Improving FPGA Accelerated Tracking with Multiple Online Trained Classifiers

Matthew Jacobsen, Siddarth Sampangi, Yoav Freund, Ryan Kastner
Computer Science & Engineering
University of California, San Diego
{mdjacobs, ssampang, yfreund, kastner}@cs.ucsd.edu

***Abstract*—Robust real time tracking is a requirement for many emerging applications. Many of these applications must track objects even as their appearance changes. Training classifiers online has become an effective approach for dealing with variability in object appearance. Classifiers can learn and adapt to changes online at the cost of additional runtime computation. In this paper, we propose a FPGA accelerated design of an online boosting algorithm that uses multiple classifiers to track and recover objects in real time. Our algorithm uses a novel method for training and comparing pose-specific classifiers along with adaptive tracking classifiers. Our FPGA accelerated design is able to track at 60 frames per second while concurrently evaluating 11 classifiers. This represents a 30$\times$ speed up over a CPU based software implementation. It also demonstrates tracking accuracy at state of the art levels on a standard set of videos.**

I. Introduction

Robust visual object tracking has improved considerably in recent years and is becoming a widely used enabling technology. It empowers a host of applications in fields such as: human computer interaction [1], autonomous vehicles [2], and video surveillance [3]. Yet it is still an especially difficult task for computers as objects change in appearance over time. Rotation, scaling, and using lighting insensitive color space transformations can compensate for some changes in appearance. But rotations out of plane, occlusions, and object deformation still present a problem for most algorithms.

Algorithms that learn an appearance model online have proven effective for robust tracking [4], [5]. Instead of training offline with large volumes of examples, the approach is to train online using a modest number of examples gathered at runtime. This approach has the benefit of training with examples that match the current appearance. But it requires additional online computation which can affect runtime performance.

Boosting for feature selection has been used with great success for detection [6] and tracking [4]. Boosting is the process of combining many weak classifiers into a single strong classifier that performs better in aggregate. Training examples are used in an iterative process to identify the most discriminative weak classifiers. In the tracking context, weak classifiers are image features. Online boosting for tracking is an adaptation of boosting that fits the online training approach. Appearance model features are boosted to generate a classifier that can detect an object as it currently appears. As the object's appearance changes, new examples are gathered, and the classifier is updated accordingly.

Our contributions in this paper deal with online boosting for tracking. Tracking has three main components: an appearance model which identifies the object, a dynamics model which governs how the object moves, and a search strategy which defines where to look for the object. We address the challenge of tracking visibly changing objects through improvements in the appearance model. This work is based on our previous work in FPGA accelerated online boosting [7]. We improve upon the single online boosted tracker approach by using multiple classifiers learned at runtime. These classifiers are trained to recognize specific poses of a target which helps maintain location accuracy. To use multiple trackers effectively, we present a novel method for comparing classifier scores. Lastly, we accelerate our algorithm using a FPGA. The additional computation power from the FPGA allows the algorithm to evaluate and train up to 11 different classifiers each frame at 60 frames per second (FPS). Compared to the original algorithm, our work shows not only improvements in runtime performance, but in tracking accuracy as well. The contributions of this paper are:

- An algorithm for learning a pool of pose-specific classifiers at runtime.
- A method for comparing multiple classifier scores.
- A FPGA-CPU design and implementation of our algorithm for robust tracking.

The rest of the paper is organized as follows. We discuss related work next. The algorithm is described in Section III. The FPGA design is presented in Section IV. This is followed by experimental results in Section V and conclusions.

II. Related work

Research in object tracking frequently takes one of two directions. Research in tracking algorithms and features often leads to improvements in accuracy. While research in hardware acceleration often leads to improvements in runtime performance. In our work, we provide an algorithm and FPGA accelerated design that leads to improvements in both.

From an algorithmic perspective, our design uses an online boosting approach for the appearance model similar to that employed by Viola [8]. It builds on the work of Babenko [5] by using classifiers trained at runtime. Several other algorithms use this adaptive approach [4], [9], [10] including the well known Predator algorithm by Kalal [11]. These algorithms attempt to learn a representation of the target object using a single classifier over time. This can be a successful approach for certain classes of objects that have a limited number of visual representations. For deformable objects or objects that can vary substantially in appearance, many classifiers

Algorithm 1 Classifier Tracking Algorithm

```
Input: New image frame at time $t$
1: Select a set of samples, cropped from frame,
       $X^s = \{x | s > \|l(x) - l^*_{t-1}\|\}$
2: Calculate Haar feature values, $f(x)$, for $x \in X^s$
3: Use classifier, $\boldsymbol{H}(x) = \sum_k h_k(x)$, to classify samples $X^s$
4: Set new location $l^*_t = l(\mathrm{argmax}_{x \in X^s} \boldsymbol{H}(x))$
5: Select positive and negative samples sets from frame,
       $X_1 = \{x | r > \|l(x) - l^*_{t-1}\|\}$
       $X_0 = \{x | q \leq \|l(x) - l^*_{t-1}\| < q\prime\}$
6: Train classifier on positive and negative sets,
       $\boldsymbol{H} = Train(X_1, X_0)$
```

