FPGA Accelerated Online Boosting for Multi-Target Tracking

Matthew Jacobsen, Pingfan Meng, Siddarth Sampangi, Ryan Kastner
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
{*mdjacobs, pmeng, ssampang, kastner*}*@cs.ucsd.edu*

***Abstract*—Robust real time tracking of multiple targets is a requisite feature for many applications. Online boosting has become an effective approach for dealing with the variability in object appearance. This approach can adapt its classifier to changes in appearance at the cost of additional runtime computation. In this paper, we address the task of accelerating online boosting for multiple target tracking. We propose a FPGA hardware accelerated architecture to evaluate and train a boosted classifier in real time. A general purpose CPU based software-only implementation can track a single target at 17 frames per second (FPS). The FPGA accelerated design is capable of tracking a single target at 1160 FPS or 57 independent targets at 30 FPS. This represents a 68× speed up over software.**

***Keywords*-FPGA, Tracking, Online Boosting**

I. Introduction

Robust object tracking is a critical component for many applications. Input and gesture recognition for human-device interaction [1], autonomous vehicle systems [2], and video surveillance [3] all require accurate tracking input. Many of these practical applications require tracking multiple independent targets at once, with low latency, several times a second. Tracking effectively translates high bandwidth sensor information into a low bandwidth set of data points for higher level algorithms. This is a challenging task because an object's appearance can change over time. Small changes in lighting, occlusions, deformations, or rotations can have a dramatic effect.

Boosting has been employed in machine learning applications with considerable success. Classifiers of boosted Haar features are commonly used as face and object detectors. Training these classifiers is typically performed offline with many training examples over several rounds. However, research has shown that online boosting can be very effective for object tracking [4], [5]. In contrast to traditional offline boosting, online boosting gathers examples at runtime and trains a classifier incrementally. This approach provides training examples from the current environment and can result in a more adaptive and accurate classifier. These benefits come with the cost of additional runtime computation.

In this paper, we consider the task of accelerating an online boosting based tracker capable of tracking multiple independent targets. We evaluate an online boosting algorithm for robust tracking and implement a FPGA accelerated hardware design. We compare this design against two software-only implementations on a general purpose CPU. We continue with a discussion of related work, the online boosting algorithm, our tracking application, and a description of the FPGA accelerated design. Experimental performance results and analysis are presented.

II. Related Work

Much of the research in online boosting for tracking focuses on improving the algorithms. While there are numerous hardware accelerated tracking applications, they focus on the evaluation of trained classifiers, not the training. Online boosting requires evaluation and training to be completed at runtime. The complexity and iterative dependency of training makes it difficult to parallelize. To our knowledge, there are no hardware accelerated designs for online boosted tracking in the literature at the time of this writing.

Accelerating boosted classifier training has been addressed by Lo [6]. They present a FPGA based architecture to reduce the time to train Viola-Jones style classifiers. They achieve a 14× speed up over a CPU. This work is similar to our own, but deals only with accelerating offline training.

Heinzle et al. describe related work to accelerate computational stereo camera processing in [7]. Their design uses a FPGA, GPU and CPU to process the stereo data in real time. Online boosted tracking is employed in their framework. However the tracking is run on the CPU. The authors point out that their system would benefit from a FPGA accelerated online boosting tracking implementation.

III. Algorithm

The tracking algorithm we employ is proposed by Babenko [5]. It is an online boosting algorithm for tracking based on MILBoost [8]. It was selected because of its robust appearance model. The algorithm consists of two steps: find the new target location, then update the trained classifier.

For each new image frame, a region surrounding the last known location is evaluated. Evaluation yields a new location. Then the region surrounding the new location is used to select positive and negative training examples. These

Algorithm 1 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 $l^*_t = l(\text{argmax}_{x \in X^s} \boldsymbol{H}(x))$
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
$\boldsymbol{H} = Train(X_1, X_0)$

examples are used update the classifier for the next frame. This tracking flow is illustrated in Algorithm 1.

