A HIGH-PERFORMANCE ARCHITECTURE FOR TRAINING
VIOLA-JONES OBJECT DETECTORS

by

Charles Lo

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by Charles Lo

Abstract

A High-Performance Architecture for Training Viola-Jones Object Detectors

Charles Lo

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2012

The object detection framework developed by Viola and Jones has become very popular due to its high quality and detection speed. However, the complexity of the computation required to train a detector makes it difficult to develop and test potential improvements to this algorithm or train detectors in the field.

In this thesis, a configurable, high-performance FPGA architecture is presented to accelerate this training process. The architecture, structured as a systolic array of pipelined compute engines, is constructed to provide high throughput and make efficient use of the available external memory bandwidth. Extensions to the Viola-Jones detection framework are implemented to demonstrate the flexibility of the architecture. The design is implemented on a Xilinx ML605 development platform running at 200 MHz and obtains a 15-fold speed-up over a multi-threaded OpenCV implementation running on a high-end processor.

Acknowledgements

First, I would like to thank Professor Paul Chow for the opportunity to work on this project and guidance throughout both my undergraduate and graduate studies. Thank you for providing the freedom to explore this research as well as your invaluable advice. Further, I would like to thank Daniel Le Ly for providing some of the early sparks of interest to explore machine learning and reconfigurable computing.

I would also like to thank everyone that I have had the pleasure of working with in the Chow group during my time here: Taneem Ahmed, Bo Bao, Fernando Martin del Campo, Tan Chia Chen, Xander Chin, David Hastings, Alireza Heidarbarghi, Keiming Kwong, Eric Lau, Sung Hoon Lee, Yuan Li, Jimmy Lin, Hao Jun Liu, Chris Madill, Michael Manning, Vince Mirian, Arun Patel, Professor Yuanxi Peng, Paolo Piguing, Wahid Rahman, Manuel Saldaña, Andrew House, Kam Pui Tang, Jasmina Vasiljevic, Rudiger Willenberg, Ling Zhong and Jay Zhou. You have all helped make this a fun and memorable experience.

I would like to acknowledge CMC Microsystems, NSERC and Xilinx for providing funding and support without which this research would not have been possible.

Finally, I would like to thank my family for their unwavering support.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	2
1.3	Overview	3
2	Background	4
2.1	Object Detection	4
2.2	Ensemble Classifier	5
2.3	Training Data	6
2.4	Committees	7
2.4.1	Discrete AdaBoost	8
2.4.2	Gentle AdaBoost	10
2.4.3	Weak Classifiers	12
2.5	Cascade Classifier	17
2.6	Computational Complexity	18
2.7	Related Work	20
3	Design Goals, Scope and Data Structures	21
3.1	Design Goals	21
3.2	FPGA Characteristics	22
3.3	Design Scope	24
3.3.1	Design Partitioning	24
3.3.2	Supported Extensions	25
3.3.3	Data Representation	26
3.4	Weak Classifier Training	27
3.4.1	Committee Testing	31

4	Training Architecture	34
4.1	Platform Architecture	34
4.2	Compute Core Architecture	36
4.3	Memory Management	38
4.3.1	External Memory Layout	38
4.3.2	Integral Image Requests	39
4.4	Weak Classifier Training Operation	41
4.5	Hardware Microarchitecture	42
4.5.1	Feature Evaluator	43
4.5.2	Sorted Tables	44
4.5.3	Threshold Selector	44
4.5.4	Update Calculator	45
4.5.5	Update Evaluator	46
5	Results and Analysis	47
5.1	Test Platforms	47
5.2	Training Parameters	48
5.3	Performance Scaling	49
5.4	Hardware Utilization	52
5.5	Training Set Size Scaling	54
5.6	Search Range Reduction	54
5.7	Feature Ordering	55
6	Conclusions and Future Work	57
6.1	Conclusions	57
6.2	Future Work	58
	Appendices	60
A	Object Detector Training Core MPI Protocol	61
B	Feature Evaluator Configuration	64
C	Table Address Generation	65
D	Threshold Selector	66

E Update Calculator	68
Bibliography	70

List of Tables

2.1	Time Complexity of Committee Training	19
3.1	Memory Requirements for Weak Classifier Training	27
4.1	Example Weak Classifier Training Operation	43
5.1	Hardware Speed-Up over OpenCV with 30 Engines	50
5.2	Hardware Discrete AdaBoost Resource Utilization	52
5.3	Hardware Gentle AdaBoost Resource Utilization	53
A.1	Object Detector Training Core Command Format	61
A.2	Object Detector Training Core Parameters	62
A.3	Object Detector Training Core Result Format	63

List of Figures

2.1	Image Window Scanning	5
2.2	Viola-Jones Classifier Ensemble	6
2.3	Committee Training Overview	8
2.4	Weak Classifier Structure	12
2.5	Rectangular Features	13
2.6	Calculation of Integrated Rectangular Areas	14
2.7	Feature Evaluation Equations	15
2.8	Decision Stump Discrete and Continuous Outputs	16
2.9	Training Algorithm Flow	19
3.1	Filtering for Sorted Subsets	28
3.2	Creation of Low-Precision Bins	29
3.3	High Precision Phase	30
3.4	Summary of the Two-Phase Search	31
4.1	High-Level Block Diagram of Hardware Accelerator Interfaces	35
4.2	Block Diagram of Compute Core Structure	37
4.3	Effect of Burst Ordering for Standard Features	40
4.4	Standard and Rotated Pixel Alignment Methods	40
4.5	Effect of Burst Ordering for Rotated Images	41
4.6	Feature Evaluator Engine	44
4.7	Systolic Comparison Method	45
5.1	Speed-up of Discrete AdaBoost Hardware over OpenCV	50
5.2	Speed-up of Gentle AdaBoost Hardware over OpenCV	51
5.3	Hardware Discrete AdaBoost Resource Utilization Scaling	53
5.4	Hardware Gentle AdaBoost Resource Utilization Scaling	53

5.5	Effect of Training Set Size on Performance	54
5.6	Effect of Search Range on Performance	56
5.7	Performance Effects of Feature Ordering	56
B.1	Feature Evaluator Configuration Word	64
C.1	Sorting Address Generation	65
D.1	Threshold Selector Implementation	67
E.1	Block Diagram of Update Calculator for Gentle AdaBoost	71

Acronyms

BRAM Block Random Access Memory

BL8 Burst Length 8

BC4 Burst Chop 4

DDR3 Double Data Rate Type 3

DSP Digital Signal Processing

GPC General Purpose Cluster

GPU Graphics Processing Unit

FF Flip-Flop

FPR False Positive Rate

FIFO First In First Out

FPGA Field Programmable Gate Array

HPC High Performance Computing

HR Hit Rate

LUT Look-Up Table

MIG Memory Interface Generator

MPI Message Passing Interface

PCIe Peripheral Component Interconnect Express

PLI Piecewise Linear Interpolation

RAM Random Access Memory

SODIMM Small Outline Dual In-line Memory Module

TBB Thread Building Blocks

TDP Thermal Design Power

Nomenclature

α	The output weight of a weak classifier
$\bar{x}$	The validation example integral or rotated integral image data
$\bar{y}$	The validation example label
β	The Discrete AdaBoost weight update factor
$\Delta\epsilon$	The uncertainty in misclassification error
$\Delta\epsilon^u$	The uncertainty in misclassification error directed up
ϵ	The misclassification error
ϵ^d	The misclassification error directed down
ϵ^u	The misclassification error directed up
$\hat{\epsilon}$	The squared error
$\hat{h}(x)$	The output of a real-valued weak classifier for image x
ϕ	The committee threshold
θ	The weak classifier threshold
$f(x)$	The feature value for image x
H	The aggregate committee output on the validation set
$h(x)$	The output of a binary weak classifier for image x
M_H	The number of elements in a high-precision sorted table
M_L	The number of elements in a low-precision sorted table

N	The total number of training examples
N^n	The number of negative training examples
N^p	The number of positive training examples
N_A	The sum of negative weights above the weak classifier threshold
N_B	The sum of negative weights below the weak classifier threshold
P_A	The sum of positive weights above the weak classifier threshold
P_B	The sum of positive weights below the weak classifier threshold
T	The maximum number of weak classifiers in a committee
V	The total number of validation examples
V^n	The number of negative validation examples
V^p	The number of positive validation examples
W	The number of weak classifiers to train
w	The training example weight
w^n	The negative example weights at a table bin
w^p	The positive example weights at a table bin
x	The training example integral or rotated integral image data
y	The training example label

Chapter 1

Introduction

1.1 Motivation

Object detection is an important and challenging task in Computer Vision. It involves searching an image of a scene and determining if and where a particular object exists. Object detection is used in many contexts including surveillance [1] and autonomous systems [2].

Construction of an object detector with good detection performance is difficult. Objects may appear at different locations, scales, orientations and be under various lighting conditions. Furthermore, detection must typically be carried out very quickly in real-time environments.

One of the seminal developments in this field was the object detector by Viola and Jones [3]. Their detector involves scanning an image window across a scene and evaluating a fast binary classifier at each point to localize objects. The binary classifier takes an image window as its input and identifies whether it contains the desired object or not. It is constructed by training a large number of weak classifiers and, over a number of iterations, carefully selecting a subset of them to form a stronger *ensemble* classifier. This technique proved to be robust for frontal face detection and provided good detection run-time performance. It has enjoyed a great deal of popularity and continues to be adapted and applied to detection problems [4].

A major disadvantage of their algorithm is the long training time associated with selecting the weak classifiers during the construction of the ensemble. The original system took several weeks to train and even on modern computers, training time may range from hours to days. This training time cost makes exploring enhancements [5] to the origi-

nal algorithm difficult. In addition, the original algorithm does not define how to trade off effectively between the computational complexity of the final detector and detection performance. Work has been done attempting to direct the construction of detectors by parameterizing the trade-off between computational complexity and detection performance [6]. However, creating a detector with specific characteristics can still involve multiple training iterations.

In addition to training static detectors, it has been shown that adapting detectors to field environments can improve their performance [7]. However, such applications could require re-training in field environments where the detector has already been deployed. In such situations, the limited computational capabilities of low-power embedded processors further increase the training time. Therefore, accelerating this training phase in a power-efficient manner is important to facilitate further exploration of this detector technique and allow for object detectors to be quickly trained in field environments.

During detection, the weak classifiers of the ensemble may be evaluated in parallel, providing an opportunity for acceleration. Accordingly, there have been a number of implementations of the Viola-Jones algorithm on various hardware platforms [8, 9, 10]. These designs focus on improving the run-time performance of a trained classifier for use in real-time systems.

Parallelism may also be exploited during training, where many weak classifiers are trained independently and one is selected to join the ensemble. However, the parallel tasks involved in training are more complex than those in detection. The large amount of resources in modern Field Programmable Gate Arrays (FPGAs) affords the opportunity to accelerate training in hardware as well. An FPGA offers several potential benefits as an acceleration platform. Its power efficiency relative to multi-core processors makes it amenable to training classifiers in the field. In addition, its reconfigurability relative to Application Specific Integrated Circuits allows modifications to be made to the training algorithm. However, to the best of the author’s knowledge there has been no previous work targeting the acceleration of training in dedicated hardware.

1.2 Contributions

In this thesis, a high-performance FPGA architecture is introduced for training Viola-Jones style object detectors. The primary contributions are as follows:

- Complexity analysis of the Viola-Jones training algorithm
- A method of restructuring the training of weak classifiers in a two-phase algorithm that allows the training algorithm to be efficiently mapped to FPGA resources for parallelization.
- A technique for ordering the evaluation of weak classifiers to optimize the use limited external memory bandwidth.
- A flexible, high-throughput hardware architecture for training object detectors
- Implementations of two weak classifier training techniques through reconfiguration of the hardware architecture
- A method of estimating non-linear training parameter updates
- Analysis of the architecture performance and comparison with an optimized, multi-threaded software implementation

1.3 Overview

The rest of this thesis is organized as follows: Chapter 2 provides background on the fundamentals of the Viola-Jones training algorithm. The approach used to map the algorithm onto FPGA resources is discussed in Chapter 3. Chapter 4 describes the implementation details of the hardware architecture. Results are presented in Chapter 5 and Conclusions given in Chapter 6.

Chapter 2

Background

In this chapter, the procedures and terminology for training Viola-Jones style object detectors will be presented. For a more detailed discussion, the reader is referred to the works of Viola and Jones, and Lienhart et al. [3, 11].

2.1 Object Detection

Object Detection is the task of locating instances of a specific object in a general scene. The Viola-Jones detector accomplishes this by translating an *image window* across an image of a scene and determining whether the desired object exists at each position. The left half of Fig. 2.1 shows an image window, desired cross-shaped object, and the image scene they are contained within. The right half of the figure shows the image window sliding across the scene as detection is progressing. At its rightmost position, the window is centered on top of the cross and the detector should identify that position as containing the object. This search, through translating an image window, is repeated at different scales to ensure that objects are detected invariant of scaling.

At each position in the scene, a *binary classifier* is evaluated on the image window producing a *positive* output if the window contains the desired object or a *negative* output if it does not. The positive outputs will be denoted as $+1$ and negative outputs as -1 . This classifier is used many times as the image window is scanned across a scene to locate objects. Thus, it must be fast to evaluate and have good detection performance.

Some common metrics of detection performance are *hit rate* (HR) and *false positive rate* (FPR). When the object detector is evaluated on a known, labelled set of test images, these metrics are calculated to give an estimate of the detector's performance

Figure 2.1: Image Window Scanning

on general scenes. The hit rate is the percentage of desired objects that are correctly identified. On the other hand, the false positive rate is the percentage of image windows searched, not containing desired objects, but producing positive classification outputs. A good detector has high hit rate and low false positive rate. Equations 2.1 and 2.2 summarize these metrics.

$$HR = \frac{\text{Positive Examples Identified as Positive}}{\text{Total Positive Examples}} \quad (2.1)$$

$$FPR = \frac{\text{Negative Examples Identified as Positive}}{\text{Total Negative Examples}} \quad (2.2)$$

2.2 Ensemble Classifier

The binary classifier used by Viola and Jones is composed of two levels of hierarchy forming an *ensemble* classifier. An ensemble can provide good classification performance by combining the outputs of a diverse set of weaker classifiers.

