# A Flexible Parallel Hardware Architecture for AdaBoost-based Real-Time Object Detection

Christos Kyrkou, *Student Member, IEEE,* and Theocharis Theocharides, *Member, IEEE*

***Abstract*—Real-time object detection is becoming necessary for a wide number of applications related to computer vision and image processing, security, bioinformatics, and several other areas. Existing software implementations of object detection algorithms are constrained in small-sized images and rely on favorable conditions in the image frame to achieve real-time detection frame rates. Efforts to design hardware architectures have yielded encouraging results, yet are mostly directed towards a single application, targeting specific operating environments. Consequently, there is a need for hardware architectures capable of detecting several objects in large image frames, and which can be used under several object detection scenarios. In this work, we present a generic, flexible parallel architecture, which is suitable for all ranges of object detection applications and image sizes. The architecture implements the AdaBoost-based detection algorithm, which is considered one of the most efficient object detection algorithms. Through both FPGA emulation and large-scale implementation, and RTL synthesis and simulation, we illustrate that the architecture can detect objects in large images (up to 1024x768 pixels) with frame rates that can vary between 64-139 fps for various applications and input image frame sizes.**

***Index Terms*—Object Detection, Systolic Arrays, Very Large Scale Integration.**

## I. Introduction

OBJECT detection in video and images is an important operation in several embedded applications, such as computer vision and image processing applications, bioinformatics, security, and artificial intelligence. Object detection involves the extraction of information from an image (or a sequence of) frames, processing of the information, and determining whether the information contains a particular object and its exact location in the image. This process is computationally intensive, and several attempts have been made to design hardware-based object detection algorithms, especially in the context of embedded and real time systems [1]-[5], [8]-[12], [16], [18]-[24], and [28]-[29]. This is particularly emphasized in safety-critical applications such as search-and-rescue operations, biomedical applications (such as laparoscopic surgeries), surveillance of critical infrastructure and several other applications. The majority of the proposed works target FPGA implementations; additionally, they are either application-specific or operate on images of relatively small sizes in order to achieve real-time response [8], [16]-[17]. As such, a generic, real-time object detection hardware architecture, independent of image sizes and types of objects, can potentially benefit several applications, and most importantly, provide the foundations for further post-detection applications such as object recognition.

There are several algorithms used to perform detection, each of which has its own advantages and disadvantages. This paper presents a generic architecture based on the object detection framework presented by Viola and Jones [6] where they utilize the AdaBoost learning algorithm introduced by Freund and Schapire [7], [13]. The proposed architecture extends our preliminary work proposed initially in [8]. In this work, we extend the implementation of the AdaBoost detection framework by several algorithm-driven design optimizations, focus on the general object detection problem, and address the image size limitations. We also expand our evaluation strategy to include both an FPGA implementation for the purposes of validation of our architecture, as well as an ASIC implementation, for which we evaluate based on three different object detection case studies.

The architecture proposed in this work is based on a massively parallel systolic computation of the classification engine using a systolic array implementation which yields extremely high detection frames per second (fps). The architecture is designed in such a way as to boost parallel computation of the classifiers used in the algorithm, and parallelize integral image computation, reducing the frequency of off-chip memory access. To make the architecture scalable in terms of image sizes, we utilize an image pyramid generation module in conjunction with the systolic array. As the array elements are modular and simple, and communication is regular and predetermined, the architecture is highly scalable and can operate on high frequency. The designer can select all the appropriate design parameters with the targeted operating environment in mind, without affecting the real-time constraints. The designer can also choose the operating frequency (with power constraints in mind), the array size (with area constraints in mind), and image size (with targeted application specifications in mind). The architecture is flexible as well in terms of input image size; the maximum input image size depends on the silicon budget available, however smaller images may easily be processed by the system as the input image size can be loaded as a parameter. Moreover, the architecture can support different training sets and different training set formats.

The architecture is evaluated by verifying its operation on a Xilinx Virtex II Pro FPGA, and by synthesizing and

implementing the architecture using Synopsys Design Compiler and a commercial CMOS 65nm cell library. The rest of this paper is organized as follows. First, a detailed description of the algorithm is given in section II, where the hardware implementation requirements are outlined. Section II also gives a review of related work. Section III presents the proposed architecture, and section IV presents the systolic computation overview, explaining dataflow and computational semantics. Section V presents the evaluation framework along with the results for both the FPGA implementation and the ASIC synthesis. Section VI concludes this paper, giving future directives.

## II. Background and Related Work

### *A. The AdaBoost Algorithm and Hardware Implementation Requirements.*

The method of utilizing *AdaBoost* as part of a learning algorithm for robust real-time object detection was first introduced by Viola and Jones [6], in order to select a number of visual features for producing efficient and accurate classifiers. AdaBoost utilizes a small number of weak-classifiers, which are then used to construct cascades of strong classifiers. The combination of the strong classifiers in a cascade results in high accuracy rates and computational efficiency.

The most popular weak classifiers used with AdaBoost are the *Haar-like features*; fixed-size images which contain a small number of black and white rectangles. These features act as filters that can detect the presence or absence of certain visual characteristics in an image. The computation of a *Haar-like feature* involves calculating the sum of the pixel values in the white rectangles of the feature minus the sum of pixel values in the black rectangles, by convolution with the input image. The original algorithm [6] used features starting at 24x24 pixels, however, the feature size can vary with the application. The number of rectangles for each feature varies also depending on the object of interest. Rectangles and features, along with the feature operation are shown in Fig. 1.

The strong classifiers constructed by AdaBoost are setup in a cascade and each strong classifier is a stage in the detection process. Each stage consists of a group of *Haar-like features* selected by AdaBoost during training. The outcome of each *Haar-like feature* in a stage is computed and accumulated. When all the features in a stage are computed, a *stage threshold value* ($t_o$) is used to determine if the sample is a successful candidate to move on to the next stage or not. This technique accelerates the process of rejecting an image region that does not contain objects of interest, so that computation time will focus only on successful candidates.

When a cascade of stages of features is computed, the outcome for the search window for which the cascade is evaluated is known. However, objects in the image frame which are larger than the search window and the feature, do not get detected. This is usually solved by downscaling the original image frame, subsequently reducing the object size, and making it detectable. However, Viola and Jones suggest enlarging the feature instead; this way, image data that could potentially be lost by downscaling remains, and the features that are simply black and white rectangles, scale linearly without loss of data. Consequently, at the end of each cascade computation, the process is repeated for a larger feature size, until the size of the feature reaches the size of the largest possible object (in terms of pixels) in the input image frame. The amount of scaling also impacts the detection frame rate significantly, which further stresses the need for rapid feature computation.

To speed up the feature computation, Viola and Jones propose an alternative input image representation, called the *integral image*. The integral image is simply a transformation of the original input image, to an image where each pixel location holds the sum of all the pixels to the left and above of that location [6]. The advantage of using the integral image is the ability to compute the sum of a rectangle in a rapid manner. As shown in Fig. 1 (rectangle computation), the computation for rectangle D is simply two additions and two subtractions of the four corner points of the rectangle when using the integral image rather than the original image. Hence, regardless of the feature or search window size, only four values per rectangle are necessary to compute the value of each feature. Additionally, the location of the rectangles within each feature is predetermined from the training set, hence to evaluate each rectangle we need the offset *dx* and *dy* values from the starting coordinate of each feature (see Fig. 1, center-bottom). The offset coordinates are part of the training set, where each feature is associated with a list of the feature's rectangles and the four pairs of (*dx,dy*) offsets necessary. This holds true during feature upscaling as well, as since the rectangle coordinates are fixed, *dx* and *dy* are also scaled linearly with the scale factor. For example, if a feature scales from 24x24 to 30x30, (i.e. a scale factor of 1.25) a rectangle that in the initial feature would be located at starting coordinate (*dx = 4, dy = 8*) would be mapped to (*dx = 5, dy=10*), with *dx* and *dy* multiplied by the scale factor (rounded to the nearest integer).