The algorithm is a *two pass* algorithm in the sense that it must access the image frame twice. First to search for the new location. Then again, after the new location is found to gather training examples. The passes must take place sequentially. The second pass cannot begin until the new location has been found. A sequential dependency exists between image frames as well. The next frame cannot be evaluated until the classifier has been trained from examples drawn from the current frame.

We continue with a detailed explanation by describing the three basic components common to most tracking algorithms: the motion model, the search strategy, and the appearance model.

A. Motion Model

The motion model assumes the object location at time t will be within a radius, s, from the object location at time $t-1$. Each location within radius s has equal probability.

B. Search Strategy

All locations within radius s are evaluated. The location, $l(x)$, with the maximum classification value is selected as the new object location, l^*. This greedy strategy lends itself well to parallel execution as there is no data dependency between locations during evaluation.

C. Appearance Model

The main idea of boosting is to combine several weak classifiers, h, into a strong classifier, $\boldsymbol{H}$. This is achieved by iteratively maximizing the log likelihood of the strong classifier. At each iteration, the existing strong classifier is combined with the next weak classifier, h, that maximizes this quantity over the training data.

The algorithm makes use of an additional technique called Multiple Instance Learning (MIL) [9] to improve classifier robustness. The idea is to group samples together into sets (called *bags*) and train using the sets instead of the samples directly. We use just one positive and one negative set.

MIL mitigates the problem of correctly labeling samples as they are collected. This is achieved by setting the positive training set to all the samples in a tight radius, r, around the newly found object location. The negative training set is a sparse sampling from an annular region between two radii, q and $q\prime$, surrounding the newly found location. See Figure 1 for an illustration of these regions.

Figure 1: Circular search region with radius s (left). Circular region with radius r for positive examples and sampled annular region with radii q and $q\prime$ for negative examples (right).

The algorithm uses Haar-like features, similar to those used for face and object detection [10]. 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. These additional parameters quantify the degree to which the feature represents the object being tracked. They are updated during training using positive and negative samples. At runtime, each weak classifier predicts the likelihood of a sample window using the log odds ratio between positive and negative examples. This formula results in higher scores the closer a sample's feature value is to the mean of the positive examples.

IV. FPGA Design

Our design is motivated by a human computer interaction application. The application requirements are to track at least three independent points at 30 frames per second (FPS). It must also evaluate frames at three different scales to track objects as they move closer and further from the camera.

The algorithm was initially implemented in software and profiled. Our FPGA design accelerates the slowest tasks, identified in Figure 2. The largest amount of time (54.5%) is spent training the classifier. Boosting is inherently sequential. Each training iteration must wait for the previous iteration to complete before starting the next.

The FPGA design is partitioned between software and hardware. The software is responsible for acquiring image frames, passing data to the FPGA, receiving data from the FPGA, and keeping track of state. The algorithm initializes in software. Targets of any size can be tracked, but they are scaled to fit a fixed hardware window size of 20×20. CPU-FPGA communication is achieved using RIFFA [11].

For each new 8 bit grayscale image frame, a square region of the frame surrounding the current location is cropped and sent to the FPGA along with parameters. The

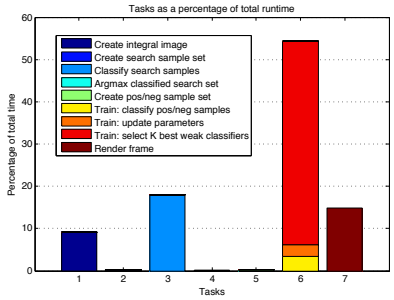

Figure 2: Duration of tasks in the algorithm as a percentage of total time.

square region includes only the pixels within the search radius s. Additional parameters define the Haar features, weak classifiers, radius, and scaling factors. The FPGA architecture is illustrated in Figure 3.

A. Target Search

Parameters and image data are sent from the PC to the FPGA. In this pass, only *Stage 0* is run. Stage 0 streams frame data through three parallel classification pipelines. Each classification pipeline scales the frame data independently, converts the scaled data into integral image data, and runs it through a systolic sliding window. Image scaling resizes the target to fit within a 20×20 pixel window. The scaling module uses Block RAM (BRAM) to buffer lines so that four pixels can be used to interpolate each output pixel. An integral image is calculated on the scaled image. Only the 17 least significant bits of each integral image pixel are kept. This has no affect on Haar feature calculation as higher order bits will be subtracted away.

