when performance is limited by memory bandwidth). Part of the appeal of the GPU implementation is the ability to write shaders using HLSL which greatly simplifies the adoption of vector instructions.

In all cases identical output is attained using both CPU and GPU versions. Our contribution is a method of GPU implementation that yields a considerable speed improvement, thereby enabling new real-time recognition applications.

We separate our results into timings for pre-processing, evaluation and training. All of the timings depends on the choice of features; we show timings for our generalized recognition features. For reference, we give timings for our feature pre-processing in Figure 8.

5.1 Tree Training

Training time can be prohibitively long for randomized trees, particularly with large databases. This leads to pragmatic short-cuts such as sub-sampling training data, which in turn has an impact on the discrimination performance of learned trees.

Our training procedure requires time linear in the number of training examples, the number of trees, the depth of the trees and the number of candidate features evaluated.

Description	Resolution (pixels)	CPU (ms)	GPU (ms)	Speed-up (×)
Convolution with 16 separable filters (§2.3)	320 × 240	94	8.5	11.1
	640 × 480	381	15.5	24.6
Integration of 16 data channels (§2.4)	320 × 240	9.2	7.0	1.31
	640 × 480	31	16.6	1.87

Fig. 8. Timings for feature pre-processing

Operation	CPU		GPU		Speed-up
	(s)	(%)	(s)	(%)	(×)
100× Leaf image computation (§4.2)	6.0	6	0.2	2	30
10^4× Feature responses (§4.3)	39.5	41	0.2	2	198
10^4× Histogram accumulations (§4.4)	52.1	53	11.8	96	4.4
Total	97.6	100	12.2	100	8.0

Fig. 9. Breakdown of time spent during one training round with 100 training examples and a pool of 100 candidate features. Note the high proportion of time spent updating the histogram.

To measure training time, we took 100 images from the labeled object recognition database of [14] with a resolution of 320×213. This data set has 23 labeled classes. We used a pool of 100 candidate features for each training round. The time taken for each training round was 12.3 seconds. With these parameters, a balanced tree containing 256 leaf nodes takes 98 seconds to train. Here we have used every pixel of every training image.

Training time is dominated by the cost of evaluating a set of candidate features on a training image and aggregating the feature responses into a histogram. Figure 9 shows a breakdown of these costs. These figures are interesting as they reveal two important insights:

First, the aggregation of the histograms on the GPU is comparatively slow, dominating the training time significantly. We experimented with various different method for accumulating the histograms, maintaining the histogram in system memory and performing the incrementation on the CPU. Unfortunately, this did not substantially reduce the time required for a training round. Most recently, we have begun to experiment with using CUDA [24] for this task and we anticipate a significant benefit over using Direct3D.

Second, the computation of the rectangular sum feature responses is extremely fast. We timed this operation as able to compute over 10 million rectangular sums per ms on the GPU. This computation time is insignificant next to the other timings, and this leads us to suggest that we could afford to experiment with more arithmetically complex features without harming training time.

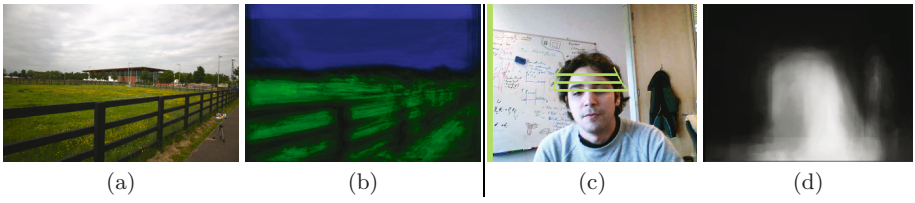
5.2 Forest Evaluation

Our main contribution is the fast and parallel evaluation of decision forests on the GPU. Figure 10 show timings for the dense evaluation of a decision forest, with various different parameters.

Our method maps naturally to the GPU, exploiting its parallelism and cache, and this is reflected in the considerable speed increase over a CPU version by around two orders of magnitude.

Resolution (pixels)	Output Mode	Trees	Classes	CPU (ms)	GPU (ms)	Speed-up (×)
320×240	`TreeLeaf`	1	N/A	70.5	0.75	94
320×240	`ForestLeaves`	4	N/A	288	3.0	96.1
320×240	`Distribution`	8	4	619	5.69	109
320×240	`ArgMax`	8	23	828	6.85	121
640×480	`TreeLeaf`	1	N/A	288	2.94	97.8
640×480	`ForestLeaves`	4	N/A	1145	12.1	95.0
640×480	`Distribution`	8	4	2495	23.1	108
640×480	`ArgMax`	8	23	3331	25.9	129

Fig. 10. Timings for evaluating a forest of decision trees. Our GPU implementation evaluates the forest in about 1% of the time required by the CPU implementation.