One important disadvantage of the integral image computation however lies in the implementation of the addition and storage required for computing and storing the integral image values. As the size of the input image grows, the adder and storage grow proportionally as well. Recall that an integral image pixel located at (*x, y*) holds the sums of all pixels above *y* and to the left of *x* (Fig. 1). As the range of x and y grows, the amount of pixels summed for computing the value of integral image pixel (*x, y*) grows exponentially (*x*y*); hence, the adder precision and memory requirements change as well. This can be addressed by applying the algorithm over smaller regions of the input image rather than the entire image.

All the above computations are essential for the classification process required by the detection algorithm. There are also some additional computations necessary to deal with the varying characteristics in which the object of interest may appear, due to the lighting and environmental variations. The AdaBoost framework uses a lighting correction technique to compensate for these variations. This technique requires the computation of the squared integral image for each input image (each image location holds the sum of the squared pixel values). This is necessary to compute the variance (VAR) and the standard deviation (σ) of the image, to compensate for the

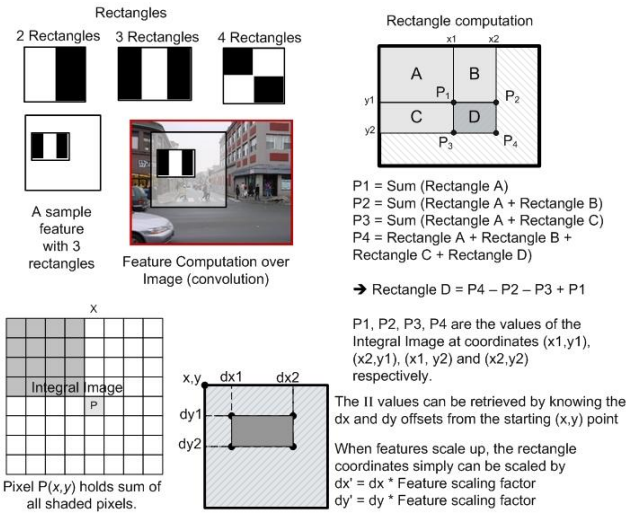


Fig. 1. Basic Concepts used in the AdaBoost object detection framework

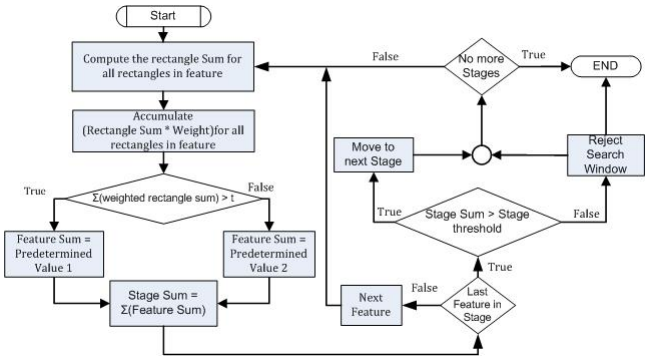


Fig. 2. Outline of the AdaBoost-based classification procedure

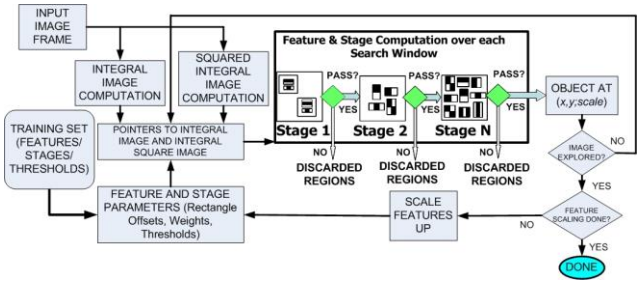


Fig. 3. Stage Evaluation outline

lighting variations, as shown in (1). The standard deviation is multiplied with the original feature threshold ($t_0$) given in the training set, to obtain the *compensated threshold* ($t$) which dynamically takes care of any lighting variations encountered during the detection stage, and improves the overall accuracy of the algorithm. It is needed to be done only once for every search window however; all subsequent features evaluated over that search window can use the computed standard deviation value as shown in (1).

Given that the search window size is known, we avoid the costly division operation using the reciprocal of the area as a constant, and multiply it. The sum of the pixels is then squared, and subtracted from the computed value of the squared integral image, to give us the variance. To compute the standard deviation, we need the square root of the variance. The square root however is a tedious operation when it comes to hardware, so a better solution could be explored. To compute the *compensated threshold* we need the product of the original threshold and the standard deviation ($\sigma_i$), as shown in the first line of (2). We therefore square both sides of the equation, multiplying the variance with the squared value of the original threshold (can be pre-computed during training and stored in the training set), to yield the squared value of the compensated threshold. Thus, the square root operation becomes a multiplication instead.

$$VAR = \sum_{i=0}^{\#\ of\ pixels} \left[\frac{\sum_{i=0}^{\#\ of\ pixels} X_i}{AREA}\right]^2 \quad and\ \sigma = \sqrt{VAR} \qquad (1)$$

$$t = t_\sigma * \sigma_i \Rightarrow t^2 = (t_0)^2 * VAR \qquad (2)$$

The sum of all weighted feature rectangles is used to determine the feature sum; if this sum exceeds the *compensated threshold* then it is set to a predetermined value obtained from the training set. Otherwise it is set to another predetermined value, also obtained from the training set. All feature sums are also accumulated to compute the stage sum. At the end of each stage, the stage sum is compared to a predetermined *stage threshold*. If it is larger, the search window is a successful candidate region to contain the object of interest; otherwise, it is discarded, and omitted from the rest of the computation. When all search windows finish, features are scaled by a predetermined factor, to detect objects larger than the original feature size. The computation is repeated again, using new training values set for the larger scale, until all objects of all sizes are detected in the image frame. The algorithm computation outline and the stage evaluation outline are shown in Fig. 2 and Fig. 3 respectively.

To map this algorithm in hardware, we need to determine an efficient access to the values of the integral image used to compute the rectangle outcomes. Given that the bulk of the computation focuses on computing each feature, and given that the computation is identical, the challenge shifts to finding an efficient way to access the values of the integral image in parallel, and be able to employ the inherent parallelism of computing these rectangles over the entire image. Thus, storing the integral and integral squared images into single memory blocks essentially limits the number of rectangle coordinates accessed in parallel and creates contention. Similarly, replicating the memory blocks to increase parallelism results in high memory requirements, and if the memory is off-chip, in an increased latency [15]. Thus, we present an architecture that provides parallel access to the integral image values, and provides parallel data movement to result in rapid computation of rectangles across the entire image frame. Next, we discuss related work and alternative implementations.

### *B. Related Work*

The majority of object detection hardware implementations deal with specific applications, and are designed aiming performance towards the specific host environment. Some early work targets neural network implementations, such as [14], [16], [17], and the AdaBoost algorithm has only recently gained attention as a promising alternative. The AdaBoost-based visual object detection framework is suitable for a wide range of computer vision applications and has been used in various tasks involving detection. The majority of these implementations however, have been done using software;