Algorithm 2 Main Tracking Algorithm

```
Input: New image frame at time $t$
1: Evaluate all classifiers at location
       $\{X^1, X^2, X^3, X^{Pool}\} = Evaluate(T^1, T^2, T^3, Pool)$
2: Calculate majority, $x^*_T = Majority(X^1, X^2, X^3)$
3: Calculate pose-specific classifier pool maximum,
       $x^*_{Pool} = \mathrm{argmax}_{x \in X^{Pool}}(Kurtosis(x) \times Score(x))$
4: Use maximum as location, $l^* = l(\mathrm{argmax}_{x^*_T, x^*_{Pool}} Score(x))$
5: Identify pool classifiers with detections,
       $D = \{p \in Pool | ValidDetection(p)\}$
6: Update least used priority queue, $Q = UpdateQueue(Q, D)$
7: Start training a new pose-specific classifier if possible,
8: if $Stable(X^1, X^2, X^3)$ and $P_{new} == none$ then
9:     $P_{new} = \min(Q)$
10:    $Pool = Pool \cup P_{new}$
11: else if not $DetectedEnough(P_{new})$ then
12:    $Pool = Pool \setminus P_{new}$
13:    $P_{new} = none$
14: end if
15: Train classifiers, $Train(T^1, T^2, T^3, D)$
```

do not have the expressiveness to learn all representations. Our approach is unlike other adaptive appearance algorithms because we employ multiple classifiers to handle changes in appearance. Our tracking classifiers adapt to a short term appearance history. Our pose-specific classifiers detect previous representations with high discriminative capability. Both classifiers are used to track objects as they move and change in appearance. This approach requires incorporating multiple classifier predictions into a single algorithmic prediction.

From a hardware perspective, our work with online boosting is similar to that contributed by Lo [12]. Lo proposes a method for accelerating boosted training for Viola-Jones style detectors. They achieve a 14× speed up over a CPU. But their work is aimed at accelerating traditional offline boosting. Other FPGA tracking accelerated work in the field focuses mostly on accelerating classifier evaluation [13] or search methods [14]. Most are standalone designs that do not improve upon the accuracy of the algorithms they accelerate. Moreover, to our knowledge, no other work accelerates online training using dedicated hardware.

III. ALGORITHM

Our algorithm uses multiple concurrent classifiers each frame and merges their outputs to select a new location.These classifiers follow an online boosting algorithm proposed by Babenko [5].

A. *Classifier Algorithm*

Babenko's algorithm is an online boosting algorithm for tracking which uses Multiple Instance Learning [15]. It was selected because of its adaptive appearance model. It uses a first order dynamics model for motion and uniform search strategy. The algorithm consists of two steps: find the new target location, then update the trained classifier. For each frame, the first step evaluates windows in a region surrounding the last known target location (radius s). Evaluation yields a new location. Then the region surrounding the new location is used to to select positive and negative training examples (radii q and $q\prime$). These examples are used to incrementally train the classifier for the next frame. This tracking flow is illustrated in Algorithm 1.

The algorithm uses a boosted collection of Haar-like features for the appearance model. A Haar-like feature is a collection of two to six weighted rectangular regions defined within a window. The rectangles define regions of the window over which to sum pixel values. The rectangles need not be adjacent. We refer to these features simply as Haar features in the rest of this paper.

Boosting is an approach which combines several weak classifiers, h, into a strong classifier, $\boldsymbol{H}$ by iteratively maximizing the log likelihood of the strong classifier log $\mathcal{L}(\boldsymbol{H}_{k-1} + h)$. At each iteration, k, the existing strong classifier is combined with the next weak classifier, h, that maximizes this quantity over the training data.

At initialization, a pool of M Haar features are generated with random rectangle coordinates and weights. During training, K of these features are selected and are used as weak classifiers, $h(x)$, to form the strong classifier, $\boldsymbol{H}(x) = \sum_k \mathbf{h}_k(x)$. Each weak classifier consists of a Haar feature and four additional parameters: μ_1, σ_1, μ_0, σ_0. These parameters quantify the distribution of Haar scores, $f(x)$, for positive and negative examples respectively. They are updated during training using examples collected from the current frame. At runtime, each weak classifier calculates the score of a sample window, x, using the log odds ratio:

$$\begin{aligned} h(x) = \frac{1}{2\sigma_0}(f(x) - \mu_0)^2 - \log(\frac{1}{\sqrt{\sigma_0}}) \\ -\frac{1}{2\sigma_1}(f(x) - \mu_1)^2 + \log(\frac{1}{\sqrt{\sigma_1}}). \end{aligned} \tag{1}$$

This formula results in higher scores the closer a sample's feature value is to the positive mean and the further it is from the negative mean.