The first level of hierarchy involves a set of *committee* classifiers. Committees consist of many *weak classifiers* that individually attempt to identify the input image window. Weak classifiers are fast to evaluate, but poorly identify the contents of the image window. Each weak classifier produces a positive or negative output scaled by a weight (α) determining the influence of the weak classifier on the committee. The committee output is formed by additively combining the weighted outputs of the weak classifiers and making a decision based on that aggregate value. The resulting committee is much better at identifying the image window than its constituent weak classifiers while still being fast to evaluate.

Figure 2.2: Viola-Jones Classifier Ensemble

In the second level of hierarchy, the committee outputs are *cascaded* such that the final binary classifier result may only be positive if all of the committees produce a positive result. The committees of the cascade are evaluated sequentially, such that each acts as a filtering step where only image windows classified as positive by all earlier committees reach the next one in the cascade. Thus, the detector can quickly classify an image window as negative if any of the committees produces a negative output. Since objects are typically rare with respect to the number of image windows tested when scanning a scene, this helps dramatically improve detection speed. Fig. 2.2 shows the full hierarchy of the ensemble binary classifier.

The ensemble is built in a step-wise process where first, weak classifiers are selected sequentially to form a committee. Once a committee has been formed, the current cascade is tested and a decision is made whether or not to add additional committees. The following sections will detail the methods used to construct the full binary classifier.

2.3 Training Data

Training the binary classifier involves constructing the ensemble to have good predictive performance on a known, labelled, *training set*. This training set usually contains thousands or tens of thousands of *examples*, each consisting of image window data x and a label $y = \{+1, -1\}$ specifying whether the example is positive (+1) or negative (-1). It is important to have a large, diverse training set to ensure that the resulting classifier performs well on general, real-world images. In addition to the training set, a separate,

smaller set of labelled data called a *validation set* is often collected. This validation set is not used to adjust the parameters of the classifier, but instead is used to test the detection performance of the classifier as it is being trained.

Training images are created containing either centered examples of the desired objects or random backgrounds. In object detection literature, these images are often square and greyscale. Objects in the real world are often subjected to different lighting conditions depending, for example, on the time of day or location of the object. These variations can cause problems with accurately detecting objects. Thus, a technique called variance normalization is often used to pre-process images. This involves calculating the pixel variance over an image (σ^2) and dividing each pixel p_i by the standard deviation

$$p_i = p_i / \sigma. \quad (2.3)$$

2.4 Committees

The operations involved in committee training are summarized in Fig. 2.3. Training begins with a large, diverse pool of possible weak classifiers each characterized by a rectangular feature. The committee is then slowly built up by sequentially selecting the best weak classifiers from the pool. After each weak classifier is selected, weights are applied to the training set examples such that the next weak classifier is focused on correcting the mistakes of the earlier members of the committee. The balance of this section will go into more details of first how weak classifiers are selected and second on structure of weak classifiers and features themselves.

Viola and Jones use a method called Adaptive Boosting (AdaBoost) [12] to select and combine many weak classifiers together to form committees. There are several variants of AdaBoost, each with different performance characteristics. The basic, *Discrete AdaBoost* procedure was used by Viola and Jones [3]. However, other variants have been applied to the Viola-Jones framework with success. In particular, *Gentle AdaBoost* [13] has been shown to generate better classifiers than Discrete AdaBoost in a face detector [11]. In this section, the use of Discrete and Gentle AdaBoost for committee construction will be described.

Figure 2.3: Committee Training Overview

2.4.1 Discrete AdaBoost

In Discrete AdaBoost, weak classifiers produce binary outputs. The algorithm begins by assigning an equal weight (w) to all of the training examples. A large, diverse set of weak classifiers is then trained, each attempting to minimize the weighted misclassification error (ϵ) on the training set

$$\epsilon = \sum_{i=1}^N w_i 1_{(h(x_i) \neq y_i)}. \quad (2.4)$$

Here N is the size of the training set, $h(x_i) \in \{+1, -1\}$ is the output of a weak classifier on the image data x_i and $y_i \in \{+1, -1\}$ is the label of training example i . The function $1_{(h(x_i) \neq y_i)}$ is an indicator that takes the value of 1 when $h(\cdot)$ misclassifies y_i and 0 otherwise. Notice that since the indicator function may only take on the values of 1 or 0, the calculation of misclassification error only involves addition of weights.

The weak classifier, $h_t(\cdot)$, that produces the lowest misclassification error, ϵ_t is selected to be added to the committee. Training example weights are now updated to put more

emphasis on misclassified examples

$$w_i = w_i \beta_t^{(1 - 1_{h_t(x_i) \neq y_i})}. \quad (2.5)$$

Here, $\beta_t \in [0, 1]$ is a scaling term that increases with the error of the selected weak classifier. It is determined using the equation

$$\beta_t = \epsilon_t / (1 - \epsilon_t). \quad (2.6)$$

Therefore, training examples that are correctly classified ($1_{h_t(x_i) \neq y_i} = 0$), have weights multiplied by β_t and are thus scaled *down*. On the other hand, incorrectly classified examples have zero in the exponent and thus retain the same weight, increasing their relative importance. After this adjustment, the weights are normalized such that they sum to 1. Finally, the influence weight (α_t) of the selected weak classifier is calculated to be inversely related to the error such that good classifiers have greater influence.

$$\alpha_t = \ln(1/\beta_t). \quad (2.7)$$

The process can now be repeated with updated weights to continue adding weak classifiers to the committee. These steps of training are shown in Lines 5-9 of Algorithm 1.

Each constituent weak classifier of a committee will tend to improve detection performance. However, too many weak classifiers in a single committee can lead to slow detection run time. Thus, it is important to know when to stop adding classifiers.

Training may be stopped either when some maximum number of classifiers, T , have been trained, or if the target hit and false positive rates are met for the committee. The output of the committee is based on a threshold (ϕ) that can be adjusted to arbitrarily set the hit rate on the validation set. To set the hit rate, the sum $H_j = \sum_{t=1}^T \alpha_t h_t(x_j)$ is calculated for each validation example j . This sum is the *aggregate output* of all weak classifiers in the committee so far. It is calculated iteratively by accumulating $\alpha_t h_t(x_j)$ for every validation example as each weak classifier is selected. The threshold (ϕ) is chosen by sorting the positive validation examples and setting ϕ such that the desired percentage of positive examples have H values greater or equal to ϕ .

Once ϕ is determined for the desired hit rate, the classifier is tested on the negative examples to calculate the false positive rate. Here, any negative example with H greater than or equal to ϕ would be a false positive. If the false positive rate is met, then training

Algorithm 1 Discrete AdaBoost for T weak classifiers

```

1: Given  $N$  training examples  $(x_1, y_1), \dots, (x_N, y_N)$ 
2: Given  $V$  validation examples  $(\bar{x}_1, \bar{y}_1), \dots, (\bar{x}_V, \bar{y}_V)$ 
3: Initialize:  $w_i = 1/N \forall i \in \{1 \dots N\}$ ,  $H_j = 0 \forall j \in \{1 \dots V\}$ 
4: for  $t = \{1 \dots T\}$  do
5:    $h_t(\cdot) = \arg \min_h \sum_{i=1}^N w_i 1_{(h(x_i) \neq y_i)}$ 
6:    $\beta_t = \epsilon_t / (1 - \epsilon_t)$ 
7:    $w_i = w_i \beta_t^{(1 - 1_{(h_t(x_i) \neq y_i)})}$ 
8:    $w_i = w_i / \sum_{j=1}^N w_j$ 
9:    $\alpha_t = \ln(1/\beta_t)$ 
10:   $H_j = H_j + \alpha_t h_t(\bar{x}_j)$ 
11:  SORT( $\{H_j : \bar{y}_j = +1\}$ )
12:  Select  $\phi$  for target hit rate
13:   $FPR = \frac{\sum_{j=1}^V 1_{(H_j \geq \phi \ \& \ \bar{y}_j = -1)}}{\sum_{j=1}^V 1_{(\bar{y}_j = -1)}}$ 
14:  if  $FPR < \text{target FPR}$  then
15:    break
16:  end if
17: end for
18: Final Classifier:  $S(x) = \sum_{t=1}^T \alpha_t h_t(x) \geq \phi$ 

```

can be stopped. The full algorithm is shown in Algorithm 1 and can be divided into three main phases. First, weak classifiers are trained (Line 5), second w and α parameters are updated (Lines 6-10) and finally the committee is tested to stop training early (Lines 10-16).

2.4.2 Gentle AdaBoost

Unlike Discrete Adaboost where weak classifiers produce binary outputs, the weak classifiers in Gentle AdaBoost are trained to fit a regression function to the training set, producing a real output in the range $[-1, +1]$. Once again, the algorithm initializes weights (w) to be equal for all training examples. The large set of weak classifiers is now trained to minimize the weighted squared error

$$\hat{\epsilon} = \sum_{i=1}^N w_i (\hat{h}(x_i) - y_i)^2, \quad (2.8)$$

Algorithm 2 Gentle AdaBoost for T weak classifiers

```

1: Given  $N$  training examples  $(x_1, y_1), \dots, (x_N, y_N)$ 
2: Given  $V$  validation examples  $(\bar{x}_1, \bar{y}_1), \dots, (\bar{x}_V, \bar{y}_V)$ 
3: Initialize:  $w_i = 1/N \forall i \in \{1 \dots N\}$ ,  $H_j = 0 \forall j \in \{1 \dots V\}$ 
4: for  $t = \{1 \dots T\}$  do
5:    $\hat{h}_t(\cdot) = \arg \min_{\hat{h}} \sum_{i=1}^N w_i (\hat{h}(x_i) - y_i)^2$ 
6:    $w_i = w_i e^{-y_i \hat{h}_t(x_i)}$ 
7:    $w_i = w_i / \sum_{j=1}^N w_j$ 
8:    $H_j = H_j + \hat{h}_t(\bar{x}_j)$ 
9:   SORT( $\{H_j : \bar{y}_j = +1\}$ )
10:  Select  $\phi$  for target hit rate
11:   $FPR = \frac{\sum_{j=1}^V 1_{(H_j \geq \phi \ \& \ \bar{y}_j = -1)}}{\sum_{j=1}^V 1_{(\bar{y}_j = -1)}}$ 
12:  if  $FPR < \text{target FPR}$  then
13:    break
14:  end if
15: end for
16: Final Classifier:  $S(x) = \sum_t^T \hat{h}_t(x) \geq \phi$ 

```

where $\hat{h}(\cdot)$ is the output of the real-valued weak classifier. The weak classifier with lowest squared error is added to the committee and weights are again adjusted based on the performance of the chosen weak classifier. In this case, the error calculation is much more complicated involving continuous values and a quadratic function. The weight update also takes on a different form:

$$w_i = w_i e^{-y_i \hat{h}_t(x_i)}. \quad (2.9)$$

Here, when $y_i \hat{h}_t(x_i) > 0$, the classifier is leaning towards correctly classifying the training example. Thus, this weight update reduces the weights of correctly classified examples and increases the weights of incorrectly classified examples, but influenced by how close the classifier is in its prediction. Algorithm 2 shows the equations of this procedure. Notice that the influence weighted output $\alpha_t h_t(\cdot)$ in the final classifier has been replaced with simply the real-valued $\hat{h}_t(\cdot)$. In this thesis, α will be used to denote the magnitude of the weak classifier $|\hat{h}_t(\cdot)|$ such that the same notation can be used as with Discrete AdaBoost. The computation takes the same general form as Discrete AdaBoost, with a few more complicated calculations due to the real-valued weak classifiers. Again, weak classifiers are first trained (Line 5), followed by weight updates (Lines 6-7) and finally committee testing (Lines 8-14).

Figure 2.4: Weak Classifier Structure

2.4.3 Weak Classifiers

As the basic building block of the Viola-Jones classifier, the structure of the weak classifiers is very important. Each weak classifier is composed of a *feature* and a *decision stump*. Fig. 2.4 shows this basic structure. During training, a unique feature is used to map the training images onto a one-dimensional *feature space*. The weights of the training set, whose magnitude is shown as horizontal bars in the figure, then become distributed in this feature space. In this transformed representation, a decision threshold is then found that best separates the weighted objects from non-objects or positive from negative examples. The classification decision for new images is then dependent on which side of the threshold an image is mapped to based on the feature transformation. For example, in the figure, everything above the threshold would be classified as positive. The feature transformation and threshold selection process will be described in the rest of this subsection.

Rectangular Features

In object detection, features are often used to represent certain key characteristics of an image. For instance, a feature might quantify how much brighter the left side of an image is relative to the right side. The basic structures of features used by Viola and Jones are shown on the top half of Fig. 2.5. A *feature value* is calculated for a particular image by integrating the pixels in the white rectangular areas and subtracting the integrated values in the black areas. Thus, these features measure the relative pixel intensities for various rectangular shapes. A large feature pool is generated by translating and scaling these shapes across an image window. For example, using the standard rectangular features in Fig. 2.5, over 500,000 possible features, and consequently weak classifiers, exist in a

Standard Rectangular Features

Rotated Rectangular Features

Figure 2.5: Rectangular Features

32x32 window.

To quickly evaluate these features, Viola and Jones introduced the *integral image* representation. The pixels of an integral image are formed in one pass over the original image using the following recursion

$$II(x, y) = I(x, y) + II(x - 1, y) + II(x, y - 1) - II(x - 1, y - 1). \quad (2.10)$$

Here, the top left corner of an image has coordinate $(0, 0)$, $I(x, y)$ and $II(x, y)$ are the pixel values at position (x, y) of the original and integral images respectively. Essentially, each pixel in the Integral Image is the integrated pixel values from the top left corner of the original image. By first transforming all of the training set images into this representation as a pre-processing step, the pixel intensities in any rectangular area can be calculated using four memory accesses as shown in Fig. 2.6.

Lienhart et al. [11] introduced a rotated integral image representation which is also calculated in one pass through the image. The recursive equation used is

$$RI(x, y) = I(x, y) + I(x, y - 1) + RI(x - 1, y - 1) + RI(x + 1, y - 1) - RI(x, y - 2) \quad (2.11)$$

where $RI(x, y)$ is the rotated integral image value at coordinate (x, y) . Each integral image pixel in this rotated representation represents the integrated pixel values in a triangular area with its apex at the pixel and base at the top of the image window. This representation allows the total pixel intensity in any rectangular area, rotated 45 degrees, to be calculated in four memory accesses as shown in Fig. 2.6. Thus, a set of rotated features can be constructed. The rotated features used in this thesis are shown

$$\begin{aligned} \text{Integrated Rectangular Area} \\ = A - B - C + D \end{aligned}$$

Figure 2.6: Calculation of Integrated Rectangular Areas

in Fig. 2.5, although additional shapes are easily supported. In this thesis, the symbol x will be used to refer to the integral or rotated integral image representations depending on the feature being evaluated.