TABLE I
RELATED WORK ALGORITHM AND METHOD COMPARISONS

| | Hiromoto [9][a] | Cho [10] | Wei [11] | Shi [12] | Lai [28] | Presented Work - FPGA |
|---|---|---|---|---|---|---|
| **# Features** | 2,913 [18] | 2,135 | 225 | 2,913 [18] | 52 | 2,913 [18] |
| **# Stages** | 25 [18] | 22 | 3 | 25 [18] | 1 | 25 [18] |
| **Image Size** | 640x480 | 320x240, 640x480 | 120x120 | 176x144 | 640x480 | 320x240 |
| **Feature Size** | 24x24 | 20x20 | 24x24 | 24x24 | 20x20 | 24x24 |
| **Scaling Method** | Image Downscaling | Image Downscaling | Image Downscaling | Not provided | Image Downscaling | Image Downscaling/ Feature Upscaling |
| **Downscale Factor(s)** | 1.2 | 1.2 | 1.25 | Not provided | 1.25 | 1.25, 0.75, 0.5 |
| **# of Downscaled Images** | 18 | 14 (320x240), 18 (640 x 480) | 4 | Not provided | ~15 | 3 downscaled images, 5 upscaled features. |
| **II Computation** | Per window, using line buffers to calculate and store the II values | Per window, by an array of line buffers and block RAMs | Sequential computation per window | Per Window, using a 24x24 Cell Array | Computed for every window by a 21x21 Register Array | For the whole 80x60 window while values are shifted in the array |
| **Rectangle & Feature Computation** | II values are loaded from a register array, processed in parallel for the first 10 stages and sequentially for the rest. | Accumulates the II values from the array and evaluates the feature | Evaluated using MAC units | The II values are loaded from the array and used to compute the rectangle and feature sums | II values are loaded from the array and are weighted and summed to evaluate a rectangle | CCUs and EUs evaluate rectangles and features during the systolic flow of II values in the array |

[a]**Uses a sequential and parallel processing execution model. Split point for the two stages is at stage 10, II = Integral Image**

hardware implementations have been limited to the application of face detection [3], [8-12], [28] and [29] and have mostly targeted FPGAs. The majority of the proposed hardware implementations follows some basic principles that were originally introduced in our preliminary implementation in [8], and suggests the computation of the integral image as well as the access of its values to be implemented as a systolic array rather than using a central (or even, distributed) memory.

Recent work by M. Hiromoto et al [9] proposed a hybrid model consisting of parallel and sequential modules. The parallel modules are assigned to the early stages of the algorithm which are frequently used whereas the latter stages are mapped onto sequential modules as they are rarely executed. However, the separation of the parallel and sequential stages needs to be reevaluated every time there is a change in the training data. [10] uses a cell array architecture for the main classification modules. The integral image is generated for each sub-window, and is then used for classification through a cell array. Additionally the input image is scaled down instead of the *Haar-like features* scaling up. A simpler version of the algorithm was implemented in [11] where only 3 stages of classification are used, with an input image size of a 120x120 image. The integral image is computed for each sub window that is generated, rather than the whole image. Furthermore, an image pyramid generation unit is used to produce downscaled versions of the input image. In the work presented by Y. Shi et al [12] some optimization methods are suggested to speed up the detection procedure when considering systolic AdaBoost implementations. The proposed work introduces two pipelines in the integral image array to increase the detection process; a vertical pipeline that computes the integral image and a horizontal pipeline that can compute a rectangle feature in one cycle. The implementation in [28] employs very similar architecture to the ones presented in [9] and [10] but with a massively reduced number of training features and thus manages to process more than 100 fps, illustrating that training set optimizations are also a factor that can be potentially explored. However, no information is given on the method used to reduce the training set.

A specialized recognition processor was presented in [29] that introduces three techniques related to the handling of Haar-like features. A cache is used to hold recently referenced training data; a feature coordinator decoder is used for faster access, and a Haar-feature value extractor is used to improve throughput. More recently, a SystemC implementation was presented in [24], where initial simulations showed an achieved frame rate of 42fps, under a modified architecture and reduced training set. The contributions of the work in [24] also detail opportunities for system-level optimization, using modern design tools such as SystemC.

Table I presents a summary of existing implementations, and gives a brief comparison in terms of the training sets used (features and stages), image and search window size, scaling techniques and the impact on the number of resulting search windows. Table I also provides a brief comparison in terms of the methodology employed in computing the integral image and the rectangle sums.

To the best of our knowledge, this work is the first that considers a full-custom generic AdaBoost hardware implementation. We tackle all issues such as input image size, object types and number of objects of interest present in the input image, but most importantly, we allow the architecture to remain generic to process different training sets and feature sizes, making the architecture suitable for all types of AdaBoost-based object detection.

## III. PROPOSED ARCHITECTURE

There are five major issues in designing the proposed architecture: image scaling, integral image computation, feature computation, stage computation and identification of regions that contain the objects of interest. Each part is considered based on its contribution towards the performance and accuracy, as well as computational resources required.

The architecture consists of 2 major blocks; an image

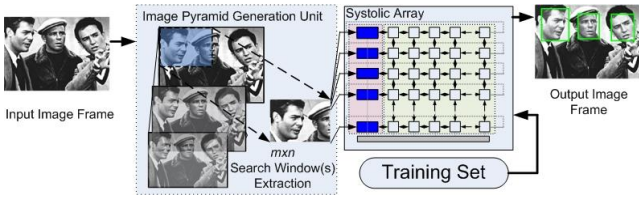


Fig. 4. System architecture block diagram.

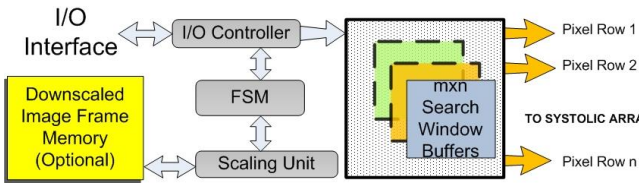


Fig. 5. Image Pyramid Generation Unit Architecture

pyramid generation (IPG) unit [14]-[17] and a systolic array. The IPG receives the input video frame and generates image regions that the systolic array processes. The systolic array evaluates the candidate regions that potentially contain objects of interest. We use a hybrid scaling mechanism that utilizes traditional input image downscaling, as well as the original feature upscaling scheme that Viola and Jones proposed. . This feature upscaling continues to iterate, in order to detect objects in the search window larger than the feature size. Iterations continue until the feature size equals the size of the smaller dimension of the search window size (typical features are square, whereas search window sizes can be rectangular). For example, if we consider a starting feature size of 24x24, and a search window size of 80x60, features will be scaled up until the feature size will be 60x60. In this way, we reduce the image size where features are being evaluated, reducing the overall cost and amount of computation, and still allow the system to process large input images.

The IPG unit processes the input image frame and generates the search window. In the context of this work, the search window is defined as the image region examined for the targeted object(s) of interest. Each search window is then processed in parallel, feature by feature and stage by stage through the systolic array. The array is responsible for computing the integral image, computing the rectangles for each feature, and evaluating both the features and the stage sumsIf an object of interest is detected within that search window, the array outputs the coordinates of the region containing the object, taking into consideration the scale of the feature that the object was found at. When the array completes the examination of a search window, a new search window is fed into the array from the IPG unit, until the entire image is searched. The original image is also downscaled through the IPG unit, producing more search windows for each downscaled version of the original image, until the downscaled image equals the search window size. This creates a hybrid architecture that evaluates features over a number of image scales, and a number of feature sizes over a single search window. A brief description of each of the units is given next, while a block diagram of the system architecture is shown in Fig. 4.

### *A. Image Pyramid Generation*

The IPG unit receives the input video frame and generates the search windows to be processed by the systolic array. The unit receives pixels row-wise, and generates *mxn* search windows, which are then buffered and fed row-wise in parallel in the systolic array. The size of the generated search windows is determined by the size of the systolic array. The IPG and the systolic array operate in a pipelined fashion, where the systolic computation happens as soon as a single search window is generated. However, the IPG continues to generate search window pixel data while the systolic array is computing, preparing the next search window(s) that will be used. Typically (depending on the systolic array size and subsequently search window size), the IPG can generate a second search window before the first one is computed by the array, therefore one search window buffer is sufficient.

The IPG unit also downscales the original image, ensuring that objects bigger than the search window size are downscaled, and eventually can fit into a search window as well. In this way, the search window can be made as large as the silicon budget allows the systolic array to be. Moreover, data loss due to downscaling is limited, as the image will not be scaled down after it reaches a certain size.