Each classifier employed by our algorithm uses a modified version of this tracking algorithm. To improve tracking efficacy, we change the classifier's scoring function to only use the positive example distribution:

$$h(x) = -\frac{1}{2\sigma_1}(f(x) - \mu_1)^2. \tag{2}$$

Positive training examples gathered across frames will likely have a similar distribution because they are all sampled from the target's location. However, negative examples are drawn from multiple different locations in a frame and across frames. A single normal distribution will model this distribution poorly

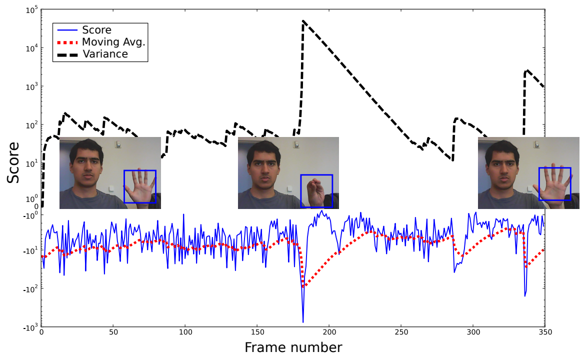

Fig. 1: Classifier scores during target appearance changes. Changes produce sharp drops in score and spikes in variance. At frame 180 the tracked hand changes as the hand closes. After a period of adaptation, tracking becomes stable again until frame 340 when the hand opens back up.

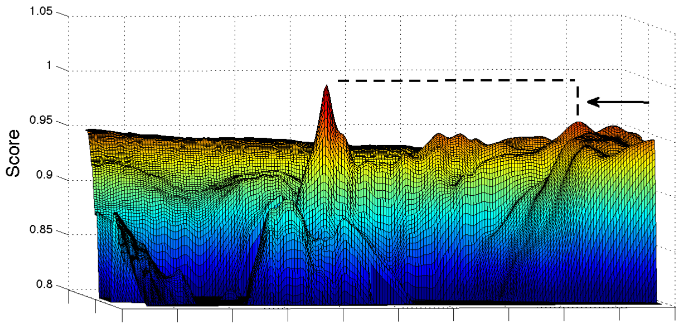

Fig. 2: Plot of a classifier's score over the X and Y dimensions. This example shows a sharp peak and the next highest peak at least d pixels away. The variance normalized difference between these two peaks is the kurtosis. Large differences between peaks indicates high confidence in the global maximum.

and distort scores. In practice, we confirmed that using our scoring function produced more reliable, more stable scores. This does not mean negative examples are ignored. Negative examples are still used during classifier training.

Each classifier is also modified to calculate normalized cross-correlation scores for each window, in addition to Haar feature based scores. The normalized cross-correlation is computed between a pixel template captured at runtime and each frame window in the search region. Both Haar and normalized cross-correlation are used as independent methods of locating the target object within the search region.

Lastly, each classifier calculates the mean and variance of scores across all windows in the search region. It also returns the 128 highest scoring locations detected in each frame, instead of just the maximum location.

B. Main Algorithm

Our algorithm is both motivated and enabled by FPGA acceleration. FPGA acceleration provides runtime resources beyond a typical CPU based implementation. These additional resources can be used to track multiple independent targets through simple replication. They can also be used to improve tracking a single target. However, combining the results of multiple classifiers is not straightforward. Our algorithm uses multiple classifiers per target in two different ways. It uses three adaptive classifiers for a single target, and trains pose-specific classifiers to be evaluated concurrently each frame. This flow is described in Algorithm 2.

Our algorithm uses three adaptive classifiers with a majority aggregation function. These classifiers are evaluated and trained every frame. The majority location is the new target location. Any adaptive tracking classifier that drifts over a threshold distance away from this location is re-centered on the majority location. Using multiple classifiers improves robustness because each classifier is able to track a different part of the target, at a different scale, with different features. We found using the majority provided better results over using the maximum because spurious, high scoring, incorrect maximal locations from a single classifier do not influence the majority location.

Before using multiple classifiers, we attempted to mitigate target loss by using more features with a single tracker. This approach showed improvement, but did not reduce drift or target loss by any significant amount. The number of features that can fit in a fixed sized sliding window are limited. Adding additional features per classifier has a diminishing margin of return but reduces runtime performance linearly.

Our algorithm also learns pose-specific classifiers at runtime and evaluates them concurrently with the adaptive tracking classifiers. Pose-specific classifiers are trained to detect a specific representation of the target. The intuition is that a single tracking classifier cannot effectively learn every representation of an object (imagine a vehicle from different angles or a human hand performing gestures). Individual classifiers can however learn a specific object pose with high discriminative ability. If such poses are revisited in future frames, trained pose-specific classifiers can be used to recover from tracking drift or even complete target loss.

Unlike the adaptive tracking classifiers, pose-specific classifiers are trained only when the target appears in the same pose. We take an active learning approach to detect these situations. For each frame, if the pose-specific classifier's maximum scored location and a normalized cross-correlation maximum location agree, the detection is valid and the classifier is trained. Otherwise, no training takes place. Valid detections are used as potential new target locations. The normalized cross-correlation is performed using a template acquired on the first frame of training.