Of the feature shapes chosen, the three-element rectangular features are unbalanced in terms of their white and black areas. In other words, over a uniform intensity image area, the feature would produce a non-zero value or offset. If the average intensity of the images in the training set are different than those used in the deployed detector system, this offset could result in inaccuracies in detection. To balance this feature, some implementations weigh the center region by a factor of two. Fig. 2.7 shows the complete equations used to evaluate the different features. The equations for zero-mean, three-element features are shown in brackets. These equations are created by using the equation for integrated rectangular area shown in Fig. 2.6 for each rectangular region of the desired feature. For example, the two-element feature consists of two rectangular areas, (A, B, D, E) and (B, C, E, F) . Thus, the feature value is

$$(A - B - D + E) - (B - C - E + F) = A - 2B + C - D + 2E - F \quad (2.12)$$

Notice that all of the features are calculated as linear combinations of six to nine integral image pixels.

Decision Stumps

The features specialize the weak classifiers to examine particular characteristics of the image window. In addition, the feature representation reduces the very high dimensional

Two-element Features: $A - 2B + C - D + 2E - F$

Three-element Features: $A - 2B + 2C - D - E + 2F - 2G + H$

Zero-mean Three-element Features: $(A - 3B + 3C - D - E + 3F - 3G + H)$

Four-element Feature: $A - 2B + C - 2D + 4E - 2F + G - 2H + I$

Figure 2.7: Feature Evaluation Equations

image window down to a single dimension. The classification decision is made by operating in this one-dimensional feature space. In particular, a *Decision Stump* classifier splits the space into two sets on either side of a *threshold* value, θ . Feature values greater or equal to the threshold value are denoted as being *above* the threshold otherwise the values are *below* the threshold. The weak classifier outputs one of two values depending on which side of the threshold a feature value falls.

As mentioned earlier, Discrete AdaBoost uses weak classifiers, $h(\cdot)$, that produce binary outputs while the weak classifiers used for Gentle AdaBoost, $\hat{h}(\cdot)$ produce real-valued outputs. The decision stump structure is used for both of these cases.

The discrete version of the decision stump has an additional binary *direction* parame-

Figure 2.8: Decision Stump Discrete and Continuous Outputs

ter. If the classifier is directed *up*, any images with feature values above the threshold are classified as positive. Otherwise, the classifier is directed *down* and feature values below the threshold are classified as positive. Fig. 2.8 shows the output of a discrete decision stump, $h(\cdot)$, directed up.

The real-valued decision stump produces its output based on class weighted probabilities

$$\hat{h}(x_i) = P_w(y_i = +1|x_i) - P_w(y_i = -1|x_i). \quad (2.13)$$

In a decision stump, these probabilities are produced based on where the training examples lie relative to the threshold. Positive and negative weights are distributed on either side of the threshold value in the feature value space. The sum of the positive and negative weights above the threshold will be denoted as P_A and N_A respectively. Likewise, the sum of the positive and negative weights below the threshold are P_B and N_B . The weighted probability that the label is 1 is the ratio of positive weights to total weights on a side of a threshold. The probabilities are computed as:

$$P_w(y = +1|x) = \begin{cases} \frac{P_A}{P_A + N_A}, & \text{if } f(x) \geq \theta \\ \frac{P_B}{P_B + N_B}, & \text{otherwise} \end{cases} \quad (2.14)$$

$$P_w(y = -1|x) = \begin{cases} \frac{N_A}{P_A + N_A}, & \text{if } f(x) \geq \theta \\ \frac{N_B}{P_B + N_B}, & \text{otherwise.} \end{cases} \quad (2.15)$$

Here, $f(x)$ is the feature value for image x . Fig. 2.8 shows the output for a real-valued decision stump given a particular distribution of training examples. Each $+$ represents one positive training example at a point in the feature space while $-$ represents one negative training example, all weights are equal. For feature values below the threshold, $P_w(y = +1|x) = 1/5$ and $P_w(y = -1|x) = 4/5$, thus the output is $1/5 - 4/5 = -3/5$.

Training the decision stump involves minimizing the misclassification or squared error,

depending on the variant of AdaBoost, by adjusting the threshold and direction. To select these parameter values, the training examples are first sorted by their feature values. Examples with the same feature values and label are combined with their weights accumulated. By iterating through this sorted array and calculating the appropriate error (Eqn. 2.4 or Eqn. 2.8) at each point and direction, a threshold dividing the training data can be found that minimizes the error.

Using the notation above, the minimum weighted misclassification error at each point is calculated using

$$\min(P_A + N_B, P_B + N_A). \quad (2.16)$$

Here, $P_B + N_A$ is the misclassification error directed up while $P_A + N_B$ is the misclassification error directed down. Similarly, the weighted squared error is calculated as

$$\left(\frac{P_A - N_A}{P_A + N_A} - 1\right)^2 P_A + \left(\frac{P_A - N_A}{P_A + N_A}\right)^2 N_A + \left(\frac{P_B - N_B}{P_B + N_B} - 1\right)^2 P_B + \left(\frac{P_B - N_B}{P_B + N_B}\right)^2 N_B. \quad (2.17)$$

Since these error calculations involve only the sums P_A , P_B , N_A and N_B , these errors are easily calculated by accumulating the sums while iterating through the sorted training examples. Although the expression for squared error seems much more complicated, if the positive and negative examples are well split by the threshold, the probabilities approach 1 and 0 and the squared error resembles the misclassification error.

To summarize, the computations involved in training each weak classifier are evaluating the feature values for the examples in the training set, sorting the examples using the feature values and finally iterating through the examples to select a threshold and direction if required. This is done for each of the possible weak classifiers defined by the unique features in the image window. Furthermore, as shown in Sections 2.4.1 and 2.4.2, all weak classifiers are trained each time the weights are updated during AdaBoost. An outline of the weak classifier training procedure is shown in Algorithm 3.

2.5 Cascade Classifier

Once a committee is trained, the full cascade is evaluated on the training set. If the cascade hit and false positive rate targets are not met, another committee is trained. However, the set of negative training and validation examples is reduced to only include those that produced false positives. Thus, each new committee is focused on identifying

Algorithm 3 Weak Classifier Training

```

1: Given  $N$  weighted training examples  $(x_1, y_1, w_1), \dots, (x_N, y_N, w_N)$ 
2: for  $i = \{1 \dots N\}$  do
3:    $f_i = f(x_i)$ 
4: end for
5: SORT( $\{w, y\}$  by  $f$ )
6: for  $i = \{1 \dots N\}$  do
7:   AccumulateWeights( $P_A, P_B, N_A, N_B, w_i, y_i$ )
8:   error = ErrorFunction( $P_A, P_B, N_A, N_B$ )
9:   if error < minError then
10:    UpdateMinimum(error,  $f_i$ )
11:   end if
12: end for

```

only training examples that reach it resulting in more complicated committees deeper into the cascade. During detection, the cascade is evaluated in the order in which it was generated such that the more complicated committees are only tested if all previous ones have given a positive result. Training the cascade is stopped when the overall hit and false positive targets are met.

2.6 Computational Complexity

To accelerate the training algorithm, it is important to consider which procedures are good candidates for hardware parallelization. Fig. 2.9 shows the flow of the computation involved in training the cascade classifier in terms of high-level stages of computation. There are two characteristics of the algorithm to see from this diagram. First, the algorithm is largely sequential in that the ensemble is slowly built up with each new weak classifier trained depending on the performance of all previously added weak classifiers. Secondly, the processes involved in committee training make up the inner loop of the algorithm. The operations of testing the cascade and adjusting the training set are relatively infrequent. In addition these operations are simple compared to the committee training procedure.

Table 2.1 shows the time complexity of the committee training algorithm. Here, N is the size of the training set, V^p is the number of positive validation examples, V^n is the number of negative validation examples and W is the number of weak classifiers to train. From Algorithms 1 and 2, the parameter update routines involve $O(N)$ operations to

Figure 2.9: Training Algorithm Flow

Computation	Time Complexity
Weak Classifier Training	$O(WN \log N)$
Parameter Update	$O(N)$
Committee Testing	$O(V^p \log V^p + V^n)$

Table 2.1: Time Complexity of Committee Training

update the weights of the training set. Committee Testing involves sorting the positive examples of the validation set as well as iterating through the negative validation set and thus requires $O(V^p \log V^p + V^n)$ operations in general. However, as shown in Algorithm 3, training each weak classifier involves two iterations over the training set as well as sorting the training set. Thus, this step requires $O(WN \log N)$ time. As discussed earlier, there are typically several hundred thousand features in an image window, thus as the training set size grows, weak classifier training quickly dominates the overall computation time. However, weak classifiers are trained independently and thus may be trained in parallel. The inherent parallelism in weak classifier training as well as its large time complexity make it an ideal target for hardware acceleration.

2.7 Related Work

Although there has not been work in hardware architecture accelerating the training process, several algorithmic and software techniques have been developed to reduce the run-time of the training phase. Threading through Intel’s Thread Building Blocks (TBB) has been used in the popular OpenCV project [14] to provide some parallelism during weak classifier training. In addition, algorithmic techniques have also been developed. For instance McCane et al. [15] describe a method of restricting the number of weak classifiers compared at each iteration by using a heuristic search. This approximation necessarily has a negative effect on the final classifier quality due to the restricted search space. Wu et al. [16] propose a method that avoids re-training weak classifiers by using a large look-up table. However, pre-computation time can still be significant especially if additional features are introduced to increase detection performance. A method of accelerating training of weak classifiers should allow further improvement using these algorithmic approaches.

Chapter 3

Design Goals, Scope and Data Structures

In Chapter 2, the structure of the training algorithm was described. It was shown that training weak classifiers in parallel presents a good opportunity for accelerating the training process. However, this kernel is non-trivial to map efficiently to FPGA fabric. In this chapter, the design philosophy and approach used to implement object detector training in hardware will be explained.

3.1 Design Goals

It is important to understand the goals of this architecture that motivated some of the decisions made during the design process.

High Performance

A primary motivator for this project is the extremely long computation time required to train Viola-Jones style object detectors. Therefore, the proposed architecture should be much faster than software when performing this task to provide worthwhile acceleration. The architecture should take advantage of the parallelism available during training to achieve speed-up over general purpose processors.

Portability

The proposed design is targeting two different application domains. First, the design should be able to integrate into High Performance Computing (HPC) environments where it may be used to facilitate algorithm experimentation and development. Second, embedded platforms should be able to use the architecture to accelerate training of object detectors with data collected in the field. The platforms and FPGA devices used in these domains differ significantly and the proposed architecture should be able to be implemented on both with minimal design effort. Furthermore, the capabilities of FPGAs are rapidly evolving and the architecture should be portable to ensure its continued applicability.

Scalability

The design should scale gracefully with the availability of additional FPGA resources. That is, performance should increase proportional to the available resources as the design is expanded. This is related to portability as the architecture is expected to function well on a variety of FPGA devices.

Flexibility

One of the main applications of this architecture is to facilitate the exploration of the Viola Jones ensemble technique. Thus, the design should be parameterizable to allow examination of parameters such as number and types of features without hardware re-synthesis. Furthermore, to allow for more intricate changes to be made, the design should be modular and easily modified.

3.2 FPGA Characteristics

The structure of FPGAs make them very flexible substrates. However, they have unique characteristics that affect the types of problems they are applicable to and how those algorithms should be implemented. In this section, some of the characteristics of FPGAs relevant to accelerating Viola-Jones object detection training will be discussed.

Parallelism

FPGAs offer an abundance of fine-grained parallelism through distributed, configurable Look-Up Tables (LUTs). This may be exploited in several ways to accelerate object detector training. Operations such as feature evaluation may be laid out spatially to provide parallelism through a pipelined structure. Also, parallelism is inherent in the independent training of weak classifiers. This allows hardware to be re-used and replicated across the FPGA for acceleration.

Flexible Wiring

FPGAs have very flexible and abundant wiring resources. This accommodates very high-bandwidth, on-chip connections as well as the construction of specifically tuned datapaths that can be reconfigured depending on desired word widths.

Memory Structures

Two general types of memory are typically available in the FPGA fabric. Block RAMs are configurable, dense memory structures, but are limited in number. Some LUTs may also be used as Distributed RAMs, providing very flexible storage but with less density. These memories are distributed throughout the FPGA and can provide very high aggregate bandwidth. However, object detector training operates on large data sets and the on-chip memories must be carefully managed to obtain the most utility from them.

Arithmetic Blocks

The general substrate of the FPGA is composed of LUTs, but modern FPGAs are often equipped with more complex digital signal processing (DSP) slices as well. Arithmetic operations such as fixed-point addition, subtraction and multiplication map well to these blocks. However, some operations such as division are still expensive in terms of hardware resources and latency. Floating-point operations also carry some overhead and should be avoided if possible.

3.3 Design Scope

During hardware design, a trade-off often appears between programmability and performance. Although a design could be made to support a very large variety of functions, it would likely suffer from resource and performance overheads. To ensure that the proposed architecture met performance and scalability goals, the problem space was constrained to support a subset of possible features and functions.

3.3.1 Design Partitioning

As discussed in Chapter 2, the kernel of weak classifier training requires the overwhelming amount of computation. Thus, it is conceivable to design a hardware system only focused on training and comparing weak classifiers and allowing a supporting processor to handle the rest of the operations. However, this approach incurs some overhead in terms of coordination and memory transfers when updating committee parameters. In particular, weights must be transferred between the hardware core and processor, with this overhead increasing with the size of the training set. Since committee parameter updates and testing occur after each weak classifier is selected, this step must be performed relatively often. Weight updates can also be handled quickly and inexpensively in hardware through a pipelined update stage. For these reasons, full committee training is carried out in the proposed architecture.

The steps of testing the cascade and updating the training set, on the other hand, occur at less frequent intervals, after each committee has been trained. In the early stages of the cascade, a committee may be composed of only a few weak classifiers, but towards the end of training, hundreds of weak classifiers may be selected for a committee. The cascade-level procedures, involve testing the full cascade and selecting the negative training and validation examples for the next committee. Although testing the cascade is very amenable in hardware, as evidenced by the previous FPGA detector implementations, the architecture for high performance detection is significantly different from training. For detection, many feature evaluation units are typically replicated to quickly determine the output of all weak classifiers. Since training weak classifiers is more complex, much fewer feature evaluators are available in the training architecture. To simplify the design, the proposed architecture does not perform the cascade testing and training set initialization tasks.