The IPG unit consists of three stages; the input stage, where pixels are received from the frame memory, the partitioning stage where incoming pixels are partitioned into the search window buffer, and the scaling stage. The first stage is customized to satisfy the input conditions (i.e. number of pixels per cycle, etc.). The second stage is a finite state machine that is responsible for generating the address of the pixel values that are to be received in the next I/O operation and directs incoming pixels in their corresponding search window buffer location. Lastly, the scaling stage simply computes the coordinates (and subsequently memory address) of the downscaled image for each incoming pixel, generating the address where each pixel is to be stored. It must be noted that depending on the choice of the downscaling algorithm used, some pixels will be mapped to the same location. In the proposed IPG, the algorithm used is a simple multiplication, and the pixel that was lastly computed to be stored in the generated location, overwrote any previously written pixels. Additionally, the downscaled image (depending on the generated size and thus its memory requirements) can be stored either on-chip or on external memory, and retrieved at a later stage during the computation in similar fashion as the original image is received. This procedure was chosen to enable flexible scaling of downsized images, allowing the designer to select the scale and the number of produced downsized images. The IPG unit is shown in Fig. 5. The output search windows are fed pixel by pixel, row-wise, in the systolic array.

### *B. Systolic Array*

The systolic array performs the bulk of the computation; it computes the integral image, collects and computes the rectangle points, computes and evaluates the feature and stage sums, and determines whether a region passes a stage so that it

can be considered for further search. The array also maintains the location of detected objects. The array consists of two types of processing elements (PEs); the collection and computation units (CCUs), and the evaluation units (EUs). The EUs are placed as the leftmost PEs in each row in the array; the CCUs make up the rest of the array. Each EU communicates via a direct link to its neighboring CCUs, and a toroidal link to the far right CCUs, as shown in Fig. 6. The array also contains distributed control units (CUs), small FSMs that direct the overall operation by global control signals. The distributed control units also maintain the temporal consistency of the entire operation, acting as coordinators throughout the entire computation. Given the modular operation of the array, and the identical operation of each of the CCUs and EUs respectively, the control units maintain that communication is uniform across the array and towards the necessary direction, and that all units are synchronized, either doing data transfer, or computation. Distributed multiple CUs can be used, in order to reduce the size of the control region for each CU, since the control signals delivered to the PEs are identical. The array units can communicate with each of its neighbors via bidirectional data links.

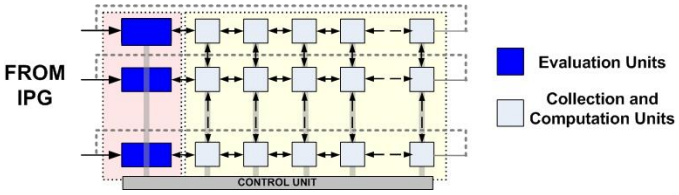


Fig. 6. Systolic Array System Architecture

The chosen array size depends largely on the application requirements in terms of frame rate and budget, and the training set and feature sizes used in the detection algorithm. The minimum size has to match the original size of the features, whereas the maximum size of the array can be determined based on the silicon budget available. The dimensions of the array can be made to match the proportions of the input image frame. The implementation presented in this paper uses an image ratio of 4:3, but the ratio can be adjusted to match input frame format.

The systolic array operates by firstly computing the integral image for the incoming frame. Next, it computes the rectangles for each feature in parallel for the entire search window. Stage evaluation is also done in parallel for all locations in the search window, and after the outcome of each stage is known, the array proceeds by evaluating the next stage and its features in parallel over the entire search window. The candidate regions that fail each stage are marked and do not participate in the computation, in an attempt to eliminate unnecessary power consumption. Every CCU can act as the top-left-most corner for each feature, and is responsible for collecting the integral image values belonging to the rectangles for that particular feature. Each CCU holds the integral and integral squared image values, partial sums from rectangle and feature computations, and the variance for the search window that they represent as the top-left-most corner. Each CCU consists of minimal hardware to propagate data in all directions in the array, and is able to perform additions and subtractions, enabling the computation of the integral and integral squared image in a systolic manner. The rectangle sum can also be computed within the CCUs. The EUs are equipped with multiplexing hardware and contain a multiplier for stage evaluation purposes (to compute the weighted sum of each rectangle in each feature and the feature sum). Fig. 6 shows a part of the systolic array with the three units composing the array. A brief description of the hardware architecture for each array unit is given next. The architecture of each array unit is shown in Fig. 7.

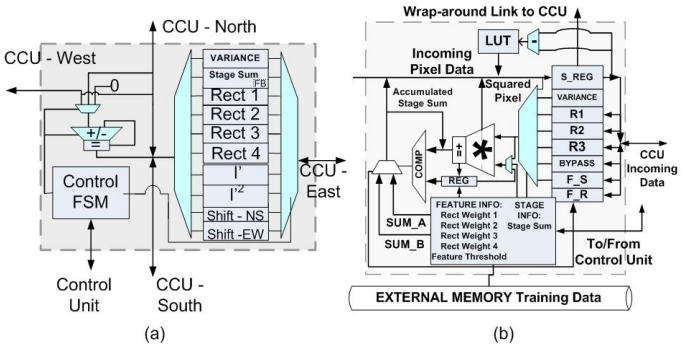


Fig. 7.(a) Collection and Computation Unit Architecture. (b) Evaluation Unit Architecture.

*1) Collection and Computation Unit (CCU)*

Each CCU represents the starting upper left corner of a search window in the image, and holds the necessary data for that window (such as image variance, whether or not the window has so far passed the classification process, etc.). The CCUs are responsible for data movement throughout the system, collecting and accumulating integral image data for rectangle computation. Each unit is composed of an adder/subtractor, a local bus controller and a register file that holds the data necessary for the computation. The register file provides data storage for the integral image value, the squared integral image value, the collected rectangle sums (supports up to four rectangles per feature), the accumulated stage sum, the standard deviation of the image for the search window represented by that particular CCU, and temporary registers used to store data during data movement. Furthermore, the CCU holds a flag bit (FB) which is reset only when the search window represented by the CCU does not contain the object of interest. The bit is set at the beginning of every computation and is reset by the EUs at the end of a stage computation if the search window represented by that CCU does not pass a stage. To maintain temporal consistency, the bit is moved with the accumulated stage sum. A detailed block diagram of the CCU is shown in Fig. 7(a). The CCU's critical path lies in the adder; depending on the required bit-width of the adder, various optimizations can be made to improve the speed of the CCUs.

Each CCU performs a set of predetermined actions. These are shifts to all four directions, addition and accumulation of incoming pixel values and squared pixel values, addition and

accumulation of incoming rectangle points, and being idle. Each action is determined by the CUs, which send a global op-code of 4 bits to all CCUs, so that all CCUs can synchronize on the appropriate action.

One particular design parameter that the algorithm dictates lies on whether all CCUs should act as collection points for each feature. If we evaluate each feature over every possible location of the search window, then each CCU has to act as a collection and computation point. This is not always necessary however; it depends on the type and size of objects of interest. For example, a large object does not need a pixel-by-pixel exhaustive search; a search offset of five pixels will probably be sufficient. This is of course something that requires experimentation and an appropriate training set. Typical object detection algorithms follow a small pixel offset [6], [17], [25], and [26], shifting each search window in an image by a few pixels (depending on the object size). Suggested values are around five pixels or more [6], [8], and [14] for this offset. The modularity of the systolic array allows the designer to take advantage of this offset, by designing CCUs that are capable of collecting and computing rectangle information, and CCUs that do not. Additionally, CCUs located at the bottom and left parts of the array are not required to act as upper left collection points, since the feature computation will have reached the end of the search window. Therefore, we can design a set of CCUs without the adder/subtractor, and without the control logic necessary to perform collection and computation. These CCUs simply act as memory elements, and can be placed in locations in the array where the CCUs are not required to collect rectangle data.