Creating a pose-specific classifier risks training on something other than the target. This can happen when the current location has drifted or the target is lost. The algorithm cannot detect drift or loss without supervision or feedback. Instead it waits until the tracking classifiers are temporarily stable. Stable simply means the target is currently being tracked (moving or not) with high confidence. Our algorithm identifies these times using the current score, average score and variance. Times of instability are punctuated by large drops in score, followed by a period of adaptation characterized by high variance (see Figure 1). Constant training makes it impossible to learn a consistent mean, so a decaying average mean is maintained, along with its variance. When the current score is above the mean and the variance has decreased below a threshold, the tracking classifiers are considered temporarily stable.

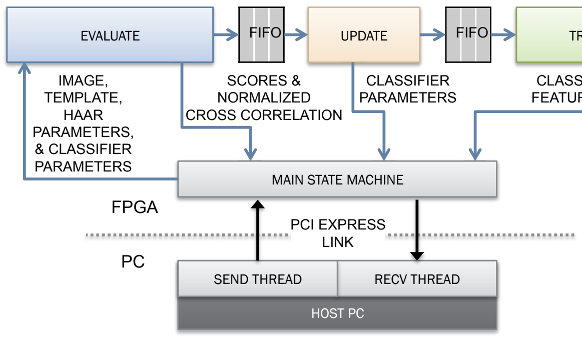

Fig. 3: FPGA-CPU high level architecture. The FPGA design is composed of a three stage pipeline: Evaluate, Update, and Train. The pipeline is controlled by a state machine that also interfaces with software running on the host PC.

Only one pose-specific classifier is trained at a time. It is given a fixed number of frames within which it must be trained a minimum number of times. If the pose does not exist long enough, the classifier will not meet the minimum training threshold and it is discarded. If it does meet the threshold, it is kept and added to a pool of concurrently evaluated pose-specific classifiers. Each frame, all classifiers in the pool are evaluated at the current location.

Each additional pose-specific classifier consumes runtime resources. For a given performance level, only a fixed number can be evaluated at a time. But video sequences may produce an unbounded number of new pose-specific classifiers. We therefore, employ a least used eviction model to limit the number of classifiers evaluated each frame. The most used classifiers are prioritized highest in a queue. The lowest in this queue is replaced when a new classifier starts training. Detections by a classifier increases its queue priority independent of other classifier detections.

Lastly, evaluating multiple pose-specific classifiers requires an aggregation function when multiple valid detections occur. Our algorithm uses the maximum classifier score multiplied by its kurtosis. The kurtosis provides a measure of the score distribution's ”peakedness”. The algorithm calculates kurtosis over the distribution as:

$$\frac{\max\{h(x)\} - \max\{h(x)|d > \|l(x) - l^*\|\}}{\sigma_{h(x)}}. \tag{3}$$

This calculates the difference between the global maximum score and a local maximum score, normalized by the standard deviation of all scores, $\sigma_{h(x)}$. The local maximum score is the largest score at least d pixels away from the location of the global maximum l^*. The d pixel radius is selected to separate independent peaks. A small difference between maximums suggests low confidence in the global maximum location, as the next highest peak is also a good candidate. Figure 2 illustrates this measurement graphically with a high confidence example.

IV. FPGA-CPU DESIGN

The FPGA-CPU design is a partitioned application that accelerates boosted classifier evaluation and classifier training. The tracking application runs in software on the CPU. However, all of the feature extraction, evaluation, and training take place on the FPGA. Mostly image cropping, rendering, and various bookkeeping tasks run on the CPU.

Each video frame is processed by the algorithm in two passes. The first pass performs the target search by evaluating classifiers in a sliding window fashion over a region of the frame. This search limits evaluation to windows within a runtime specified radius of the last known location. The search is a dense evaluation of all possible (overlapping) windows, at a single scale. After the search identifies the new target location, the classifiers are retrained. This takes place during the second pass. Training examples are sampled from the current frame at the newly identified target location. All windows in a small radius at the new location are used as positive examples. Negative examples are sparsely sampled from an annular region surrounding the new location. This process is described in more detail in previous work [7].

The high level architecture of this design is illustrated in Figure 3. The CPU communicates with the FPGA over a PCIe link using the RIFFA framework [16]. CPU software invokes the IP cores on the FPGA with input data and receives the response as output. The FPGA design consists primarily of a three stage pipeline. The *Evaluate* stage evaluates trained classifiers on the video frame. The *Update* stage uses the output of the Evaluate stage to update classifier parameters. These updated parameters provide the basis for a new classifier. The *Train* stage selects which features, and thus which updated parameters, are part of the new classifier.