This design choice results in some overhead, particularly when transferring training

images and associated weights after cascade testing. However, this transfer occurs infrequently relative to the computation time of training. Furthermore, as training progresses, the negative training and validation sets grow smaller, resulting in less data transfer. In an embedded system where external memory bandwidth is shared between the processor and FPGA fabric, such as the Xilinx Zynq platform [17], the overhead in these data transfers would be minimal. In such a platform, committee training tasks such as estimating parameter updates could be offloaded to the processor as well, depending on its capabilities.

3.3.2 Supported Extensions

The basic design goal for the architecture is support for all standard operations involved in training a Viola-Jones detector committee. This includes support for the standard rectangular features in Fig. 2.5 as well as Discrete AdaBoost for creating committees. However, to demonstrate the flexibility of the design, several extensions of the algorithm are also supported.

Features are an integral part of the Viola-Jones object detector. Experimentation with different features can be very fruitful in terms of improving detection performance. For example, the rotated features used by Lienhart et al. [11] improved performance of a face detector. In addition to support for the standard rectangular features, the architecture supports features based on the rotated integral image. Further, more general linear combinations of up to nine integral image pixels may be used to create a variety of other features. More detail is provided in Section 4.5.1.

There are several variants of AdaBoost, and even more methods of forming ensembles. Gentle AdaBoost can be used to create ensembles with improved detection performance relative to Discrete AdaBoost in some situations. However, the computations involved in minimizing squared error are complex and expensive. Thus, a variant of Gentle AdaBoost is implemented where the thresholds are found using misclassification error while the updates are performed using the normal Gentle AdaBoost equations. This avoids the requirement of calculating probabilities until a weak classifier is selected. This is also the approach used by Demirkir et al. [18]. The Gentle AdaBoost extension is implemented as a modification of the hardware architecture and requires FPGA reconfiguration, while the extended set of features is enabled through parameters sent to the hardware engine.

3.3.3 Data Representation

One of the important considerations for hardware design is the data width and representation. Typical applications of the Viola-Jones object detection algorithm use 8-bit greyscale images and are trained at window sizes around 20x20 pixels. It has been found, for instance, that a 20x20 window size achieves a higher hit rate than larger windows in a face detection experiment [11]. To facilitate more general exploration of image window sizes, a maximum window of 32x32 was chosen. This choice requires the integral image representation to be 18 bits per pixel to support the maximum integral image value. Furthermore, since the feature values are signed, the value range for the three-element feature in Fig. 2.5 requires 19 bits to be represented. By reconfiguring the datapath, even larger window sizes may be implemented while smaller windows can be tested without any hardware re-synthesis.

Image and classifier weights (w, α) are typically represented as floating-point numbers. Since floating-point operations are expensive in hardware, these values are represented as 32-bit fixed-point numbers. Weights are normalized values and take on values between 0 and 1. Thus, they are represented with 32 fractional bits affording a minimum precision of 2^{-32} . Relative to floating-point, the fixed-point representation has reduced precision towards smaller values in the range. It is possible for training examples to have very small weight if they are consistently correctly classified. However, as weights approach values near the minimum allowed precision, they represent very small fractions of the total weight of 1 and thus do not contribute significantly to the training result.

The range of α values depend on whether Discrete or Gentle AdaBoost is being used. In Discrete AdaBoost, α is defined as $\ln((1 - \epsilon_t)/\epsilon_t)$. Since the error, ϵ_t , is determined by the sum of the misclassified weights, the range of α is governed by the minimum precision of the weights. With 32-bit weights, the maximum value is approximately 22.2 and the minimum value 9.3×10^{-10} . This range cannot be represented at full precision with a 32-bit fixed-point number, particularly when α values are accumulated to test the validation set. However, once again precision can be reduced for small values of α since at that point the selected weak classifier very poorly classifies the training set and does not contribute much to the committee. Further, if α becomes extremely small, a more fundamental problem may exist with the training set or features being used.

Finally, the α values for Gentle AdaBoost are more well-behaved taking values in the range $[-1, +1]$, with precision again governed by the weight representation. Since

Variable	Memory Requirements
Integral Image Data	$p_w * p_n * N$
Training Weights	$w_w * N$
Training Labels	N
Feature Values	$f_w * N$

Table 3.1: Memory Requirements for Weak Classifier Training

one bit is used for the sign, the minimum precision is 2^{-31} . Once more, some precision must be given up when accumulating these values, but values near zero represent poor classifiers that do not significantly contribute to the committee decision. Thus, this loss in precision relative to floating point values at such ranges can be justified.

3.4 Weak Classifier Training

As discussed in Chapter 2, a primary opportunity for acceleration of the training algorithm is through parallelization of training weak classifiers. However, the amount of data that must be accessed and stored for each weak classifier and the complexity of the computations complicates the implementation of this kernel in hardware.

Table 3.1 lists the variables used during weak classifier training and their memory requirements. Here, p_w is the integral image pixel bit width, p_n is the number of pixels required for the feature being trained, f_w is the bit width of feature values and w_w is the bit width of the weight representation. With the data bit widths chosen in Section 3.3.3, $p_w = 18$, $f_w = 19$ and $w_w = 32$. All of these values scale with the, typically large, number of training examples, N . Integral Image Data is used to generate the feature values and can be shared among multiple training engines. However, the feature values and resulting sorted order of weights and labels are unique to each weak classifier. Thus, training multiple weak classifiers in parallel requires high capacity and high bandwidth memory interfaces.

A second issue is the computational complexity of sorting the large set of training data for each weak classifier. Instead of explicitly sorting the data, the feature values are used to index large arrays in which training examples weights are inserted. Separate tables are kept for positive and negative weights to facilitate the error calculation. Using this technique, training examples are inserted in sorted order as the feature values are calculated. However, the size of the tables is the full range of the feature values rather

Figure 3.1: Filtering for Sorted Subsets

than just the number of training examples. With tables of 2^{19} , 32-bit weights, this technique exacerbates the memory storage issues.

The limited memory resources on an FPGA are quickly exhausted if storing the sorted tables on-chip. However, storing the data in external memory would result in a linear increase in required memory bandwidth with the number of engines. Instead of creating the full sorted tables at once, the sorting technique described above can be used to create *sorted subsets* of the full feature value range by calculating feature values for the full training set and filtering out training examples not contained in the desired range. For example in Fig. 3.1, the top nine bits of the 19-bit feature value are compared against a filter. If the upper bits match the desired filter, the bottom ten bits are used to insert the training example weights into the memory. On the other hand, if the upper bits do not match the subset, the weights are accumulated in registers keeping track of the number of weights above and below the subset. This technique dramatically reduces the required memory capacity and allows FPGA Block RAMs to be used for storage. In particular, 32-bit, 1024-element Block RAMs are used to store the positive and negative weights for each engine. At most two ports are required at a time to accumulate weights in the tables and read the values.

The above technique reduces the memory requirements, but increases the time to iterate through the feature value range since tables must be filled multiple times. In the example above with 1024-element Block RAMs, 512 sorted subsets would need to be explored for a 19-bit feature value. In general, the range of values for a particular feature will not be very large. Therefore, searching through all possible feature values is inefficient. Instead, the number of explored sorted subsets can be reduced by performing a two-phase threshold selection.

The two-phase algorithm first involves a low-precision pass where small tables of size M_L are used to locate areas in the feature space that are good candidates for thresholds. The second phase, called the high-precision pass uses second tables of size M_H to search the candidate regions and select a final threshold. In practice, single set of memories of

Full Feature Range				Low Precision Bins					
$f(x)$	w^p	w^n	ϵ^u	$f(x)$	w^p	w^n	ϵ^u	$\Delta\epsilon^u$	$\epsilon^u - \Delta\epsilon^u$
111	0.1	0.1	0.5	11x	0.1	0.1	0.5	0.0	0.5
110	0.0	0.0	0.5				0.5	0.1	0.4
101	0.2	0.0	0.5	10x	0.2	0.0	0.3	0.0	0.3
100	0.0	0.0	0.3				0.3	0.1	0.2
011	0.1	0.0	0.2	01x	0.1	0.1	0.3	0.1	0.2
010	0.0	0.1	0.3				0.5	0.3	0.2
001	0.1	0.1	0.3	00x	0.1	0.3	0.5	0.3	0.2
000	0.0	0.2	0.5						

Figure 3.2: Creation of Low-Precision Bins

size $\max(M_L, M_H)$ is used to store both tables. For example, with 19-bit features and 1024-element Block RAMs, M_L is 1024 and since the remaining precision is 9 bits, M_H is 512. Algorithm 4 shows the steps of this procedure.

The low-precision pass begins by loading examples from the full feature value range into a small set of sorted tables where table bins are coarsened to combine groups of training example weights (lines 2-5 of Algorithm 4). In contrast to the sorted subset filtering technique, the upper bits are now used to index tables instead of being used as filters and the lower bits of the feature value are ignored. The weights of training examples with the same upper bits and label are accumulated into a single bin. Fig. 3.2 shows the construction of the low-precision tables. Here, w_p represents the accumulated positive weights at a particular table bin, while w^n indicates the accumulated negative weights. In this example, the low-precision tables can only hold four elements at once ($M_L = 4$), thus the upper two bits of the three-bit feature values are used to aggregate weights into low-precision bins. For instance, the bin 00x contains 0.3 negative weights accumulated from the 0.2 weights in bin 000 and the 0.1 weights in bin 001.

Once the low-precision tables has been filled, errors are first calculated at the coarse thresholds between the bins (lines 6-12 of Algorithm 4). This is useful for roughly updating the minimum error, particularly when the training has just begun. In Fig. 3.2, error values for a classifier directed up (ϵ^u) are shown. These are calculated as in Eqn. 2.16 by summing the negative weights above (N_A) and the positive weights below (P_B) each potential threshold. An example is shown for the threshold at 110, between the low-precision bins 11x and 10x. Here, the negative weights above the threshold are circled

00x High Precision Expansion				01x High Precision Expansion			
$f(x)$	w^p	w^n	ϵ^u	$f(x)$	w^p	w^n	ϵ^u
> 001	0.4	0.2	0.3	> 011	0.3	0.1	0.3
001	0.1	0.1		011	0.1	0.0	
000	0.0	0.2		010	0.0	0.1	
< 000	0.0	0.0	0.5	< 010	0.1	0.3	0.3

Figure 3.3: High Precision Phase

in grey, while the positive weights below are circled in black. In this case $N_A = 0.1$ and $P_B = 0.4$, thus the error is 0.5 directed up.

During a second pass through the table, an uncertainty ($\Delta\epsilon$) is calculated for each bin, representing the amount that the error could decrease by searching through the range in higher precision. The $\Delta\epsilon$ is simply the negative weights in the bin above a threshold for a classifier directed up or the negative weights in the bin below when directed down. To understand this, consider the low-precision bin 00x in Fig. 3.2. There are 0.1 positive and 0.3 negative weights accumulated within this bin. If all of the positive examples are at higher feature values than the negative examples, then a threshold can be found, splitting these examples, such that the 0.3 negative weights are no longer misclassified when directed up, thus the $\Delta\epsilon^u$ is 0.3.

Once the errors and $\Delta\epsilon$ have been calculated, the *potential errors* ($\epsilon - \Delta\epsilon$) are found. This calculation is combined with the $\Delta\epsilon$ calculation in lines 14-20 Algorithm 4. If any of these potential errors are lower than the current minimum error, then that subset becomes a *candidate* for a high-precision expansion.

The high-precision phase uses the sorted subsets technique described earlier to fill the high-precision table with elements belonging to a candidate subset (lines 23-30 of Algorithm 4). The bins of the subset are now searched to determine if the minimum error can be improved (lines 31-37 of Algorithm 4). In Fig. 3.2, both ranges 00x and 01x have low potential errors and are thus candidates for the high-precision phase. The expanded ranges are shown in Fig. 3.3. Extra values in grey, are stored to keep track of the weights above and below the subset. In range, 00x, the potential error is 0.2, but the negative weight is not fully below the positive weight, thus the minimum error in this subset is 0.3. On the other hand, the weights are perfectly separated in the range 01x, thus the minimum error is 0.2.

The compression and expansion of the feature space in the low and high-precision

Figure 3.4: Summary of the Two-Phase Search

phases are shown graphically in Fig. 3.4. Magnitudes of the weights are shown as horizontal bars in the tables. Here, only the candidate at low-precision bin 01x is shown and expanded in high-precision. From this diagram, it can be seen that the two-phase approach locates a threshold in the full feature space without ever having to store the full set of values at once. In particular, the low and high-precision tables require only four and two elements respectively.

In comparison with the original algorithm, in Algorithm 3, the worst case complexity is greater. During the low-precision phase, $O(\max(N, M_L))$ operations are required and $O(M_L \cdot \max(N, M_H))$ are required in the worst case during high-precision search. However, the worst case would only occur if all low-precision bins were candidates for high-precision search. In most cases, only a few candidates are generated.

The speed of this algorithm relative to the original depends largely on the training set size N and the distribution of training set examples. If N is much smaller than M_L , the full table range of M_L will be searched even though only a few bins can ever be occupied. Thus, this algorithm will perform poorly on small datasets. However, if N is large and data is more normally distributed, this algorithm will scale closer to $O(N)$ and perform better than the $O(N \log N)$ original algorithm.

3.4.1 Committee Testing

The two-phase technique, presented above, is a general method of searching through a set of data. It may also be used to determine a committee threshold value ϕ instead of explicitly sorting the positive validation set. In this case, the procedure is simpler. The value of ϕ should be set such that it is below the aggregate committee output values

(H) of some percentage of positive validation examples. This is done by using the H values to index the sorted tables instead of $f(x)$ and using $1/V^p$ as the weight for each example. By performing a low-precision pass through the H value range from low to high, a subset can be found after which the hit rate is not met. This range is then selected for a high-precision expansion and the final committee threshold value is determined using the same search.