An additional design optimization lies in the design of the adder inside each CCU and the registers holding the integral image and integral squared image values. As the location of each pixel in the integral image, relative to the integral image's origin increases, the value of the integral image (and the integral square image) increases in terms of required adder precision and in terms of storage requirements. This is illustrated in Fig. 8(a). For example, if we are dealing with a 320x240 grayscale image (8-bits per pixel, maximum intensity of 255), the maximum value that needs to be stored at the bottom-right corner (*location 320,240*) of the integral image, is 320x240x255 (in the unlikely event that all pixels have intensity value of 255). This requires 25 bits. Since we also need the integral squared image, the bitwidth requirements increase to 33 bits for the adder. However, at location (*20, 40*), the maximum value that will be stored is 20x40x255, which requires only 18 bits for the integral image, and 26 bits for the adder, to compute the integral squared image. Fig. 8(b) shows the bit-width requirements of the adder, in a sample 320x240 image, to indicate a relative hardware demands as the location in the array changes.

Consequently, we can design parameterized CCUs, with variable adder bit-widths and variable-sized registers, which can be appropriately placed depending on the distance of each CCU relative to the origin of the array. This can be done either by one-by-one CCU case, or by designing different groups of CCUs with different bitwidths that can cover regions in the array, allowing some CCUs to have redundant bits. We followed this approach, as it is less time consuming; moreover, an extra bit or two in each CCU adder, does not add much in the hardware overheads. Alternatively, this can be done by limiting the search window size; this helps keeping the overall number of pixels required for both integral and integral square image summations small, resulting in relatively small bit-widths.

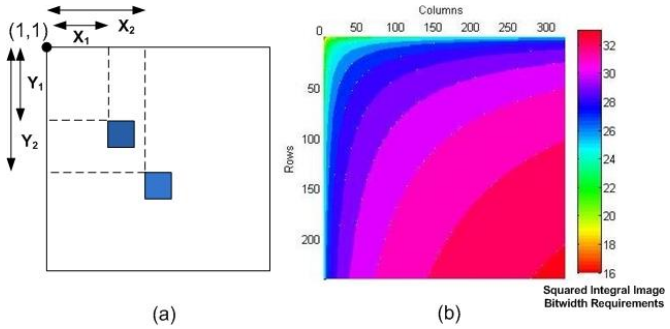


Fig. 8. (a) As we move away from the origin coordinates of the integral image, the demands of the required bit-width for the storage of the rectangle sum increase. Position ($X_1$, $Y_1$) requires fewer bits than position ($X_2$, $Y_2$) because position ($X_1$, $Y_1$) holds the sum of less integral image points. The required bit width for coordinates ($X_i$, $Y_i$) is $\log_2\big((\#intensity\ values - 1) \times X_i \times Y_i\big)$. (b) Illustration of how the requirements for the adder and register bitwidth changes according to their location (row and column) in the array for a 320x240 array. The label on the right denotes the bitwidth requirements per array region.

*2) Evaluation Unit (EU)*

The EUs are located to the left of the array, at the beginning of each row, and act as input terminals to the array. The EUs are first used during the input of the search window into the array to compute the integral squared image values, by squaring each incoming pixel value to be used towards the squared integral image computation. During the computation, the EUs receive data from their neighboring CCUs (and through systolic manner, eventually from all CCUs in the corresponding row of each EU), starting from the rectangle values, the variance of the image and lastly the accumulated stage sum. The rectangle sums are multiplied with the rectangle weights read from the training set stored in off-chip memory. The variance is multiplied with the squared feature threshold, to determine the compensated threshold, which in turn is used to determine the feature sum to be added to the accumulated stage sum. If the computed feature is the last feature of a stage, the accumulated stage sum is compared to the stage threshold and the flag bit is reset if the stage computation discards the search window. Else, both the accumulated stage sum and flag bit are shifted out using a toroidal link into the far right CCU. The EU starts the computation when signaled from the CUs; when the computation ends, the EU signals to the CUs to proceed with the next feature. The CU in the meantime stalls shifting in the CCU values during an EU computation, waiting on the EU to complete. When a stage is evaluated, the EU sends a signal to the CUs, so that they can coordinate all CCUs in starting the next stage of features.

Each of the EUs interfaces to the external memory that holds the training data necessary for the feature computation, and reads the training data in a FIFO manner. Given that features are evaluated one at a time, the latency to retrieve the feature values does not affect the overall latency, as this can happen while the CCUs evaluate the feature's rectangles. This also enables storing of the training data (offsets) for all feature sizes, thus removing the need to scale the rectangle offsets (*dx, dy)* dynamically when the computation shifts to larger feature sizes. It must be noted however, that feature scaling can be done on-chip as well, where each (*dx, dy*) offset can be scaled according to the preset feature scale factor. Upon computation of each feature, the next feature rectangle off-sets are read from the training memory and propagated along with the resulting feature sum, using the torroidal link, back to the CCUs. Consequently, when a feature is evaluated with the rightmost CCUs receiving the last outcome of each computation, the entire array already holds the required off-sets for all rectangles associated with the next feature.

The EU block diagram is shown in Fig. 7(b). The EU multiplier has the longest critical path in the array, and various optimizations can be done to improve the frequency of the unit, such as using pipelined or wave pipelined multipliers.

## IV. Systolic Computation Overview

The operation essentially is partitioned into the following stages: configuration, computation of integral and integral squared images, computation of image variance, computation of rectangles per feature, feature computation, stage evaluation and image evaluation. The computation is repeated for all upscaled features over a single search window, and for all search windows generated by the IPG. When the image has been searched at all search window sizes, the system is ready for the next image frame.

In each case, all units collaborate to perform the computation. Incoming pixels, stream in the array in parallel along all rows of the array, and are shifted in row-wise every cycle. The integral image and integral squared image are computed first. The computation consists of horizontal and vertical shifts and additions. Incoming pixels are shifted inside the array on each row. Depending on the current pixel column, each of the computation units performs one of three operations; it either adds the incoming pixel value into the stored sum, or propagates the incoming value to the next-in-row processing element while, either shifting and adding in the vertical dimension (downwards) the accumulated sum or simply doing nothing in the vertical dimension. The computation is illustrated in Fig. 9. To compute the squared integral image, the same procedure is followed. The incoming pixel passes through the multiplier in the EU, which computes the square of the pixel value, and then that value alternates with the original pixel value as inputs to the array. As such, the integral and squared integral image are computed in alternate cycles with the entire computation taking 2 * [(*m* + (*m*-1) + (*n*-1)] cycles, for an input image of *n* rows by *m* columns.

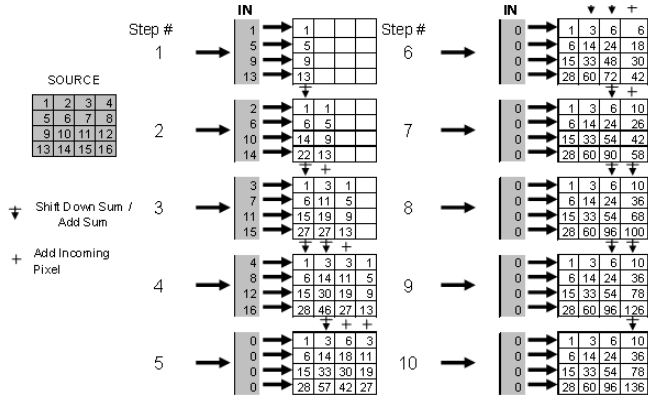


Fig. 9. Illustration of the Integral Image Computation and Data Movement

The rectangle computation takes place next. For each rectangle in a feature, each corner point is shifted towards the CCU acting as collection point. The points move one at a time, but in parallel for all rectangles in the array. At each collection point, the point is either added or subtracted to the accumulated rectangle sum, with the final rectangle value computed when all points of each rectangle arrive at the collection point. As such, each point requires *dx+dy* cycles to reach the collection point, where *dx* and *dy* are the offset coordinates of the point with respect to the upper left corner of the search window. When all rectangle sums for a single feature have been collected in the CCU that represents the starting corner for each feature, they are then shifted leftwards, towards the EUs, one sum at a time per EU. From left to right, eventually all sums arrive in each EU, where the rectangle weights are multiplied with the incoming sums, in order to evaluate the feature. It must be noted that each CCU contains the rectangle sums, the accumulated feature sum from the previous feature computation and the variance. Hence each CCU takes *n+2* cycles to shift the data to its neighboring CCU, where *n* equals the number of rectangle sums per feature. When each rectangle sum enters the EU, it is multiplied with the respective rectangle weight given by the training set, and accumulated together to compute the feature sum. The compensated threshold is then computed using the original threshold and the variance as described earlier. The feature sum is then squared using the multiplier, and compared to the compensated threshold to set the feature result. The partial stage sum is accumulated with the feature result and shifted with the flag bit in a toroidal fashion to the CCU on the far right of the grid, to continue the computation. Eventually, when all feature results are computed, they are stored back into the CCUs in the grid and the computation resumes with the next feature. The computation is illustrated in Fig. 10.