A. *Evaluate stage*

The Evaluate stage is responsible for extracting Haar features, calculating classifier scores and normalized cross correlation on windows within the frame. The architecture of this stage is illustrated in Figure 4. Input frame pixels, an image template, and other parameters are supplied via the PCIe link. Parameters and template pixels are stored in Block RAM (BRAM). Frame pixels are processed as they are streamed through the data path.

Input frame pixels are first rescaled to fit within a fixed size window in a sliding window pipeline. In our previous work accelerating online boosting for tracking, we used a 20$\times$20 pixel window of registers. This size was selected based on numerous other publications using similar sized sliding window designs. In practice however, we found that many of the targets we tracked required considerable down sampling to fit in these dimensions. This resulted in loss of texture and detail in the image. Because Haar features work best with high frequency texture and sharp edges, this introduced a source of error for our calculations. We expect this is a common source of error for many sliding window designs, yet it is often unaddressed in the literature. In this work we use a 45$\times$45 pixel window, backed by BRAM. Experiments with the two sizes confirmed an improvement in tracking accuracy with the larger window size. Using BRAM instead of registers requires multiple BRAM modules to maintain parallel access to the window data. However an equivalent sized register backed window would exceed the look up table (LUT) capacity of most commercially available FPGAs. This is due to the quadratic growth of register and MUX usage with size.

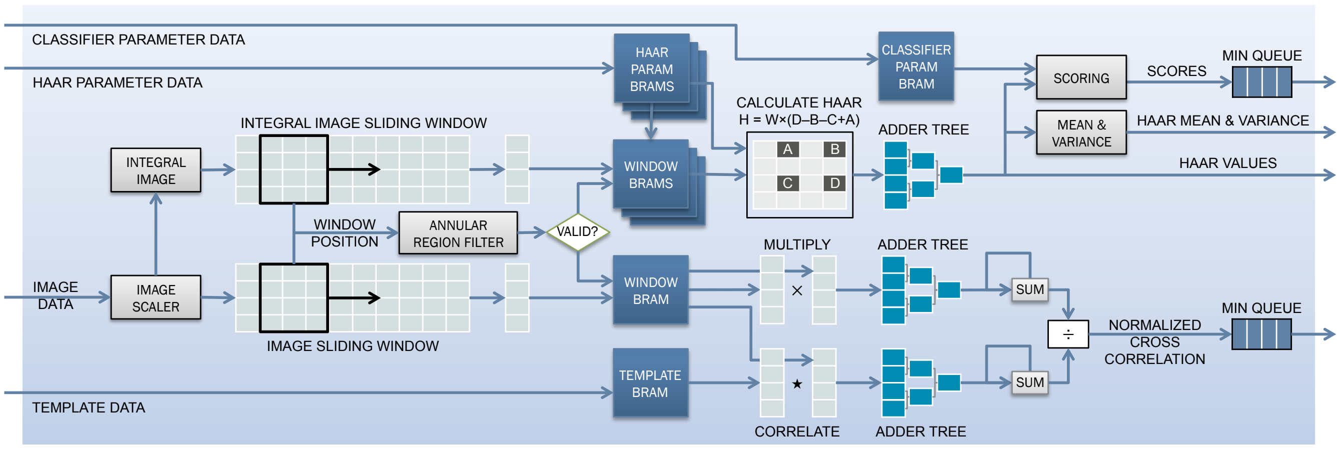

Fig. 4: Evaluate stage architecture. Input pixels are scaled and then converted into integral image format. Two parallel sliding window pipelines calculate Haar and NCC values. Haar values are scored according to weak classifier parameters. The score mean and variance are calculated incrementally. Both the scores and NCC values are added to a priority queue. The priority queue captures the top 128 maximum values across the frame. These values are outputted as the result.

The image scaler uses bilinear interpolation to scale. Four weighted pixels are used to generate a single output pixel. This provides arbitrary precision output scaling. The scaled pixel data is converted into integral image format and both streams are passed to parallel sliding window buffers. During conversion, only the least significant 19 bits of each integral image pixel are retained. This does not affect Haar feature calculation as higher order bits will be subtracted away during calculation. The two sliding window buffers move in unison to maintain the same window position across the frame. The image sliding window provides data for normalized cross correlation (NCC) calculation. The integral image sliding window does the same for Haar feature extraction and classifier scoring. An annular region filter determines if the current window is within a circular radius or annual region, for both calculations.

The sliding window buffers provide a new column of 45 pixels every cycle. For both calculation data paths, each new column of pixel data is buffered into a BRAM with a 45 pixel wide data port. After 45 cycles, an entire window has been buffered. Each subsequent cycle provides a new column of pixels and results in a new window. Column data is stored in circular manner in the window BRAMs.

For each newly buffered window, the NCC calculation processes the 45 saved columns, one column per cycle. It squares the column data and cross correlates it with a column of template pixel data. Template data has the same dimensions as a window and is accessed from a separate BRAM. This is done with 45 parallel modules. The results are summed in adder trees and then added to separate running sums. It requires 45 cycles to access all columns of the data and process a single window. The NCC values are then streamed to a priority queue.