Algorithm 4 Two-Phase Weak Classifier Training

```

1: clear table,  $P_A, P_B, N_A, N_B$ 
2: for  $i = \{1 \dots N\}$  do
3:    $f_i = f(x_i)$ 
4:   table[UpperBits( $f_i$ )] [ $y_i$ ] +=  $w_i$ 
5: end for
6: for  $i = \{1 \dots M_L\}$  do
7:   AccumulateWeights( $P_A, P_B, N_A, N_B$ , table [ $i$ ])
8:   error = ErrorFunction( $P_A, P_B, N_A, N_B$ )
9:   if error < minError then
10:     UpdateMinimum(error,  $i$ )
11:   end if
12: end for
13: clear  $P_A, P_B, N_A, N_B$ 
14: for  $i = \{1 \dots M_L\}$  do
15:   AccumulateWeights( $P_A, P_B, N_A, N_B$ , table [ $i$ ])
16:   perror = PotentialError( $P_A, P_B, N_A, N_B$ , table [ $i$ ], table [ $i - 1$ ])
17:   if perror < minError then
18:     push  $i$  onto candidates
19:   end if
20: end for
21: while candidates not empty do
22:   clear table,  $P_A, P_B, N_A, N_B$ 
23:   for  $i = \{1 \dots N\}$  do
24:      $f_i = f(x_i)$ 
25:     if Filter(candidate) = UpperBits( $f_i$ ) then
26:       table[LowerBits( $f_i$ )] [ $y_i$ ] +=  $w_i$ 
27:     else
28:       AccumulateWeights( $P_A, P_B, N_A, N_B, w_i, y_i, f_i$ , candidate)
29:     end if
30:   end for
31:   for  $i = \{1 \dots M_H\}$  do
32:     AccumulateWeights( $P_A, P_B, N_A, N_B$ , table [ $i$ ])
33:     error = ErrorFunction(table [ $i$ ])
34:     if error < minError then
35:       UpdateMinimum(error,  $i$ )
36:     end if
37:   end for
38:   pop candidate
39: end while

```

Chapter 4

Training Architecture

4.1 Platform Architecture

The hardware system is coupled with a coordinating general-purpose processor. The processor is responsible for transferring training and validation data to the FPGA, starting training and determining whether additional committees should be added to the cascade. Thus, it is important to have a high-bandwidth and flexible connection between the hardware accelerator and processor. The proposed architecture is built using a Message Passing Interface (MPI) platform called ArchES-MPI [19]. ArchES-MPI creates an abstraction of hardware and software components and provides a standardized method of communicating between these components agnostic to their implementation. By using a standardized interface, a variety of processors including MicroBlazes, PowerPCs and x86 CPUs may be seamlessly connected to the hardware engine without extra design effort. This is particularly important if adapting the design from an HPC environment to an embedded device where different processors are available.

ArchES-MPI also provides a high bandwidth and low overhead on-chip network to ensure the data is quickly transferred. In the tested platform, a Peripheral Component Interconnect Express (PCIe) bridge is used to connect an external x86 processor to the on-chip MPI network. Using this link, large amounts of training data are quickly transferred to the external memory connected to the FPGA.

Fig. 4.1 outlines the high level structure of the Object Detector Training Core. The core consists of three main parts: an MPI Controller, Memory Controller and the Compute Core. In the adapted committee training algorithm described in Section 3.4, feature values are evaluated many times to repeatedly fill small sorted tables. The memory ac-

Figure 4.1: High-Level Block Diagram of Hardware Accelerator Interfaces

cesses required to fetch the integral image pixels for feature evaluation puts high demands on the external memory bandwidth. To ensure maximum bandwidth to the compute core, the memory interface is directly integrated into the object detector engine.

Communication with the memory controller is facilitated by command, read and write First In First Out (FIFO) channels. The command FIFO acts as an instruction stream for the controller; commands encode whether to perform read or write operations as well as the address and size of access to perform. These commands are decoded and requests are made to a memory interface implemented using the Xilinx Memory Interface Generator (MIG).

The MPI Controller handles the interface to the MPI network, arbitrating access to the external memory between MPI requests and the Compute Core. While the Compute Core is idle, data may be read from or written to external memory through the MPI interface. However, once the Compute Core has started processing, that core has sole access to external memory. To start the Compute Core, parameters are first sent through the MPI network to the Object Detector Training Core. These parameters detail the number and location of training and validation data in memory as well as other training parameters such as hit rate and false positive rate. Once these parameters are loaded, the Compute Core begins training a committee. As each weak classifier is selected, its trained parameters are sent back through the MPI network. The full command format is described in Appendix A.

4.2 Compute Core Architecture

The Compute Core is designed to provide high performance given the modified training algorithm. Key to accelerating training effectively is ensuring that many weak classifiers can be trained in parallel. The process of training weak classifiers is independent and thus several training engines could operate with distributed control until all weak classifiers are trained and compared. This would keep the engines busy and avoid the overheads involved in centralized control. However, there are benefits to using running the engines in lockstep.

During feature evaluation, training engines require access to external memory for training data. However, some of the memory requests may overlap. Thus, by ensuring that all parallel engines perform feature evaluation simultaneously, external memory bandwidth can be used more efficiently. Furthermore, if each training engine calculates candidates and error simultaneously, the results can be easily consolidated, reducing the amount of memory required to keep track of these values. For these reasons, the architecture consists of a set of parallel engines connected together with centralized control.

An important concern when using centralized control is how to distribute data and commands to all parallel engines. In particular, feature evaluation requires high memory bandwidth to operate efficiently. To distribute data to all engines, a systolic architecture was chosen. A block diagram of the architecture is shown in Fig. 4.2. The primary datapath consists of a systolic array of engines, each containing a Feature Evaluator, set of Sorted Tables and a Threshold Selector. The systolic array allows pixel data from the memory interface to be distributed to all training engines in a pipelined fashion. Each cycle, a new memory vector is available at each engine. To configure the array, a shift register-like method is used where a configuration string is pushed through the engines. The systolic structure is also used to consolidate results among the Threshold Selectors working in parallel. The lowest error and selected candidates for all parallel engines are collected in the last Threshold Selector in the chain and returned to the Control Unit.

The wide, pipelined datapaths in the systolic structure map well to the abundant wiring and register resources available on the FPGA substrate. Since connections only appear between adjacent engines, the structure is very scalable where, provided there are enough resources, engines can be added without loss in clock frequency. The system topology forms a ring structure with the control unit feeding parameters to the first engine and receiving consolidated results from the last engine in the chain.

Figure 4.2: Block Diagram of Compute Core Structure

In each element of the array, Feature Evaluators receive the training data, calculate the feature transformation ($f(x)$) and pass the data on to be sorted. Sorted Tables perform the filtering required for the low or high-precision table searches. Finally the Threshold Selectors calculate the error or high-precision candidates depending on the phase of the computation. Valid signals are used to control flow between engines, and the design is intended to be very modular, where for instance, modifications to the Feature Evaluator can be made without changing the rest of the training structure.

In general, Sorted Tables must be filled before threshold selection can occur. However, this would cause half of the compute core to be idle at any one time. To keep the engines busy, the memories in the Sorted Tables are replicated, creating a double buffer. Thus, while the Feature Evaluators fill one table, the Threshold Selectors calculate error on the previously filled table.

Although the design is intended to maximize the performance of weak classifier training, the steps of parameter update and committee testing are also easily mapped to this architecture. Once the minimum error is found by the training engines, the weight updates and α values are calculated by the Update Calculator unit. To update the training set weights, a single engine in the chain is configured as the minimum error weak classifier. Training data is streamed in, as in weak classifier training, but in this case the Update Evaluator applies the weight update and the updated value is written back to memory. The accumulation of the aggregate committee output values (H) is handled in

the same manner, by streaming in validation examples and accumulating the aggregate weak classifier output.

As discussed in Section 3.4.1, the committee threshold (ϕ) is also determined through the two-phase search. Again, one engine is configured as the minimum error weak classifier and the Threshold Selector calculates the hit rate (HR) at various thresholds until an appropriate ϕ is found. The False Positive Rate (FPR) is then calculated by testing the H values of the negative validation set against the threshold.

4.3 Memory Management

Object Detector training involves operating on a large set of data that is impractical to store in on-chip memory. Thus, making efficient use of the external memory bandwidth is very important to achieving high performance. In this work, the test platform is equipped with 8-byte wide DDR3 memory. At an I/O clock of 400 MHz, this interface provides a maximum bandwidth of 6.4 GB/s. The Xilinx MIG makes this data available as 32-byte vectors every cycle at 200 MHz. The DDR3 standard [20] lists two types of burst operations, burst length eight (BL8) and burst chop four (BC4). In BL8 operation, the eight words, or 64 bytes, are pre-fetched and produced over two cycles at the MIG user interface. On the other hand, during BC4 operation, eight words are still pre-fetched, but only the first four words are returned. When switching between read and write operations, BC4 can reduce the latency between requests. However, no time is saved when continuously reading or writing data. The vast majority of requests during object detector training are continuous read requests. Therefore, BL8 mode is used with eight words as the basic request size for the memory controller.

4.3.1 External Memory Layout

For each training example, the integral image pixels, weight and label must be fetched from external memory. Similarly for validation data, the pixel data, aggregate output H , and label are required. The arrangement of these variables in external memory can greatly impact the performance of the system.

As discussed in Section 3.3.3, the integral image pixels are 18 bits wide. These pixels are arranged such that 14 pixels are packed in each 32-byte vector for a total of 28 pixels per burst. This arrangement means four bits are unused in every 32-byte vector, but the

datapath is simplified. Positive and negative training and validation images are stored in four separate regions of the external memory space. By keeping these sets of image data separate, the labels become implicit to the data being accessed.

Weights or aggregate outputs are required for each training or validation image. To make use of the burst request size and reduce the number of memory requests, these values are fetched in blocks at a time. The values are cached in a FIFO and popped out as the pixels for each image are streamed through the Feature Evaluators.

4.3.2 Integral Image Requests

Each training engine requires six to nine random integral image pixels depending on the feature being evaluated. The required memory bandwidth thus necessarily increases with the number of engines. However, the burst organization of DDR3 RAM provides some opportunity for data reuse. Since unique features are created through scaling and translating rectangular shapes, in many cases multiple features use pixels located in the same bursts. This locality can be taken advantage of by rearranging the order in which independent weak classifiers are trained.

To order the features, they are first sorted by the number of bursts required to evaluate them. Next, sets of features are created, sized to the number of compute engines available. Each new set begins with the feature with the current minimum number of bursts. The list of features is then searched and additional features are added such that the total number of bursts in the set is minimal.

Fig. 4.3 shows the result of ordering the standard rectangular features on a 32x32 image window. The plot demonstrates that as the number of parallel engines grows, the number of bursts that can be shared diminishes. If only one engine is requesting data, 4.5 bursts are required on average. This is shown as the horizontal grey line and represents the best case limit. The “unordered” configurations are formed by grouping features in the order in which they are generated through translating and scaling each rectangular shape across the image window. Notice that the number of bursts required grows quickly with engines as they are not shared efficiently, resulting in long periods where individual engines are idle. When features are ordered on the other hand, the majority of requested data is shared, with 30 engines requiring 5.2 bursts per configuration on average.

The structure of rotated rectangular features presents a few problems. First, the rotated features require their own integral image data and thus cannot share bursts

Figure 4.3: Effect of Burst Ordering for Standard Features

Figure 4.4: Standard and Rotated Pixel Alignment Methods

with the standard features. Second, the standard layout of pixels in memory involves enumerating them by iterating along rows of the image. This representation does not facilitate the sharing of pixels for rotated features. Scaling a rotated rectangular region results in a shift of the corner pixels in both horizontal and vertical directions making it unlikely that a burst will cover both the original feature and the scaled version. To resolve this issue, a rotated pixel alignment is used where, starting from the top left corner of an image, pixels are enumerated along diagonals. Fig.4.4 shows the difference in memory structure. Here, a 3x3 image region is shown, where the numbers in each pixel represents its location in memory. The shaded regions represent three-pixel bursts.

Using this rotated pixel alignment, many more rotated features may share bursts. Fig. 4.5 shows the results for the rotated features in a 32x32 image window. With only one engine requesting data, the best case is four bursts per configuration. This increases up to five bursts in the ordered case and almost eleven bursts when unordered. The “Standard” curve shows the bursts required when features are ordered, but the standard,

Figure 4.5: Effect of Burst Ordering for Rotated Images

row-based enumeration of pixels is used. This representation results in many more bursts per configuration since there are fewer opportunities to share bursts.

4.4 Weak Classifier Training Operation

Since weak classifier training dominates the training computation, it is important to consider how the systolic architecture works to perform this task. In this section, the function of each component of the Compute Core during weak classifier training will be described. During this function, all engines are used except for the Update Calculator and Update Evaluator. The Update Evaluator is set to simply pass the training data $(f(x), y, w)$ to the Sorted Tables without modification.

For a set of weak classifiers, the feature configurations are first fetched from external memory. These configurations specify the pixels required and operations to perform on them. The configurations are shifted into the Feature Evaluator chain and the Sorted Tables are also configured to create low-precision bins. During this configuration step, the bursts required for the set of features are decoded and stored in a circular FIFO. Feature evaluation now begins where the positive and negative training examples are streamed from memory through the hardware engines.

Once the tables have been filled, the memories are swapped such that the Feature Evaluators fill a fresh table, while the Threshold Selectors are connected to the filled

table. Since data is now available for the Threshold Selectors, all array engines are configured to begin operation. In particular, the Feature Evaluators are configured for the next set of features, the Sorted Tables continue to filter for low-precision bins and the Threshold Selector calculates errors and candidate ranges. This process of training continues while no candidate ranges are found. When some feature and threshold are found to have lower error than the current global minimum error, the parameters for the minimum error classifier are updated.

If a high-precision candidate is found by the Threshold Selectors, the Feature Evaluators break from their task and the system switches to high-precision operation. The Feature Evaluators and the Sorted Tables are then configured to fill the candidate ranges. The same feature could be configured on multiple Feature Evaluators if several candidate subsets were found for that feature. Threshold Selectors work to calculate the error and determine if the global minimum error can be updated. Once all of the candidates have been tested, the system reverts back to low-precision operation.

The switch between low and high-precision modes, serializes the operations of the Feature Evaluator and Threshold Selector. When switching to high-precision, the Threshold Selector must wait while the Feature Evaluators fill the Sorted Tables with the appropriate subset weights. The same delay occurs when switching back to low-precision. Table 4.1 lists an example set of procedures carried out to train 40 weak classifiers using 10 engines. The boxes under each engine show the operations or set of data being processed, an empty box represents an idle engine. High-precision phases are highlighted in grey. Notice that the Feature Evaluators and Threshold Selectors are serialized in steps 3 and 5 and the progress through the features is halted during the high-precision pass.