Integral image pixels are streamed to a sliding window architecture, similar to that employed by Cho [12]. It is a systolic architecture where each new pixel generates a new vertical column of pixels in the window. A two dimensional register array represents the current window so that any arrangement of pixels can be accessed each cycle. The register window is accessed by six parallel Haar rectangle extractor modules. This allows each classification pipeline to calculate one Haar feature every cycle. Radial and annular window filtering is accomplished by tracking window positions within the cropped region.

After the Haar features are calculated, they are scored by the *WeakClassifier* module. This module is fully pipelined and produces a new value every cycle. The values are summed to achieve a single value per window location. We avoid division, square root, and logarithm operations by precomputing multiplicative coefficients on the PC. Floating point operators are still used however because feature values take on a wide range of values. The algorithm's ability to discriminate between these values directly impacts its accuracy. Finally, the window with the maximum feature value across all three classification pipelines is returned to the PC. This is used as the new target location.

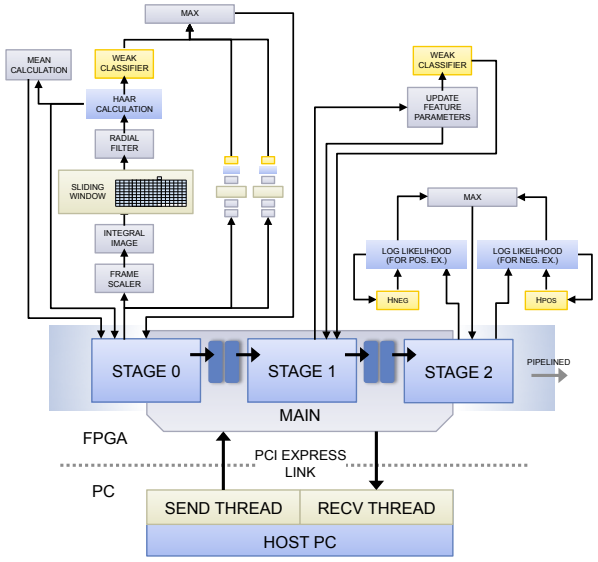

Figure 3: FPGA architecture. Stage 0 evaluates the classifier (at 3 scales). Stages 1 and 2 update and train the classifier respectively. During the search pass, only Stage 0 is used. All Stages are used during the training pass.

B. Classifier Training

Training has dependencies between iterations and also within each iteration. Log likelihood values depend on all the weak classifier scores, which depend on all the updated feature parameters which depend on Haar values. To accommodate these dependencies, we split the data path into three *Stages* that can run concurrently. Data generated between Stages are stored in FIFOs.

As before, Stage 0 calculates Haar feature values for each sample location. In addition, the mean, $E(X)$, and squared mean, $E(X^2)$, for each Haar feature are calculated incrementally as values are generated. *Stage 1* uses the saved Haar values and means to update feature parameters and calculate weak classifier scores for each window. The updated feature parameters are also sent to the PC. *Stage 2* uses these scores and parameters to calculate the log likelihood for each feature. The log likelihoods are calculated using 16 parallel pipelines of floating point operators. This preserves accuracy for the wide range of values resulting from the exponential, division, and logarithm operations. It also balances the runtime across stages. Stage 2 iterates K times, selecting the feature with the minimum negative log likelihood each time. It outputs K numbers in total, representing the trained classifier $\boldsymbol{H}$. The PC uses this to update its definition of $\boldsymbol{H}$ for the next frame.