At the end of a stage, the computed stage sum is compared against the stage threshold obtained from the training set. Depending on the outcome of the comparison, the location of the CCU is flagged as an object of interest candidate and continues further evaluation, or is discarded by resetting the flag bit that is shifted with the stage sum. When a location which does not contain an object of interest arrives for computation at the EUs, the EU does not compute the feature sum, rather remains idle, and simply propagates the data to the

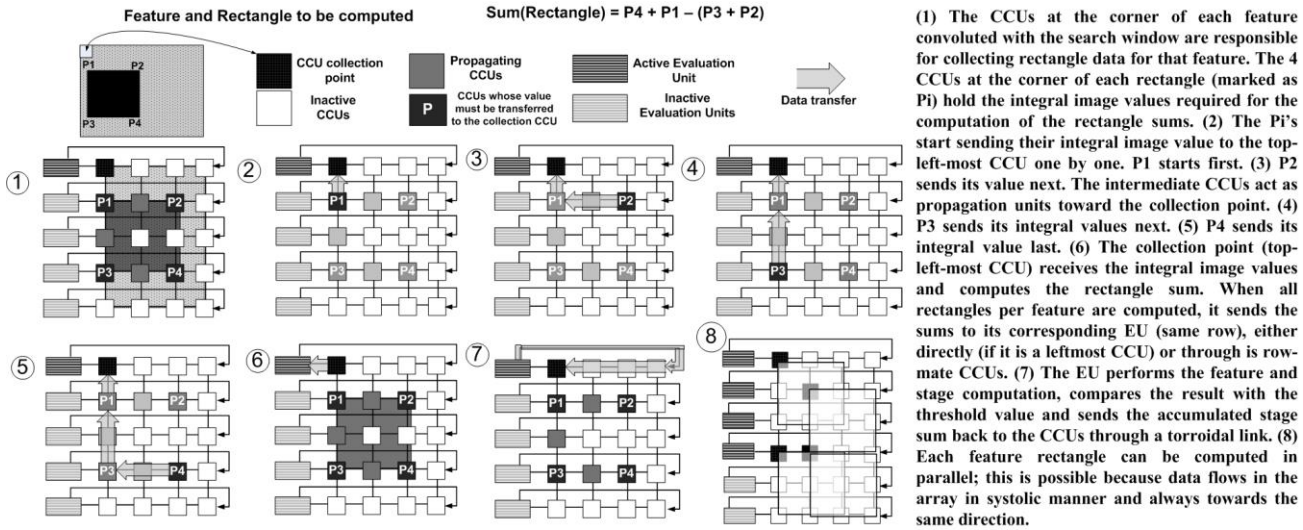


Fig. 10. Systolic Array Computation Flow

far right CCU to resume computation. The CCUs that do not act as collection/computation points, (only hold integral image info as mentioned earlier) simply propagate their values in order to maintain temporal consistency.

When all stages complete for a certain feature scale inside the search window, the flagged locations correspond to the ones that contain the object of interest. If the feature computed is the last one, the computation ends. Each location that contains an object of interest is shifted to the right and outside of the array, for the host application to proceed.

This parallel approach yields several advantages. First, it does not require the training set data to be stored on-chip, as it only computes one feature at a time. Instead, it operates on the image data in parallel. Second, the systolic implementation returns predictable and fast operation, in the context of high frequency. Third, computations that are expensive in terms of delay and hardware overhead such as multiplication are isolated and computed together in parallel during each evaluation step, thus amortizing their delay towards the whole operation. Fourth, in contrast to the software implementation of the AdaBoost algorithm, the detection rate remains constant regardless of the number of objects of interest that exist in a single frame, whether they are detected positively or negatively (i.e. false positives). The software implementation suffers when the amount of object increases, as the algorithm will have to follow the entire classifier cascade multiple times. In contrast, the proposed architecture searches and performs the cascade only once for the entire image, rather than for each object, as done in software. Lastly, when used in conjunction with an IPG process, it can be scaled to the application's requirements and available budget, as the array size can vary from being equal to the size of the training feature to as much as the budget and performance requirements allow and demand.

## V. Evaluation and Results

To evaluate the proposed architecture, we followed two major steps. First, we designed and verified the architecture using FPGA emulation. Second, we performed a full functional simulation using an ASIC implementation over a commercial CMOS library, with three different object detection applications used as benchmarks. Both systems were designed with emphasis on the corresponding hardware constraints, and evaluation was performed taking into consideration several design constraints and limitations. Prior to detailing the simulation details, we first discuss the concept of performance under an object detection system, and list the factors, which explicitly impact the performance of the system.

### *A. Performance metrics, limitations and constraints*

There are two important performance metrics in object detection, the detection frame rate which defines the ability of the system to process a number of input image frames per second (fps), and the detection accuracy, which defines the effectiveness of the system in detecting the object(s) of interest. For real-time video processing, the system needs to detect at least 30fps (NTSC). However, if other image processing and recognition algorithms have to co-exist with detection, the system needs to be much faster, which is typically the case. Moreover, the system's accuracy largely depends on the training, and partially on the way that the training set is represented when implemented in hardware. In designing our architecture, we took into consideration several performance metrics, limitations and constraints, which are outlined in this section.

Firstly, the training set size, particularly the number of features and stages in the training set, largely impacts the performance. As each feature is processed in parallel, the algorithm depends on the total number of features and stages to return a positive result. Training set optimization can

TABLE II
RELATED WORK IMPLEMENTATION ON FPGAS RESULTS COMPARISON

| | | Hiromoto [9] | Cho [10][a] | Wei [11] | Shi [12] [b] | Lai [28] [c] | Presented Work |
|---|---|---|---|---|---|---|---|
| **FPGA** | | XC5VLX330-2 | XC5VLX110 | XC2V2000 | *Not Applicable* | XC2VP30 | XC2VP30 |
| **Frames per Second** | | 30 | 26 (320 x 240)[a] | 15 | 102 | 143(126 Hz) [c] | 64 |
| **Area (Used/Total)** | **Slice LUTs** | 63,443/207,360 | 66,851/69,120 | 13,853/21,504 | *Not Applicable* | 20,901/27,392 | 25,818/27,392 |
| | **Slice Registers** | 55,515/207,360 | 21,902/69120 | 4,573/21,504 | *Not Applicable* | 7,782/27,392 | 23,744/27,392 |
| | **Multipliers** | Not Provided | Not Provided | 28/56 | *Not Applicable* | Not Provided | 68/136 |
| **Memory (Block RAMs)** | | Not Provided | 41/128 | 56/56 | *Not Applicable* | 44/136 | 24/136 |
| **Clock Frequency (MHz)** | | 160.9 | Not Provided | 91 | 200 | 126 | 100 |
| **Face Detection Accuracy** | | Not Provided | Not Provided | 85% non-faces, 50% faces | Not Provided | 86% on faces | 93% (overall) |

[a]Using three classification modules, [b]Implementation of a cycle accurate simulator, [c]Using only 52 features and 1 stage

improve the performance by maintaining a high accuracy in the detection while reducing the number of features. This was done in [28]; however, no detailed discussions were given related to the accuracy of the detector, especially the false positives. In our implementation the architecture is developed independent of the training set size, taking advantage of potential emerging research that reduces training set data.