Rotated features are tested once all of the standard features have completed. These phases are separated since bursts cannot be shared between standard integral and rotated integral images. The procedures are identical except for changes in memory addresses.

4.5 Hardware Microarchitecture

The computational units that compose the systolic array must be modular and provide high throughput. In this section, the capabilities and some of the methods used to implement these cores will be presented.

Step	Feature Evaluators	Sorted Tables	Threshold Selectors
1	Features 1-10	Low Precision	
2	Features 11-20	Low Precision	Features 1-10 <i>13 Candidates Found</i>
3	Features 1-10	Candidates 1-10	
4	Features 1-10	Candidates 11-13	Candidates 1-10
5			Candidates 11-13
6	Features 11-20	Low Precision	
7	Features 21-30	Low Precision	Features 11-20
8	Features 31-40	Low Precision	Features 21-30
6			Features 31-40

Table 4.1: Example Weak Classifier Training Operation

4.5.1 Feature Evaluator

The main purpose of the Feature Evaluator is the calculation of feature values using the equations shown in Fig. 2.7. These are linear combinations of six to nine integral image pixels. Instead of supporting a fixed set of shapes, the hardware engine is designed to calculate any linear combination of up to nine pixels using the following operators: $\{+, -, +2\times, -2\times, +3\times, -3\times, +4\times\}$. This gives the user the flexibility to freely experiment with combinations of pixels. Further, the integral image representation could be discarded altogether and features could be created as linear combinations other representations of input data.

Fig. 4.6 shows the datapath of the Feature Evaluator. As mentioned in Section 4.3, 32 bytes of integral image data are available from the memory controller each cycle for a total of 14 pixels per cycle. It's possible for all pixels required by a feature to be contained within a single burst. Therefore, 14 filters, $g(\cdot)$, are aligned to each pixel location. Each burst of image data is accompanied by an index denoting the position of the pixels in the image window. The filters then either ignore the pixel data or apply one of the aforementioned operators. The result is fed into a tree accumulator structure. In modern 6-LUT based Xilinx FPGAs, ternary adders require only extra routing resources compared to a binary adder. Thus, a ternary tree adder is used to reduce resource requirements and improve latency.

Feature evaluation is a key part of weak classifier training and is used multiple times to fill the small sorted tables. Thus, the engine is designed for high throughput. In particular, the structure is fully pipelined with a latency of eight cycles to accommodate

Figure 4.6: Feature Evaluator Engine

all operations as well as input and output registers. With the chosen integral image memory organization, pixels from multiple images may not share bursts. Thus, the maximum throughput is limited to one feature value per cycle.

As mentioned earlier, the feature configurations are loaded from external memory and shifted through the systolic array to reach each engine. The configuration word for each feature contains the burst number, pixel offset and operation to perform for nine pixels. The full details are shown in Appendix B.

4.5.2 Sorted Tables

The Sorted Tables are composed of four Block RAMs to store the buffered positive and negative weights as well as address generation logic to sort training examples. Table addresses are calculated using feature values based on whether low or high precision search is being carried out. The range of feature values searched is configurable such that smaller feature representations can be searched without reconfiguring the hardware. For instance, an 18-bit range can be searched by using the upper ten bits for a low-precision pass and the remaining eight bits during the high-precision search. As with the Feature Evaluator, the core is fully pipelined with a latency of four cycles to accumulate weights into the sorted tables. Additional details of table address generation are provided in Appendix C.

4.5.3 Threshold Selector

Once the Sorted Tables have been filled, the Threshold Selector is used to calculate the error or determine the candidacy of each table element. A counter is used to iterate

Figure 4.7: Synchronized systolic comparison. $\epsilon_{i,k}$ represents the error calculated for table address i for engine k ; $\min \epsilon_{i,k}$ is the minimum error at table address i amongst k engines.

through the tables and the sums P_A , P_B , N_A , and N_B are updated at each step. These sums are then used to compute the misclassification error (Eqn. 2.16) or the potential error. Finally, a comparison is made to the global minimum error to determine if a new minimum or candidate range has been found.

In addition to comparing local errors with the global minimum error, the errors and candidates amongst multiple Threshold Selectors must be consolidated for each table bin. However, a synchronous comparison is not desirable as that would require simultaneous reduction from all engines, resulting in long wire delays. Instead, comparison between adjacent engines is performed using the systolic array structure. Each Threshold Selector in the array begins operation delayed by one cycle. Thus, as one engine is calculating the error for table address i , the result from the previous engine has already been produced and can be used for comparison. This is demonstrated in Fig. 4.7. Using this comparison method, the last Threshold Selector in the array outputs the minimum error or consolidated candidates for all engines at each table address. Error and candidate calculation is again fully pipelined with a latency of eight cycles between requesting the data from the table and producing an error or candidate. Additional datapath details are provided in Appendix D.

4.5.4 Update Calculator

The Update Calculator is responsible for calculating the weight update and weights of the classifier outputs. These calculations involve the evaluation of several non-linear functions as well as multiplications and divisions. It is vital that they are calculated quickly and precisely for training to proceed as expected.

In Discrete AdaBoost, the weight update involves an error factor β_t and a normalization. These operations are combined to rewrite the weight update in terms of two multi-

plicative factors. Weights of correctly classified examples are multiplied by $1/(2(J - \epsilon_t))$ while incorrect examples are multiplied by $1/(2\epsilon_t)$. Here, J is the sum of all the weights which is approximately 1, but may deviate depending on inaccuracies in the weight update. This deviation does not accumulate since weights are normalized during each update. From these equations, the weight update term involves simple operations to calculate the denominators and a reciprocal function. A look-up table and three-stage piecewise linear interpolation (PLI) unit [21] is used to estimate the reciprocal. To estimate, $\alpha_t = \ln(J - \epsilon_t) - \ln(\epsilon_t)$, a similar approach is used where the look-up and table PLI are now used to estimate the natural logarithm. These approximation functions provide resource efficient methods of calculating parameter updates.

The update calculation phase represents the main difference between Discrete and Gentle AdaBoost. In Gentle AdaBoost, four weight update terms are required depending on the location of the training example relative to the threshold as well as the label of the example. Furthermore, a number of divisions are required to calculate the probabilities in Eqn. 2.14 and Eqn. 2.15 and to normalize the terms. Thus, a more traditional computation method is used, making use of the available DSP slices to calculate divisions and multiplications. Appendix E expands on the implementation details.

4.5.5 Update Evaluator

The Update Evaluator core has several functions. It is used to update weights, accumulate the aggregate committee outputs and calculate the false positive rate. When updating weights, the update factor to use is determined by comparing the feature value with the threshold (θ) as well as the label in the case of Gentle AdaBoost. A pipelined DSP multiplier slice is used to carry out the update.

Update of the aggregate committee outputs (H) is similar to weight updates. In this case, both Discrete and Gentle AdaBoost use the feature value and threshold to determine which update to apply. The aggregate outputs are then accumulated and returned to the Control Unit for write-back. Finally, the Update Evaluator also calculates the false positive rate by comparing the H values to the committee threshold (ϕ).

Each compute core in this architecture is designed to provide high throughput via heavy pipelining. In addition, each core is very modular with control flow managed via simple valid signals. Thus, changes to the functionality or latency of engines can be made without restructuring the rest of the architecture.

Chapter 5

Results and Analysis

5.1 Test Platforms

The proposed architecture was evaluated using a Xilinx ML605 development platform, containing a Virtex-6 LX240T FPGA. Xilinx ISE 13.1 tools were used to floorplan and implement the designs. The LX240T FPGA is equipped with 150,720 6-input LUTs, 301,440 Flip Flops (FF), 416 36-Kbit Block RAMs and 768 DSP slices. A Xilinx MIG DDR3 memory interface with 400 MHz I/O clock connected the memory controller to an on-board 512 MB DDR3 SODIMM. The compute engines ran at 200 MHz to match the interface clock of the MIG. To interact with the hardware system, a PCIe-MPI bridge was used, with the on-chip MPI network running at 125 MHz. Clock domain crossing was handled by an asynchronous FIFO interface.

To ensure correctness and measure performance of the hardware, the optimized, multi-threaded OpenCV [14] library was used for comparison. OpenCV is a popular computer vision framework that supports training of, and detection with Viola-Jones style object detectors. The *traincascade* procedure of OpenCV version 2.3.1 was used as a performance benchmark. This implementation contains several deviations from the original algorithm of Viola and Jones. First, variance normalization, discussed in Section 2.3, is performed as each feature is evaluated. However, normalization can instead be done once, as a pre-processing step during training, thus it was disabled for testing. Second, OpenCV, by default, uses three-element features with the center element weighted by a factor of two. The hardware architecture supports both weighted and unweighted versions of these features, but during testing, OpenCV was modified to use the unweighted style. Finally, the OpenCV version of Gentle AdaBoost was modified to use misclassification

error to train decision stumps, to match the hardware implementation.

In addition to these fundamental changes, OpenCV is packaged with several optimizations reduce training time. Influence trimming is a technique that reduces the number of training examples examined as the algorithm progresses at the cost of some accuracy. As weak classifiers are added to a committee, some correctly classified examples end up with very low weights and have little influence on selecting the next weak classifier. Thus, these examples are trimmed from the dataset to reduce the amount of computation required. A more direct method of improving performance is pre-calculating and sorting some of the feature values. Since the feature values do not change during training, a great deal of computation may be saved. However, the applicability of this technique is limited by the available system memory since data must be stored for many combinations of features and training examples. During testing, the default settings for both of these optimizations were used. Examples representing less than five percent of the total weight were trimmed and 512 MB of memory was reserved for pre-calculated data.

Software performance testing was done on a node of the General Purpose Cluster (GPC) supercomputer at the SciNet HPC Consortium [22]. A GPC node contains two 2.53 GHz Intel Xeon E5540 CPUs, each with four cores and hyperthreading support for a total of 16 logical processors. In this thesis, CPU will be defined as a single Xeon chip with multiple processor cores within that chip. The OpenCV library was compiled with Intel C Compiler version 12.1.3 and Intel Thread Building Blocks version 4.0. The software was instrumented to measure only the committee training time, using `clock_gettime()`, and results were averaged over twenty runs. To experiment with the number of threads, the `task_scheduler_init()` procedure was used. The linux `taskset` command was used to set processor affinity when limiting threads to particular processor. Hardware computation time was measured as the time between sending the start command and parameters over PCIe to receiving the last committee results.

5.2 Training Parameters

To ensure correctness and measure performance, a face detector was trained using a set of 1000 faces and 1000 non-faces as positive and negative examples. Faces were collected from the FERET [23] dataset, while non-face images were collected from patches of the Caltech background [24] dataset. The examples were cropped and resized to a 32x32 window and saved as equalized 8-bit greyscale images. This dataset is small for

training a good face detector, but it is sufficient to test the performance of the training implementations. OpenCV uses the training set for both training and validation, thus all tests were performed with the training and validation sets using the same images.

Both Discrete and Gentle AdaBoost were used to train committees using the full feature set. The combination of standard and rotated features results in a total of 642,992 features for a 32x32 window. Committees were trained with a desired hit rate of 0.99 and a false positive rate of 0.5. Although this false positive rate seems high, when committees are put together in a cascade, the combination will produce a much lower false positive rate since each committee will reject about half of the examples that reach it. The parameters resulted in committees with two or three weak classifiers on the collected training set.

All tests were performed where the hardware and software produced committees with the same weak classifiers. In general, the hardware system will produce the same results as software except for the case where multiple options for the minimum error weak classifier exist. This can occur if multiple weak classifiers produce the minimum error or multiple thresholds exist that have the same error. The algorithm does not differentiate between these options and thus they are equally valid solutions.

5.3 Performance Scaling

Fig. 5.1 shows the speed-up of the Discrete AdaBoost hardware platform over OpenCV for different numbers of compute engines. Results for one Xeon CPU using one and eight threads as well as two CPUs using 16 threads are plotted. Despite the burst ordering technique described in Section 4.3.2, not all bursts can be efficiently shared as the number of engines increases. Thus, performance scales slightly less than linearly.

The parameter update procedure is the largest difference between the Discrete and Gentle AdaBoost configurations. As discussed, this represents a relatively small amount of computation relative to weak classifier training. Fig. 5.2 shows the performance scaling trend of the Gentle AdaBoost hardware. As expected, the performance characteristics are very similar to those of Discrete AdaBoost.

Table 5.1 summarizes the hardware acceleration using a 30-engine configuration. The Intel Xeon E5540 and Xilinx Virtex-6 were launched in the same quarter and built on similar process technologies, 45 nm and 40 nm respectively. Thus, they represent good candidates for a platform comparison. In a single-chip comparison, the custom hardware

Figure 5.1: Speed-up of Discrete AdaBoost Hardware over OpenCV

AdaBoost Method	Threads	Speed-Up
Discrete	1	71.8
	8	15.0
	16	8.3
Gentle	1	72.8
	8	15.2
	16	8.4

Table 5.1: Hardware Speed-Up over OpenCV with 30 Engines

is able to achieve a 15-fold speed over the Intel CPU. Furthermore, the FPGA maintains an eight-fold speed-up against two CPUs.

Although it is difficult to measure power for the different platforms, particularly without access to the physical SciNet nodes, some estimates can be made. Each Intel Xeon processor is rated for 80 Watts Thermal Design Power (TDP), this is the power that must be dissipated by the heatsink although the actual power consumption may be over or under that amount at any particular moment. Using the Xilinx Power Analyzer, an estimate for total FPGA power of 11 Watts was obtained. Thus, the hardware platform is able to achieve considerable acceleration while being much more power efficient, making it amenable for both HPC and embedded applications.

The Discrete AdaBoost test produced a committee with only two weak classifiers, but it required about 340 seconds to train using a single-threaded OpenCV implementation. This was reduced to less than five seconds when using the FPGA platform. Thus, when

Figure 5.2: Speed-up of Gentle AdaBoost Hardware over OpenCV

thousands of weak classifiers are trained for a full detector, the reduction in training time provided by the FPGA system can be very dramatic. In addition, in terms of energy consumption, the FPGA platform consumes much less power over a shorter time period than the software implementation. This is an important factor for battery-powered systems.