Concurrently, Haar feature values are computed in the feature scoring computation data path. Six parallel modules access Haar rectangle coordinates from independent window data and parameter BRAMs. This allows a new Haar value to be calculated each cycle with up to six rectangles per feature. The Haar rectangles are weighted and summed via an adder tree and the resultant Haar feature values are simultaneously outputted and streamed to the Mean & Variance and the Scoring module. The design supports up to 64 Haar features during evaluation. We used 50 features in our experiments. Therefore, it takes 50 cycles to calculate all the Haar features for each window.

The Mean & Variance module calculates and outputs the $E(x)$ and $E(x^2)$ values for each Haar feature across all windows. The Scoring module uses the Haar feature values and classifier parameters to score each window according to Equation 2. These window scores are then streamed to a priority queue. Both modules use floating point operators to preserve precision over the wide range of values each output takes. The algorithm is sensitive to small changes in value. A fixed point data path with sufficient accuracy to accommodate output values would require an unreasonably large number of bits.

The priority queues for both data paths are minimum priority queues, 128 values deep. They keep the NCC and score values in a partial sorted order. The minimum value is always maintained. Values are paired with the window position to which they correspond. New values are added to the queue as they are generated. After the queue fills, the lowest value is removed to make room for the next new value. At the end of processing all windows, the queues hold the top 128 maximum values. The queues are then drained and outputted in increasing value order.

B. Update stage

The Update stage uses the the Haar feature means of the positive and negative examples, calculated during the Evaluate stage, to update the current classifier parameters. These new parameters define the weak classifiers that can be boosted during training into the next classifier. After each parameter is updated, it is used to score the Haar feature values from the Evaluate stage. This requires iterating over the Haar features in feature major order instead of window major order as they were generated. Scoring performs the same calculations as in the Evaluate stage, but uses the updated parameters. The parameters are outputted and the window scores are stored for the Train stage. Figure 5 illustrates this process.

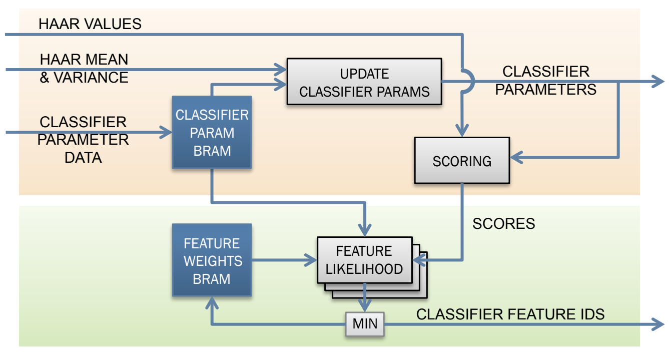

Fig. 5: Update state (top) and Train stage (bottom) architectures. The Update stage updates classifier parameters using the positive and negative examples. It also scores all examples using the updated parameters. The Train stage uses the scored examples to iteratively select the best weak classifiers.

C. Train stage

The Train stage calculates the log likelihood of each positive and negative example window using the scores from the Update stage. Log likelihoods for each weak classifier are summed over examples. Then the weak classifier which contributes the minimum negative log likelihood across all examples is added to the classifier. This is an iterative process that evaluates 256 weak classifiers to select 50 for inclusion in the classifier. The log likelihood calculation is performed by 16 parallel modules that operate on positive and negative examples. This parallelization factor was selected to balance the runtime between stages. Floating point operators are used for the exponentiation, logarithm, and division in this calculation. After each iteration, the selected weak classifier id is outputted and the intermediate classifier is updated. Figure 5 illustrates the high level architecture. Further details on the Train and the Update stages can be found in our previous work [7].

V. Experimental Results

The FPGA-CPU design is implemented in Verilog on a Xilinx Virtex 7 VC707 using Vivado 2013.3. It runs at a frequency of 250 MHz. It is connected to a 4 core Intel i7 3.6 GHz system with 16 GB RAM via a x8 PCIe Gen 2 slot. The portion of the algorithm running on the CPU is written in C++. The software primarily renders video and performs bookkeeping tasks. Multiple threads are used to coordinate communication with the FPGA. The FPGA design resource utilization is listed in Table I

TABLE I: FPGA design resource and VC707 utilization.

Slice Reg.	Slice LUT	BRAM	DSP48E
181541	135169	520	320
30%	45%	50%	11%

A software-only implementation of our algorithm was also written in C++ and run on the same computer. It is multi-threaded with the OpenMP library and uses Intel Integrated Primitives vector instructions. This highly optimized software-only implementation can run a single tracker at 17 FPS. Our algorithm uses three trackers along with up to 8 additional classifiers each frame. When running the full algorithm, the software-only implementation runs at 2 FPS. The FPGA-CPU implementation runs the full algorithm with all trackers and classifiers at 60 FPS. This represents a 30$\times$ speed up over the software-only version.