(a) (b) (c) (d)

Fig. 11. (a)-(b) Object class recognition. A forest of 8 trees was trained on labelled data for grass, sky and background labels. *(a)* An outdoor image which is not part of the training set for this example. *(b)* Using the `Distribution` output option, the blue channel represents the probability of sky and the green channel the probability of grass at each pixel (5 ms). **(c)-(d) Head tracking in video.** A random forest was trained using spatial and temporal derivative features instead of the texton filter bank. *(c)* A typical webcam video frame with an overlay showing the detected head position. This frame was not part of the training set for this example. *(d)* The probability that each pixel is in the foreground (5 ms).

5.3 Conclusion

We have shown how it is possible to use GPUs for the training and evaluation of general purpose decision trees and forests, yielding speed gains of around 100 times.

References

1. Amit, Y., Geman, D.: Shape quantization and recognition with randomized trees. Neural Computation 9(7), 1545–1588 (1997)
2. Breiman, L.: Random forests. ML Journal 45(1), 5–32 (2001)
3. Lepetit, V., Fua, P.: Keypoint recognition using randomized trees. IEEE Trans. Pattern Anal. Mach. Intell. 28(9), 1465–1479 (2006)
4. Ozuysal, M., Fua, P., Lepetit, V.: Fast keypoint recognition in ten lines of code. In: IEEE CVPR (2007)
5. Winn, J., Criminisi, A.: Object class recognition at a glance. In: IEEE CVPR, video track (2006)
6. Shotton, J., Johnson, M., Cipolla, R.: Semantic texton forests for image categorization and segmentation. In: IEEE CVPR, Anchorage (2008)
7. Yin, P., Criminisi, A., Winn, J.M., Essa, I.A.: Tree-based classifiers for bilayer video segmentation. In: CVPR (2007)
8. Bosh, A., Zisserman, A., Munoz, X.: Image classification using random forests and ferns. In: IEEE ICCV (2007)
9. Apostolof, N., Zisserman, A.: Who are you? - real-time person identification. In: BMVC (2007)
10. Yang, R., Pollefeys, M.: Multi-resolution real-time stereo on commodity graphics hardware. In: CVPR, vol. (1), pp. 211–220 (2003)
11. Brunton, A., Shu, C., Roth, G.: Belief propagation on the gpu for stereo vision. In: CRV, p. 76 (2006)
12. Felzenszwalb, P.F., Huttenlocher, D.P.: Efficient belief propagation for early vision. International Journal of Computer Vision 70(1), 41–54 (2006)

13. Steinkraus, D., Buck, I., Simard, P.: Using gpus for machine learning algorithms. In: Proceedings of Eighth International Conference on Document Analysis and Recognition, 2005, 29 August-1 September 2005, vol. 2, pp. 1115–1120 (2005)
14. Winn, J.M., Criminisi, A., Minka, T.P.: Object categorization by learned universal visual dictionary. In: ICCV, pp. 1800–1807 (2005)
15. Viola, P.A., Jones, M.J.: Robust real-time face detection. International Journal of Computer Vision 57(2), 137–154 (2004)
16. Shotton, J., Winn, J.M., Rother, C., Criminisi, A.: TextonBoost: Joint appearance, shape and context modeling for multi-class object recognition and segmentation. In: Leonardis, A., Bischof, H., Pinz, A. (eds.) ECCV 2006. LNCS, vol. 3951, pp. 1–15. Springer, Heidelberg (2006)
17. Deselaers, T., Criminisi, A., Winn, J.M., Agarwal, A.: Incorporating on-demand stereo for real time recognition. In: CVPR (2007)
18. James, G., O'Rorke, J.: Real-time glow. In: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, pp. 343–362. Addison-Wesley, Reading (2004)
19. Blelloch, G.E.: Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (November 1990)
20. Hensley, J., Scheuermann, T., Coombe, G., Singh, M., Lastra, A.: Fast summed-area table generation and its applications. Comput. Graph. Forum 24(3), 547–555 (2005)
21. Quinlan, J.R.: Induction of decision trees. Machine Learning 1(1), 81–106 (1986)
22. Quinlan, J.: C4.5: Programs for Machine Learning. Morgan Kaufmann, California (1992)
23. Scheuermann, T., Hensley, J.: Efficient histogram generation using scattering on gpus. In: SI3D, pp. 33–37 (2007)
24. `http://www.nvidia.com/cuda`