An interesting secondary question that can be answered is how well Intel’s hyper-threading technology applies to the multi-threading technique used in OpenCV. In theory, running eight threads across the eight independent cores of two CPUs should provide better performance than eight threads running on four hyperthreaded cores on one CPU. It turns out that when running eight threads across two CPUs, performance of the OpenCV system improves by about five percent over eight threads on one CPU. Thus, the use of hyperthreading results in a relatively small decrease in performance in comparison with using actual cores.

Initial tests were also conducted to see the effect of the optimizations in OpenCV. With eight threads running on a single CPU, disabling optimizations resulted in training times about eight percent longer than with them enabled. The results with optimizations enabled have been used for comparison against the FPGA hardware.

Component	# Engines	FFs	LUTs	BRAMs	DSP
Compute Core	5	17068 (6%)	21912 (15%)	23 (6%)	4 (1%)
	10	28635 (9%)	35308 (23%)	43 (10%)	4 (1%)
	15	40184 (13%)	48546 (32%)	63 (15%)	4 (1%)
	20	51861 (17%)	61972 (41%)	83 (20%)	4 (1%)
	25	63496 (21%)	75283 (50%)	103 (25%)	4 (1%)
	30	75158 (25%)	88614 (59%)	123 (30%)	4 (1%)
Full System	30	94,136 (31%)	107,930 (71%)	214 (51%)	4 (1%)

Table 5.2: Hardware Discrete AdaBoost Resource Utilization

5.4 Hardware Utilization

Resource utilization is an important measure of system scalability. Preferably, performance should increase proportional to the resources consumed. Table 5.2 shows the post-mapping utilization of the Compute Core for the Discrete AdaBoost configuration. As training engines are replicated and added to the systolic array, utilization grows approximately linearly, with the bottleneck being the number of LUTs consumed to carry out the computations. Due to the systolic architecture, all configurations maintained a clock speed of 200 MHz and should continue to scale on larger devices.

The resource utilization trend for Gentle AdaBoost, shown in Table 5.3, is similar. Here, additional resources are required to store and compute the more complex update terms. In particular, 21 DSP slices are required to implement the division and multiplications required. Trends of the resource utilization for Discrete and Gentle AdaBoost are plotted in Fig. 5.3 and Fig. 5.4.

To ensure the system met timing, the design was floorplanned to form a ring structure. The chosen floorplan results in timing violations when a 31-engine system was constructed since the reserved area became overly dense. In particular, a 31-engine system resulted in a minimum period of 5.485 ns for the compute system or a clock frequency of 182 MHz instead of the desired 200 MHz. Roughly extrapolating the performance trends, speed-up over one CPU with eight threads would increase to about 15.5-fold with 31 engines if the clock frequency could be maintained. However, with the reduction in clock frequency, the overall speed-up would drop to about 14.2-fold. Therefore, without additional effort spent floorplanning the design, the addition of engines would not result in improved system performance.

Component	# Engines	FFs	LUTs	BRAMs	DSP
Compute Core	5	19114 (6%)	24356 (16%)	23 (6%)	21 (3%)
	10	32,121 (11%)	39,851 (26%)	43 (10%)	21 (3%)
	15	45,110 (15%)	55,174 (37%)	63 (15%)	21 (3%)
	20	58,227 (19%)	70,679 (47%)	83 (20%)	21 (3%)
	25	71,302 (24%)	86,071 (57%)	103 (25%)	21 (3%)
	30	84,406 (28%)	101,483 (67%)	123 (30%)	21 (3%)
Full System	30	104,269 (34%)	110,501 (73%)	214 (51%)	21 (3%)

Table 5.3: Hardware Gentle AdaBoost Resource Utilization

Figure 5.3: Hardware Discrete AdaBoost Resource Utilization Scaling

Figure 5.4: Hardware Gentle AdaBoost Resource Utilization Scaling

Figure 5.5: Performance scaling with respect to training set size. Hardware speed-up is shown relative to OpenCV using a single CPU and eight threads.

5.5 Training Set Size Scaling

The decision to use two-phase, table-based sorting results in unique performance characteristics. Full iterations through the 1024-element tables must be performed even when the training set consists of only a few hundred examples. On the other hand, this approach performs well on large training sets. A weak classifier may be examined and bypassed using only iterations in the low-precision phase if no candidate subsets are found. Also, if the feature values are well clustered, only a few high-precision subsets need to be examined regardless of the training set size. Fig. 5.5 shows how the performance of the 30 engine system scales as the training set size increases. As can be seen, the performance relative to software increases with the training set size. This scaling should continue to increase for larger datasets as the feature evaluation step dominates the computation.

5.6 Search Range Reduction

In general, the full 19-bit feature value range must be searched when selecting threshold values during training. However, if the thresholds resulting in low error are expected to be within a much smaller range, the search space can be reduced. Reducing the search space while keeping the same sorted table size has a number of benefits. First,

each subset range becomes smaller. For example, searching an 18-bit range with a 1024-element low-precision pass results in an 8-bit or 256-element high-precision ranges instead of 512-element ranges for a 19-bit search space. Thus, the time to iterate through high-precision ranges is reduced. In addition, since fewer training examples are compressed into each low-precision bin, the $\Delta\epsilon$ measures become more meaningful. This results in fewer candidates as the potential errors are more precise.

Fig. 5.6 shows how the performance of the hardware architecture decreases as the feature search range is expanded. Also plotted is the percentage of times the hardware system transitions into high-precision search. From Section 4.4, it was shown that each entry into high-precision search not only stalls the progress through the features, but also serializes the feature evaluation and threshold selection computations. Thus, it is expected that many transitions to high-precision search result in a large decrease in performance. This relationship is reflected in the diagram where performance is inversely related to the number of transitions into high-precision search. Note that as the number of transitions approaches zero, the gains from reducing the search space level off.

It is important to note that reducing the search range could result in low quality classifiers since thresholds at the limits of the feature value range will not be precisely determined. However for many image datasets, the range of thresholds for useful features will tend to be close to zero, particularly when using zero-mean features.

5.7 Feature Ordering

Results of the feature ordering procedure in terms of the average number of bursts were given in Section 4.3.2. Fig. 5.7 shows the effect of ordering both standard and rotated features on the full system performance. For 30 engines, ordering results in 3.6-fold more speed-up than the unordered features. These results demonstrate the importance of memory bandwidth management and the effect of feature evaluation speed on the overall system performance.

Figure 5.6: Relationship between performance, reduced search range and the number of transitions to high-precision search. Speed-up is shown relative to OpenCV using a single CPU and eight threads.

Figure 5.7: Effect of feature ordering on overall system performance. Speed-Up is shown relative to OpenCV using a single CPU and eight threads.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The Viola-Jones style of object detector has been widely used due to its detection performance and run-time. However, a major drawback of this framework is the extensive training time. A great deal of parallelism is inherent in the training algorithm, but consists of complex individual tasks. To run these parallel tasks to hardware, a multi-pass threshold selection algorithm was developed to take advantage of the high-bandwidth, but limited Block RAM resources on FPGAs. In addition, a method of ordering features was demonstrated to make efficient use of the limited external memory bandwidth. These techniques were shown to map well to a high-throughput, systolic array architecture.

Performance analysis, carried out against an optimized, multi-threaded OpenCV implementation demonstrated the high-performance and scalability of the architecture, where a 15-fold speed-up was achieved over a single processor. These results demonstrate that FPGAs are applicable not only for accelerating the detection portion of the Viola-Jones framework, but also the training process.

An underlying goal of this thesis has been providing the flexibility to explore modifications to the Viola-Jones algorithm while maintaining high performance. By creating highly configurable hardware engines, users may experiment with different committee structures, features, image pre-processing methods and window sizes without re-synthesis of the hardware. However more complicated modifications become somewhat limited by the characteristics of the FPGA substrate and structure of the hardware architecture. Although, the modularity and extensibility of the hardware architecture was demonstrated through the implementation of Gentle AdaBoost, a variant of the algorithm was

chosen when the arithmetic resources of the FPGA were considered. Thus, while the architecture and FPGA fabric are versatile, limited avenues of exploration are available if high performance and minimal design effort are desired.

A further cost of hardware reconfiguration is the design synthesis time. On a large device, hardware synthesis can take several hours; on the order of training a object detector cascade. The performance and energy efficiency of the hardware platform must be weighed against the time to reconfigure the hardware if changes are made. Here, the soft parameterization of the architecture can help to amortize the cost of hardware synthesis if, for instance, several training or feature variations are used for each configuration.

When deployed on an embedded platform in the field, the training algorithm is much less likely to change and the energy efficiency of dedicated hardware outweighs the design effort and synthesis costs. In this case, the scalability and performance of the proposed architecture make it well suited as an optimized detector training engine. Further, the architecture can be used as a framework for developing high-performance training engines on non-programmable substrates for such fixed applications.

6.2 Future Work

As this work is an initial step in an attempt to accelerate Viola-Jones training, there are a number of possible avenues of further research.

Full Cascade Training

The current architecture is optimized to train committees quickly and is thus, not suited to quickly evaluate full cascades. However, this results in a dependence on a host processor to carry out cascade testing after each committee is trained. Work may be done to implement cascade testing within the hardware for use as a standalone system.

Multi-FPGA Training

Partitioning of the algorithm to multiple Object Detector Training Cores is relatively simple since weak classifiers are independent. Furthermore, the MPI platform facilitates the connection of multiple hardware cores together across FPGAs. Care must be taken to avoid synchronization overheads when updating parameters. Nonetheless, extending the

architecture to multiple FPGAs with the resulting increases in available FPGA resources and memory bandwidth should provide substantial improvement in performance.

Algorithmic Optimizations

Some of the algorithmic optimizations used, for example, by OpenCV may also be applied to the hardware architecture. In particular, pre-calculation of features and weight trimming are relatively simple procedures that can provide some extra speed-up, given enough external memory capacity. More complex optimizations to reduce the number of features evaluated could also be an avenue for further development.

Further Algorithmic Extensions

The architecture was designed to be flexible for Viola-Jones style detectors using rectangular features, decision stumps and AdaBoost. However, there are many variants of features that could provide better detection or run-time performance. For instance, local binary patterns [25] are very popular features for fast object detection. They consist of binary strings encoding the local contrast patterns surrounding each pixel. By modifying the feature evaluation core and memory request structure, the architecture could be extended to accelerate training using a wide array of features.

Decision Stumps are extremely simple classifiers and thus amenable to hardware training. However, as with features, there is a large family of possible classifiers for use as the basis of training. A simple extension to the decision stump classifier would be to construct decision trees. Instead of finding a single threshold to divide the feature space, multiple thresholds would be found to subdivide the space further and provide stronger classification performance. Much of the same computational structure could be used to generate trees, but more iterations through the feature space would be required to generate branches.

Finally, Discrete and Gentle AdaBoost were implemented in this work, but there are other methods of creating ensembles such as Real AdaBoost [13] and Asymmetric Boosting [26]. Future research could be carried out to implement and compare these techniques in hardware. An interesting method for implementing these extensions could be the use of high-level synthesis to generate complicated Feature Evaluators or Update Calculators with minimal design effort.

Appendices

Appendix A

Object Detector Training Core MPI Protocol

Commands to the Object Detector Training Core are encoded in 128-bit vectors. Table A.1 shows the instruction encoding. The core supports three operations: *write*, *read* and *run* encoded as 0, 1 and 2 respectively. Write and read operations allow another MPI rank to access the memory interface. Memory request addresses and sizes are encoded in terms of 8-word DDR3 bursts. The MPI Target encodes which MPI rank to send the result to and expected data from, while the MPI Tag specifies the Tag which will be associated with the outgoing or incoming data.

After the initial command has been sent, data transfer will begin. In the case of memory access, data will either be sent to or received from the target rank. However when the run command is sent, the Target rank first sends a set of parameters before awaiting training results from the core. Table A.2 lists the parameters sent to the Object Detector Training Core over six 128-bit vectors. Note that the reserved number of bits for a parameter may be larger than the required number of bits required in a particular platform, in which case the extra bits are ignored.

Bits	Contents
0-31	Memory Address
32-63	Size of Memory Request
64-95	MPI Tag
119-112	MPI Target
121-120	Instruction

Table A.1: Object Detector Training Core Command Format

Vector	Bits	Contents
0	0-31	Number of Standard Features
	32-63	Base Memory Address for Standard Features
	64-79	Image Window Size
	80-127	Initial Sum of All Weights
1	0-31	Base Weight for Hit Rate Calculation
	32-63	Desired Hit Rate
	64-95	Desired False Positive Rate
	96-127	Maximum Number of Weak Classifiers to Train (T)
2	0-31	Number of Negative Validation Examples (V^n)
	32-63	Number of Positive Validation Examples (V^p)
	64-95	Base Memory Address for Positive Aggregate Committee Outputs
	96-127	Base Memory Address for Negative Aggregate Committee Outputs
3	0-31	Base Memory Address for Positive Validation Images
	32-63	Base Memory Address for Negative Validation Images
	64-95	Number of Positive Training Examples (N^p)
	96-127	Number of Negative Training Examples (N^n)
4	0-31	Base Memory Address for Negative Weights
	32-63	Base Memory Address for Positive Weights
	64-95	Base Memory Address for Positive Training Images
	96-127	Base Memory Address for Negative Training Images
5	0-31	Base Memory Address for Rotated Positive Training Images
	32-63	Base Memory Address for Rotated Negative Training Images
	64-95	Base Memory Address for Rotated Positive Validation Images
	96-127	Base Memory Address for Rotated Negative Validation Images

Table A.2: Object Detector Training Core Parameters

Finally, the training results are returned in two 128-bit vectors each time a weak classifier is selected. The format is shown in Table A.3. Here, the weak classifier left and right output may be replaced with just the classifier weight (α) and direction in the case of a discrete weak classifier. When the committee has met its training targets, the Committee Complete bit is set to 1 to signal the termination of training.

Vector	Bits	Contents
0	0	Committee Complete
	1	Rotated Feature
	2-20	Weak Classifier Threshold (θ)
	21-52	Weak Classifier Output Below the Threshold
	53-84	Weak Classifier Output Above the Threshold
	85-104	Stage Threshold (ϕ)
1	0-125	Feature Configuration

Table A.3: Object Detector Training Core Result Format

Appendix B

Feature Evaluator Configuration

Feature Evaluators can combine nine integral image pixels in a linear combination. The 14-bit configuration word for one pixel is shown in Fig. B.1. A pixel location is encoded by the burst in which it is contained and the offset within that burst. Since a full 8-word memory burst is produced by the memory interface over two cycles, the “burst” index in the configuration actually reflects a 4-word vector. This allows a unique burst number to be associated with each 4-word memory vector. The widths shown in the figure reflect a 32x32, 8-bit image, in which an integral image is covered by 74, 4-word bursts, each containing 14 pixels. With nine configuration words required, the total feature configuration vector is 126 bits long.