To evaluate the accuracy of the FPGA design, we tested the implementation using a set of standard tracking videos from recent publications [5], [10], [17]. Tracker error is measured in terms of distance (in pixels) between the center pixel and a manually determined ground truth. The average pixel error for the entire sequence is provided in parentheses. We compare our algorithm against that proposed by Babenko (MILTrack) as well as several other state of the art algorithms. Figures 6[a-f] show the error plots.

The plots show that our algorithm performs very well on the Occluded Face, David, and Coke Can video sequences. The Occluded Face sequences track a face while it is repeatedly occluded and revealed. During occlusion, the tracking classifiers inevitably update to track features on the occluding surface. Each time the face is revealed, these classifiers drift off the face. The error plot shows this behavior is consistent across all the tracking algorithms. However, our pose-specific classifiers detect the face location as soon as it is revealed and quickly re-centers the tracker.

In the David video sequence, the target is again a face. But this face rotates in and out of plane and the scene lighting changes considerably. Additionally, eyeglasses are present at the beginning of the sequence and are later removed. Again, the tracking classifiers perform well but are prone to drift and loss. Many different poses of the face are learned during the sequence. This helps recover from loss several times, as illustrated by the sharp drops in error.

The Coke Can video tracks a hand held can of soda as it is spun, rotated, moved between different lighting conditions, and behind house plants. The rigid shape of the can provides a good tracking feature for all the algorithms. Pose-specific classifiers quickly learn the few poses the target can take and provide quick recovery after occlusion or temporary changes in appearance. This recovery improves the accuracy considerably over the MILTrack algorithm.

In the Tiger2 video sequence, the performance is about the same as the MILTrack algorithm. This sequence tracks a stuffed animal tiger as it's moved in and around plant foliage. The target is occluded quite a bit of the time by different plant leaves and the motion is fast and erratic. As a result, no pose-specific classifiers are given the opportunity to train. Even when the training threshold is reduced, the plant occlusion makes it extremely difficult to detect similar poses in the sequence. Thus, the performance of our algorithm is on par with MILTrack.

The Sylvester sequence exhibits target loss during tracking and highlights a weakness in our algorithm. The Sylvester sequence tracks a stuffed cat as it is articulated and flown around a room. The scene is cluttered and has extremely high lighting variation. The motion is rapid, but there is occasionally enough stability for pose-specific classifiers to train. This provides recovery several times in the sequence. However the back and forth motion challenges the tracking classifiers. The majority aggregation function for the tracking classifiers suppresses spurious high scoring incorrect detections. But this