The second factor impacting the performance, which in our case affects the performance heavily, is the size of the search window and the size of the array. As features are enlarged and computation is repeated to detect bigger objects, the number of enlargements necessary is defined by the search window size (i.e. until the size of the enlarged feature meets the search window size). Consequently, a large search window size will result in computation over several feature sizes. This increases the amount of computations and limits the performance. A small search window on the other hand limits the amount of feature enlargements and results in a larger number of search windows generated for each input image frame. However, the number of generated search windows increases exponentially as the input image frame size increases, and potentially results in loss of data due to several downscaling iterations. We consider this scenario in our approach, thus we combine the IPG and the systolic array to provide flexibility in selecting an appropriate search window size as the application demands, depending on the performance and cost requirements. The nature of the application, such as the amount of training data required, the feature size, the size of the object(s) of interest, input image size and number of objects concurrently appearing on the input image are all factors that play a role in determining a good ratio of the IPG to the search window size (and subsequently the systolic array size). A system-level optimization framework can potentially be used to determine these sizes for various applications, and is left as future work.

The third factor that impacts the performance is the input image frame size. Obviously a large frame results in more data to be explored and a larger number of search windows, but it also impacts the size of objects relative to the input image frame. Large-sized objects typically result in wasted computations when using small-sized features, whereas small objects result in wasted computations when using large-sized features. This is even worst when two or more objects of interest appear in different sizes; the largest the variance in the sizes of the objects, the larger the number of feature and stage computations overall. The proposed architecture is ideal for large image sizes, as the high degree of parallelism can process large images in parallel, resulting in satisfactory frame rates. We used three image sizes in our simulations, and noticed a relatively small decline in the frame rate when going from a small to a much larger image frame size.

The fourth factor relates to the object of interest itself, and the targeted video application. In particular, the amount and size of objects of interest contained in a single frame plays a dominant role in the overall performance, especially in sequential software implementations. The big advantage of the AdaBoost algorithm, which results in large detection frame rates, lies in the ability of the algorithm to reject several search windows which do not pass certain thresholds during an early stage in the computation. However, if the amount of objects of interest in an image frame is relatively large, the algorithm slows down significantly, as it will have to go through the full computation several times. Our architecture however, is independent of the number of objects; as the entire search window is explored in parallel, the time required to search for a single object, is the same time as the time required to search for all objects in the search window. Furthermore, when two or more objects of interest of different sizes are present in the source image, detection will occur at different feature scales. A worst case scenario would be at for least one object of interest inside the search window, in every scale where a feature is evaluated. In reality however, this is a highly improbable scenario; a large object will usually cover smaller objects in an image. Furthermore, there are cases where objects of interest are not present in an image frame; the search windows will likely fail somewhere through the first few stages for all search windows at all feature sizes, thus enabling a new image frame to be processed. In such cases, the frame rate obviously increases. Additionally, changes within a video signal (i.e. new objects of interest entering the image frame or other objects leaving) typically happen within a few frames apart. Hence, consecutive frames are usually similar to each other. This of course implies that a lower frame rate than the video frame rate could be sufficient; however, in the likely scenario that object detection is part of a chain of operations that have to meet real time video processing, the detector still has to operate as fast as possible. Consequently, to conclusively evaluate any architecture, one has to choose a sequence of test frames containing a number of objects, of different sizes, taking these observations into consideration.

The last factor obviously lies on the hardware itself, most notably the operating frequency. In designing our architecture, we took aim in using a regular, modular approach, with small critical paths. The CCU contains minimal hardware, with a fast carry-look-ahead adder. The EUs, which take more cycles,

and burden potential delays in memory accesses and multiplications, only operate during certain time intervals (i.e. during each feature and stage evaluation and I/O operation), allowing the bulk of the computation (i.e. rectangle collection and summation) to the much faster CCUs.

### B. FPGA Implementation and Emulation

As proof of concept, we designed an experimental version of the proposed architecture targeting the Xilinx XUP Virtex II Pro platform [27]. The FPGA evaluation targets a face detection application, using the training set and parameters given with the Open Computer Vision (CV) library [18]. The CV library provides a state-of-the-art software implementation of the AdaBoost detector, utilizing a very accurate pool of features. The training set uses a starting feature size of 24x24 pixels, and scales each feature by a factor of 1.25 (taking the ceiling of the result), resulting in 5 scaled feature sizes (24x24, 30x30, 38x38, 48x48 and 60x60). Each feature has between 2 to 4 rectangles. The training set consists of 2,913 features in 25 stages. The number of features per stage range from 9 to 211, and the total number of rectangles is 6,383.

The training set is organized on a feature by feature basis. Each feature data includes the feature sequence number; the number of rectangles in the feature; the *dx* and *dy* offset values for each rectangle in the feature; the weight associated with each rectangle; and the squared threshold value for each feature. Additionally the stage data includes a certain threshold per stage. To represent the training set, we use 8 bits per rectangle weight, for each threshold value and for each predetermined feature sum, using signed fixed point numbers of 2 integer bits and 5 decimal bits. The dynamic range supported is *+/- 3.96875*, close to the required accuracy for the *OpenCV* training set. The external memory that holds the training set, holds also the upscaled feature data, that is rectangle offsets and weights. We use 6 bits to store each rectangle offset, as the largest feature size we utilize is 60x60 (to fit the 80x60 array). Each rectangle needs to store up to 4 *dx* and *dy* values. The training set was stored in the off-chip (on-board) memory, as features and stage data are used only once every array collection and computation. As already mentioned, we store the upscaled feature data in off-chip memory as well, as when features are enlarged, new rectangle offsets are used. This however is of minor importance, as the offsets can simply be scaled on-chip, dynamically, since the feature training set is loaded feature by feature. For simplicity purposes, we stored the rectangle offsets for all feature sizes in off-chip memory, as part of the training set.

In designing the CCUs, we need to provide storage for the case where all pixels will have an intensity value of 255, an unlikely scenario, but necessary for correct operation. Thus, the maximum integer value that can be stored in an integral image is 255x80x60 and the maximum integer value that can be stored in an integral squared image is $(255)^2$x80x60. This requires 21 bits and 29 bits respectively. Knowing these requirements, we designed the architecture using 80x60 CCTUs, 60 MEUs and 4 CUs. Each CCU connects to its neighbors through an 8 bit bus, which however can increase to a larger size if necessary for bandwidth purposes. The platform contains external memory (DRAM), which was used to store input image frames and the training set. We used landscape grayscale images of size 320x240 pixels, and an array size of 80x60 cells (60 rows by 80 columns), the largest size that could fit on the targeted FPGA, maintaining the 4:3 ratio of the initial image frame). The IPG receives 8 pixels per clock cycle (the DRAM I/O bandwidth), and generates 80x60 sized search windows, at a pixel offset of five pixels (i.e. every search window starts five pixels after the previous). The IPG also downscales the image by scale factors of 0.75 and 0.5, creating three downscaled images of sizes 240x180, 160x120 and 80x60. The generated downscaled images are stored on the FPGA Block RAM. The number of downscaled images is parametrizable; the scale factor is simply stored in a register, and scaling is done by matrix multiplication. The IPG uses two 80x60 frame buffers, generating search windows in lockstep fashion (i.e. it generates the first, and then proceeds to generate the second while the systolic array processes the first one, with both the IPG and the array alternating between each buffer). FPGA synthesis and utilization results are shown in Table III. The system, which system operates at 100MHz, was verified and evaluated using the application of face detection, through a sample of 300 test images which contained several faces of different sizes, obtained through the World Wide Web, and sized and formatted to the design requirements. The test images were stored in a Compact Flash card during the system initialization stage, and then loaded on the DRAM prior to running the detection framework. The frames were input to the detector, which processed them. A custom VGA controller was then designed and used in order to output the result of the detector to a VGA monitor, for visual verification, along with markings on where the candidate faces were detected. A diagram of the FPGA prototype and a photo of the experimental system are shown in Fig. 11. The system was designed to operate in two modes, verification where image frames were displayed one at a time (for verification and debugging), and runtime, processing all input images continuously, for measuring the detection frame rate, using a stop-watch timer. As said earlier, the frame rate depends on several factors, some of which are independent of the architecture. The system processed all 300 test images in