OP Encoding Scheme:

000	—	+
001	—	—
010	—	2×
011	—	−2×
100	—	+4×
101	—	+3×
110	—	−3×
111	—	NULL

Figure B.1: Feature Evaluator Configuration Word

Appendix C

Table Address Generation

The Sorted Tables core must generate table addresses based on both signed and unsigned values. During a low-precision search, the full feature value range is signed. However, including the sign bit in the table address does not insert training examples in ascending order. Thus, the sign bit must be flipped to create the correct ordering. On the other hand, high-precision searches are always unsigned since no low-precision bin spans zero.

Fig. C.1 demonstrates a case where a subset of the full value range is being searched. The smaller value range is created by shifting the low-precision filter towards the least significant bit (LSB), leaving a set of ignored bits near the sign bit. Accordingly the number of bits used for high-precision search are reduced.

Figure C.1: Sorting Address Generation

Appendix D

Threshold Selector

The Threshold Selector core iterates through the table elements calculating misclassification error at each step. Both misclassification errors directed up and down are calculated using Eqn. 2.16 and are compared to the global minimum error. The minimum of these three values constitutes the local minimum error for a particular Threshold Selector. As the local error for each engine threshold is calculated, it is compared to the error from the previous Threshold Selector in the systolic array. This is shown in the highlighted paths of Fig. D.1.

To keep track of which weak classifier produced the minimum threshold, a one-hot encoding scheme is used. Engine k receives a $(k - 1)$ -bit vector. If it has produced the lowest error so far, it appends a 1 to the vector and sets the rest of the bits to zero. Otherwise, a 0 is appended to the vector with the rest of the contents remaining the same. Using this systolic comparison method, the last Threshold Selector in the array outputs the minimum error, direction, and which engine produced that error. One output is produced for each table entry making the threshold of the error easy to calculate.

The Threshold Selectors must also calculate candidates during the low-precision computation. In this mode, the $\Delta\epsilon$ values, which are simply the contents of negative table bins, are used in addition to errors to calculate potential errors. If these potential errors are less than the global minimum error, then that bin is a candidate range. For classifiers directed down, the candidate bin is the bin *below* the current threshold, or the previous table address. Thus, to find the candidates for a particular bin, the right candidate is delayed by one cycle. The output is encoded in a similar manner as when calculating only error. However, multiple Threshold Selectors could have valid candidates, thus the encoding is no longer one-hot and is instead a bit vector encoding of all engines. The

Figure D.1: Threshold Selector Implementation. The highlighted path shows the error comparison datapath. ϵ_i denotes the misclassification error for table entry i .

minimum potential error for the group of engines is also produced.

Although the squared error is not used for Gentle AdaBoost in this architecture, the P_A, P_B, N_A, N_B values are required to calculate the real-valued classifier output $\hat{h}(\cdot)$ to update the parameters. Thus, these values are also forwarded with the minimum error when the architecture is configured for Gentle AdaBoost.

A final application of the Threshold Selectors is the calculation of hit rate. As explained earlier, finding hit rate involves searching for the threshold where the majority of positive examples are above it. Since only positive validation examples are loaded into the tables during this procedure, the percentage of positive examples above each bin is the same as the error directed down, $P_A + N_B$ since $N_B = 0$.

Appendix E

Update Calculator

The Update Calculator is responsible for calculating the weight update and weights of the classifier outputs.

Discrete AdaBoost

Weight update involves an error factor and a normalization shown in lines 7 and 8 of Algorithm 1. These two steps can be combined to create multiplicative factors for correctly classified examples and incorrectly classified examples. First, notice that the normalization term can be re-written as

$$Z = \beta_t \sum_{i=1}^N 1_{(h_t(x_i)=y_i)} + \sum_{i=1}^N 1_{(h_t(x_i) \neq y_i)}. \quad (\text{E.1})$$

Here, Z is the sum of the weights after the update term β_t is applied. These sums can be written in terms of the misclassification error (Eqn. 2.4) such that

$$Z = \beta_t(1 - \epsilon_t) + \epsilon_t. \quad (\text{E.2})$$

Now, this is combined with the update term in Eqn. 2.5 such that the updates become:

$$\Delta w_{(h_t(x_i)=y_i)} = \frac{\beta_t}{\beta_t(1 - \epsilon_t) + \epsilon_t} = \frac{1}{2(1 - \epsilon_t)} \quad (\text{E.3})$$

$$\Delta w_{(h_t(x_i) \neq y_i)} = \frac{1}{\beta_t(1 - \epsilon_t) + \epsilon_t} = \frac{1}{2\epsilon_t} \quad (\text{E.4})$$

Making use of the fact $\beta_t = \epsilon_t / (1 - \epsilon_t)$. In general, the sum of the weights after normalization should be 1. However, due to imprecision in the weight update, the total weight could be greater to or less than 1, thus it is used as part of the computation and given the symbol J . During each weight update the weights are normalized, thus the error in J does not accumulate. The equations then become $1/(2(J - \epsilon_t))$ and $1/(2\epsilon_t)$. In the case where $\epsilon_t = 0$, the weak classifier perfectly separates the positive and negative examples and the algorithm terminates.

In hardware, the weight update terms are thus determined by calculating the denominators and evaluating a reciprocal function. Since ϵ_t is made up of a sum of weights, it is in the approximate range $(0, J/2]$. If the error were to be greater than $J/2$, the direction would be flipped to produce the smaller error. Therefore, the range of the weight update $\Delta w_{(h_t(x_i)=y_i)}$ is $1/(2J)$ to $1/J$. The lower bound of the incorrect weight update $\Delta w_{(h_t(x_i) \neq y_i)}$ is $1/J$. However, when ϵ_t is very small, $\Delta w_{(h_t(x_i) \neq y_i)}$ approaches infinity, making the reciprocal difficult to estimate. In this case the weight value is also small since it is always less than or equal to ϵ_t . Thus, the final weight value will still be less than 1.

To help estimate the reciprocal, ϵ_t is shifted by k bits such that the denominator comes into the range $[1, 2]$. This process makes the reciprocal function better behaved and easier to estimate. The same floating-point approach is used for $1/(2(J - \epsilon_t))$ in case J deviates far from 1. This just results in a 2^k multiplicative factor, which can be removed when multiplying with the weight. The reciprocal function is finally estimated using a look-up table and three-stage piecewise linear interpolation (PLI) unit [21].

When estimating, $\alpha_t = \ln(J - \epsilon_t) - \ln(\epsilon_t)$, if ϵ_t approaches 0, $-\ln \epsilon_t$ approaches infinity and thus the error in this estimate can become very large. This error is controlled by scaling the ratio $\frac{J - \epsilon_t}{\epsilon_t}$ with a multiplicative factor. A look-up table and three-stage PLI are also used to estimate the natural logarithm. For more precision, the value of α_t may be calculated in software given the trained committee.

Gentle AdaBoost

The weight updates in Gentle AdaBoost are more complicated. Here, there are four possible weight update terms:

$$\Delta w = \begin{cases} \exp(\frac{P_A - N_A}{P_A + N_A})/Z, & \text{if } y_i = +1, f(x_i) \geq \theta \\ \exp(-\frac{P_A - N_A}{P_A + N_A})/Z, & \text{if } y_i = -1, f(x_i) \geq \theta \\ \exp(\frac{P_B - N_B}{P_B + N_B})/Z, & \text{if } y_i = +1, f(x_i) < \theta \\ \exp(-\frac{P_B - N_B}{P_B + N_B})/Z, & \text{if } y_i = -1, f(x_i) < \theta \end{cases} \quad (\text{E.5})$$

The updates are more well behaved than the reciprocals in Discrete AdaBoost, but require more complicated operations. Fig. E.1 shows the computation structure of the Gentle AdaBoost update calculator. The terms within the exponent are first calculated for each of the four terms; this is shown in the heavy black line. As these terms are calculated, the normalization factor Z is also generated based on the equation:

$$Z = P_A \exp(\frac{P_A - N_A}{P_A + N_A}) + N_A \exp(-\frac{P_A - N_A}{P_A + N_A}) + P_B \exp(\frac{P_B - N_B}{P_B + N_B}) + N_B \exp(-\frac{P_B - N_B}{P_B + N_B}) \quad (\text{E.6})$$

Finally, the weight update terms are normalized through division by Z .

In addition to the weight updates, the two possible values for the classifier output need to be determined. Based on Eqn. 2.13, these are:

$$\hat{h}(x_i) = \begin{cases} \frac{P_A - N_A}{P_A + N_A}, & \text{if } f(x_i) \geq \theta \\ \frac{P_B - N_B}{P_B + N_B}, & \text{if } f(x_i) < \theta \end{cases} \quad (\text{E.7})$$

These quotients are available as intermediate results from the calculation of the weight update, and are pulled out as shown in the grey path of Fig. E.1. The path is fully pipelined to calculate the four terms. A look-up table and three-stage PLI are used to estimate the exponential function. Since there are four weight updates, a dual-ported Distributed RAM is used to store the weight updates.

Figure E.1: Block Diagram of Update Calculator for Gentle AdaBoost

Bibliography

- [1] J. Bai, J. Wu, and C. Liu. Motion and Haar-like Features Based Vehicle Detection. In *In Proceedings of the 12th International Multimedia Modelling Conference*, pages 356–359, January 2006.
- [2] T. Breckon, S. Barnes, M. Eichner, and K. Wahren. Autonomous Real-Time Vehicle Detection from a Medium-Level UAV. In *In Proceedings of the 24th International Conference on Unmanned Air Vehicle Systems*, pages 29.1–29.9, March 2009.
- [3] P. Viola and M. Jones. Robust Real-time Face Detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.
- [4] R. Cabrera, T. Tuytelaars, and L. Gool. Efficient Multi-Camera Detection, Tracking, and Identification using a Shared Set of Haar-Features. In *In Proceedings of the 24th International Conference on Computer Vision and Pattern Recognition*, pages 66–71, June 2011.
- [5] C. Zhang and Z. Zhang. A Survey of Recent Advances in Face detection. Technical report, Microsoft Research, 2010.
- [6] M. Saberian and N. Vasconcelos. Boosting Classifier Cascades. In *In Neural Information Processing Systems*. MIT Press, 2010.
- [7] S. Sivaraman and M. Trivedi. Active learning based robust monocular vehicle detection for on-road safety systems. In *Intelligent Vehicles Symposium, 2009 IEEE*, pages 399–404, June 2009.
- [8] D. Hefenbrock, J. Oberg, N. Thanh, R. Kastner, and S. Baden. Accelerating Viola-Jones Face Detection to FPGA-Level Using GPUs. *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 11–18, 2010.

- [9] C. Kyrkou and T. Theocharides. A Flexible Parallel Hardware Architecture for AdaBoost-Based Real-Time Object Detection. *IEEE Transactions on Very Large Scale Integration Systems*, 19(6):1034–1047, 2011.
- [10] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner. FPGA-Based Face Detection System Using Haar Classifiers. In *In Proceedings of the ACM/SIGDA International Symposium on Field programmable Gate Arrays*, pages 103–112. ACM Press, 2009.
- [11] R. Lienhart, E. Kuranov, and V. Pisarevsky. Empirical analysis of detection cascades of boosted classifiers for rapid object detection. In *In DAGM 25th Pattern Recognition Symposium*, pages 297–304, 2003.
- [12] Y. Freund and R. Schapire. Experiments with a New Boosting Algorithm. In *Thirteenth International Conference on Machine Learning*, pages 148–156, 1996.
- [13] J. Friedman, T. Hastie, and R. Tibshirani. Additive Logistic Regression: A Statistical View of Boosting. *Annals of Statistics*, 28(2):337–407, 2009.
- [14] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [15] B. McCane and Kevin Novins. On Training Cascade Face Detectors. *Image and Vision Computing*, 9:11, 2003.
- [16] J. Wu, J. Rehg, and M. Mullin. Learning a Rare Event Detection Cascade by Direct Feature Selection. In *In Neural Information Processing Systems*. MIT Press, 2003.
- [17] Xilinx Inc. Zynq-7000 Extensible Processing Platform. Product Brief.
- [18] C. Demirkir and B. Sankur. Face Detection Using Look-Up Table Based Gentle AdaBoost. In *In Proceedings of the 5th International Conference on Audio- and -Video-Based Biometric Person Authentication*, pages 339–345, July 2005.
- [19] M. Saldaña, A. Patel, C. Madill, D. Nunes, D. Wang, H. Styles, A. Putnam, R. Wittig, and P. Chow. MPI as an Abstraction for Software-Hardware Interaction for HPRCs. *International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 1–10, November 2008.
- [20] JEDEC Solid State Technology Association. DDR3 SDRAM Specification, JESD79-3E. JEDEC Standard, July 2010.

- [21] D. Ly and P. Chow. A Multi-FPGA Architecture for Stochastic Restricted Boltzmann Machines. In *In Proceedings of the 19th International Conference on Field Programmable Logic*, pages 168–173, August 2009.
- [22] C. Loken, D. Gruner, L. Groer, R. Peltier, R. Bunn, M. Craig, T. Henriques, J. Dempsey, C. Yu, J. Chen, J. Dursi, J. Chong, S. Northrup, J. Pinto, N. Knecht, and R. Zon. SciNet: Lessons Learned from Building a Power-efficient Top-20 System and Data Centre. *Journal of Physics: Conference Series*, 256(1):012026, 2010.
- [23] J. Phillips, H. Moon, S. Rizvi, and P. Rauss. The FERET Evaluation Methodology for Face-Recognition Algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(10):1090–1104, 2000.
- [24] R. Fergus, P. Perona, and A. Zisserman. Object Class Recognition by Unsupervised Scale-Invariant Learning. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 2, pages 264–271, June 2003.
- [25] T. Ojala, M. Pietikäinen, and D. Harwood. A Comparative Study of Texture Measures with Classification Based on Featured Distributions. *Pattern Recognition*, 29(1):51–59, 1996.
- [26] P. Viola and M. Jones. Fast and Robust Classification using Asymmetric AdaBoost and a Detector Cascade. In *In Neural Information Processing Systems*, pages 1311–1318. MIT Press, 2001.