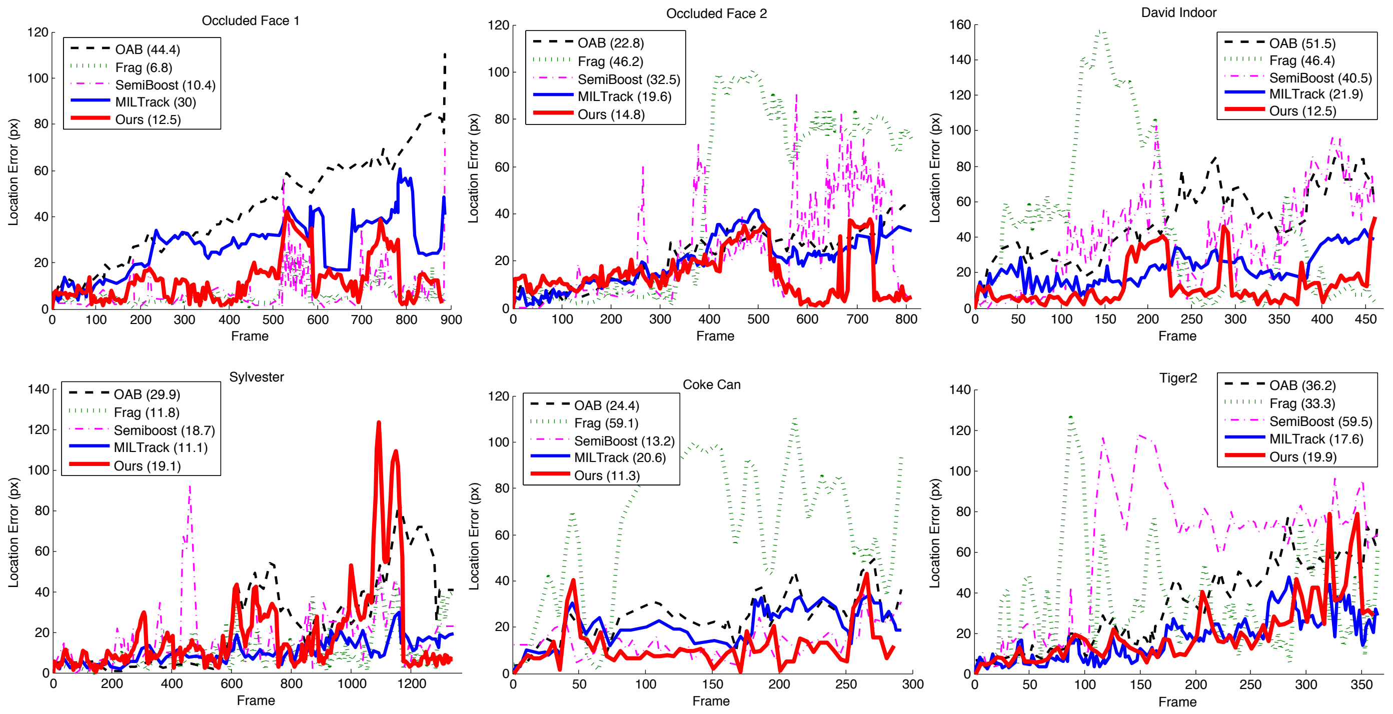

Fig. 6: Location errors on video sequence from several tracking publications. Error is the difference between predicted tracking location and the ground truth, in pixels. Average pixel error over the entire sequence is shown in parentheses. Algorithms compared include: Online Adaboost (OAB) [4], FragTrack (Frag) [10], Semi-supervised Online Boosting (SemiBoost) [9], and MILBoost Tracker (MILTrack) [5]. Video sources are: Occluded Face 1 [10], Occluded Face 2 [5], David Indoor [17], Sylvester [17], Coke Can [5], and Tiger2 [5].

can impose a lag when reacting to motion as the majority of the classifiers must agree to the change. In this video sequence, this lag results in drift and then loss. It should be noted, that a pose-specific classifier recovers from the target loss before the end of the sequence.

Across all but one of the video sequences, our algorithm performs equal to or significantly better in terms of average accuracy. These sequences come from several different tracking publications. They were selected based on the availability of results from existing algorithms, not based on the performance of our algorithm. They are difficult sequences representative of many appearance changing scenarios.

VI. Conclusion

We have provided a FPGA-CPU accelerated design for tracking objects through appearance changes, using multiple online boosted classifiers. The design accelerates our algorithm for learning a pool of pose-specific and tracking classifiers at runtime. It also employs a novel method for comparing multiple classifier scores using a kurtosis of the score distributions. Compared to a multi-threaded software-only CPU based implementation, it boasts a 30× speed up. Our algorithm performs at state of the art levels and shows an improvement in accuracy over existing tracking algorithms.

References

[1] G. R. Bradski, "Computer vision face tracking for use in a perceptual user interface," *Intel Technology Journal*, 1998.

[2] S. Avidan, "Support vector tracking," *IEEE Trans. Pattern Anal. Mach. Intell*, vol. 26, no. 8, pp. 1064–1072, 2004.

[3] D. Snow, M. J. Jones, and P. Viola, "Detecting pedestrians using patterns of motion and appearance," in *ICCV*, 2003.

[4] H. Grabner, M. Grabner, and H. Bischof, "Real-time tracking via on-line boosting." in *BMVC*, vol. 1, no. 5, 2006, p. 6.

[5] B. Babenko, M.-H. Yang, and S. Belongie, "Robust object tracking with online multiple instance learning," *TPAMI*, 2011.

[6] P. A. Viola and M. J. Jones, "Robust real-time face detection," in *ICCV*, 2001, p. 747.

[7] M. Jacobsen, P. Meng, S. Sampangi, and R. Kastner, "Fpga accelerated online boosting for multi-target tracking," in *FCCM*, 2014.

[8] P. A. Viola, J. C. Platt, and C. Zhang, "Multiple instance boosting for object detection," in *NIPS*, 2005.

[9] H. Grabner, C. Leistner, and H. Bischof, "Semi-supervised on-line boosting for robust tracking," in *Computer Vision–ECCV*, 2008.

[10] A. Adam, E. Rivlin, and I. Shimshoni, "Robust fragments-based tracking using the integral histogram," in *CVPR*, 2006.

[11] Z. Kalal, K. Mikolajczyk, and J. Matas, "Tracking-learning-detection," *PAMI*, 2012.

[12] C. Lo and P. Chow, "A high-performance architecture for training viola-jones object detectors," in *FPT*. IEEE, 2012.

[13] M. Arias-Estrada and E. Rodríguez-Palacios, "An fpga co-processor for real-time visual tracking," in *FPL*, 2002.

[14] J. U. Cho, S. H. Jin, X. Dai Pham, J. W. Jeon, J. E. Byun, and H. Kang, "A real-time object tracking system using a particle filter," in *Intelligent Robots and Systems*, 2006.

[15] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Perez, "Solving the multiple instance problem with axis-parallel rectangles," *AI*, 1997.

[16] M. Jacobsen and R. Kastner, "Riffa 2.0: A reusable integration framework for fpga accelerators," in *FPL*, 2013.

[17] D. A. Ross, J. Lim, R.-S. Lin, and M.-H. Yang, "Incremental learning for robust visual tracking," *IJCV*, 2008.