V. Results And Analysis

We tested our implementations on a 4 core Intel i7 3.6 GHz system with 16 GB RAM with 640x480 resolution video. For all experiments, the feature pool size, M, was set to 250, with a classifier size, K, of 50. We set s to 35, r to 4, q to 6, and $q\prime$ to 8. Figure 4 shows the performance of

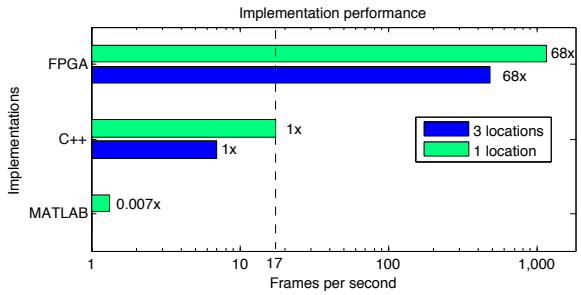

Figure 4: Performance of implementations (longer is better). Speed up over C++ software is shown to right of each bar.

our implementations. Because many practical applications do not need to render the image frame, we did not include the time to render in our measurements. Otherwise, time measurements for the performance reported include the time for running the entire algorithm, not just the kernels.

A. Software-only

We implemented the algorithm using MATLAB and C++. The MATLAB implementation is single threaded. It is capable of running the algorithm at 0.123 FPS. We did not attempt to run the algorithm with multiple targets.

Our C++ implementation is highly optimized, multi-threaded, and makes use of Intel Integrated Performance Primitives vector instructions. OpenMP is used to parallelize the application. This implementation is based on a version provided by the authors of the algorithm. It is capable of running at 17 FPS while tracking a single target and at 7 FPS when tracking three targets concurrently. Despite being multi-threaded, it performed effectively the same when pinned to a single processor due to its sequential nature.

B. FPGA

The FPGA design was built using Xilinx Vivado 2013.3 in Verilog. It was implemented on a Xilinx Virtex 7 VC707 development board and run at 250 MHz. It has a x8 Gen 2 PCIe connection to the PC. Table I lists the resource utilization of the entire design. Arithmetic results between the FPGA and software implementations differ by a maximum of 0.77%. This is due to differences in floating point operator implementations. This difference did not affect final values.

The FPGA implementation communicates with a C++ application. However, as all the image processing is performed on the FPGA, the software only performs bookkeeping functions. It is capable of tracking a single target at 1160 FPS, three targets at 480 FPS, and 57 targets at 30 FPS. When run independently, the kernels that evaluate windows and train the classifier run at 6635 FPS and 1383 FPS respectively.

C. Comparison

The FPGA implementation runs 68× faster than the C++ CPU implementation. This is due to the manually scheduled pipelined data path and dedicated parallel operators. Although multiple targets are processed sequentially by the FPGA, Stage modules can overlap in time. This contributes most significantly to the better performance. The data path has also been parallelized at bottlenecks to balance pipeline stage runtimes.

Table I: FPGA design resource and VC707 utilization.

Slice Reg.	Slice LUT	BRAM	DSP48E
187505	231971	224	304
31%	76%	22%	11%

The runtime performance of the FPGA design is dependent on the size of the target being tracked. Larger targets require more pixels to be transferred and more time to scale on the FPGA. In our experiments, we tracked targets with sizes between 20 and 40 pixels square without any significant change in performance. Runtime performance also scales sub-linearly with additional targets. For the same period of time, the total number of tracked targets is higher at lower frame rates.

VI. Conclusion

In this paper, we have addressed the task of accelerating an online boosting based multi-target tracker. We have proposed and evaluated a FPGA accelerated design. It achieves a 68× speed up over a general purpose CPU based, highly optimized, software-only C++ implementation. The FPGA design is capable of tracking a single target at 1160 FPS or 57 independent targets at 30 FPS.

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*, 2006, p. I:47.

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

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

[7] S. Heinzle, P. Greisen, D. Gallup, C. Chen, D. Saner, A. Smolic, A. Burg, W. Matusik, and M. H. Gross, "Computational stereo camera system with programmable control loop," *ACM Trans. Graph*, vol. 30, no. 4, p. 94, 2011.

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

[9] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Perez, "Solving the multiple instance problem with axis-parallel rectangles," *Artificial Intelligence*, vol. 89, pp. 31–71, 1997.

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

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

[12] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fpga-based face detection system using haar classifiers," in *FPGA*, 2009.