TABLE III
SYNTHESIS RESULTS FOR THE VIRTEX II PRO FPGA IMPLEMENTATION

| FPGA Resources | Slices (13696) | Flip Flops (27392) | LUTs (27392) | BlockRAM (312.5kB) | Multipliers (136) |
|---|---|---|---|---|---|
| *IPG* | 2,248 (16%) | 2,101 (8%) | 2,445 (9%) | 52.8kB (17%) | 8 (5%) |
| *Array (80x60)* | 10918 (80%) | 20185 (74%) | 22706 (83%) | 0 (0%) | 60 (44%) |
| *System* | 13455 (98%) | 23744 (87%) | 25118 (92%) | 52.8kB (17%) | 68 (50%) |

TABLE IV
DETECTION APPLICATIONS TRAINING DATA

| Detection Application | Object Per Frame | Feature Size (pixels) | # of Rectangles per feature | # of Stages | Total # of Features | Detection Accuracy |
|---|---|---|---|---|---|---|
| Face | 1-7 | 24x24 | 2 to 4 | 25 | 2913 | 93% |
| Road Sign | 1-2 | 12x12 | 2 to 3 | 12 | 414 | 83% |
| Vehicle | 2-10 | 24x36 | 2 to 4 | 20 | 1715 | 78% |

4.68 seconds, an estimated rate of 64 frames per second, which for the type and frequency of the FPGA is relatively high, especially when compared to existing implementations (almost twice as fast). Additionally, the FPGA implementation achieved 96% accuracy detecting the faces on the images when compared to the corresponding Open CV software implementation, running on the same test images. This discrepancy can be justified to the fact that the Open CV implementation only scales the features up (no image downscaling) which does not result in data loss. Additionally, some training data was not able to be represented within the dynamic range employed by the hardware design. Table II presents a comparison table between existing FPGA implementations along with their characteristics, and the proposed architecture implemented on FPGA, for the application of face detection. As seen in the table, the proposed architecture is significantly faster and more accurate than most implementations. The implementation in [12] was based on pure cycle-accurate simulation rather than implementation, and the clock frequency is twice as the one used in our implementation (which was limited by the capabilities of our FPGA board).The work in [28] yields a reported 143 fps, but uses a much smaller training set (two orders of magnitude smaller than the one in OpenCV) in order to achieve such a high frame rate. Furthermore, the reported accuracy focuses on a very specific data set, and no detailed discussion is provided on the rate of false positives. We did not optimize the training set, as it extends beyond the scope of this work.

### *C. ASIC Implementation and Evaluation*

In addition to the FPGA emulation, we also designed a larger system (that could not fit on large existing FPGAs), targeting an ASIC implementation. The objective of the ASIC implementation was to obtain experimental insights on the scalability and feasibility of the proposed architecture, towards large scale integration. We evaluated the ASIC implementation using three object detection applications; face detection, road sign detection [5], and vehicle detection [1], [2]. We focus only on the rear of the vehicle as the point of detection. The training set used in the face detection application was the same one used in our FPGA prototype. For the other two applications, we used Open CV and Matlab to construct a training set for each case, using sample images obtained from the World Wide Web. Our objective was not to construct an accurate training set per se, rather than a realistic one to be used as an experimental set. The training sets were constructed using road sign and vehicle images, and training set details for each application (including the face detection) are given in Table IV. We targeted input images of four sizes (1024x768, 800x600, 640x480 and 320x240), again obtained through the World Wide Web, containing several faces, road signs and vehicles, depending on the targeted application. We then proceeded to design and implement an architecture which could receive as input at least a 1024x768 grayscale image, and process it as fast as possible, using the training sets mentioned. It must be noted that each application differs from each other in the context of their training sets (and feature sizes); the underlying hardware architecture is the same for all the targeted applications, as well as input image sizes and formats.

The experimental platform was designed using search windows of size 320x240 pixels. Consequently, the size of the array was set to be the same, consisting of 320x240 CCUs and 240 EUs. The IPG was designed with two search window buffers, producing search windows in similar fashion to the FPGA implementation. The original input image size was scaled down using a scale factor of 0.75, and the features were scaled up using a scale factor of 1.33. The training set, downscaled images from the IPG and input image frames were modeled as external memory; everything else was considered on-chip. Additionally, all parameters outlined in the FPGA implementation were modified to reflect the new search window size (such as storage considerations for the integral and integral squared images, data bus between CCUs and EUs, etc.).

The system was synthesized with using Synopsys Design Compiler targeting a commercial TSMC 65nm CMOS library, in order to obtain relevant metrics such as area, operating frequency and power consumption. We used the default library values, and Synopsys' synthesis primitives (focused on area optimization over performance), as well as components from Synopsys Designware IP library. Pre-layout results indicated that the critical path in the system was identified in the EU multiplier (a 64-bit multiplier). We used an 8-stage multiplier from Designware IP library in our design to target high frequency. It must be noted that the synthesized design does not consider the IPG memory modules; we used the CACTI toolset [30] to obtain the potential operating frequency for the two IPG memory modules, estimated at 800 MHz. As such, we set the targeted frequency of the entire system to 800MHz. The post-synthesis, pre-layout results also indicate an area estimation of roughly 88 million transistors.

Preliminary results also were collected for some indicative power consumption merits using 1V power supply voltage and 50% probability of switching activity on all lines. Prior to reporting the obtained power consumption results however, we must state that the overall power consumption depends on several factors not related only to the architecture. The power consumption depends on the input image size and subsequently the number of downscaled images produced, the number of search windows, the number of features in the training set and the number of necessary computations. The latter is determined by the number of objects of interest found in the input frame. Obviously the chosen operating frequency and power supply of the system are important as well. Consequently, power comparison with architectures found in literature is not suitable without the use of the same input data sets and input image sizes. Hence, instead of reporting only the total power consumption for one frame, we also analyze how this power is consumed throughout the computation.

To compute and evaluate a 24x24 feature (rectangle collection and computation, propagation to the EUs, computation and evaluation in the EUs, and propagation of the feature sum back to the array), the system consumes 0.06mW. The IPG unit also consumes ~0.014mW to downscale a 320x240 image to a 240x180 image and 0.0023 mW to produce one 80x60 search window. Overall, to compute a single 320x240 frame with one object of interest (human face), the unit consumes ~2.45mW of power. We did not consider any power optimization mechanisms other than not computing features when a region was marked as a non-candidate, and overall, our focus was not on optimizing the architecture for power savings. Power optimizations are left as future work.

Using text files that contained the input image files and training set data, we next proceeded to run functional RTL simulation of the system using Modelsim. We run a set of 10 test images per image size per application (i.e. 40 test images per each application), and obtained the total number of cycles required to process each test case. The resulting frames were stored as text files, and reconstructed to images using Matlab, so that we could visually verify the results. Using the obtained clock frequency from the synthesis results, we then estimated the detection frame rate (as well as the detection accuracy when compared to the corresponding software implementation). Table V summarizes the results for each application, under the four input image frame sizes, and Fig. 12 shows some resulting frames from the simulation.

Table VI presents a summary of the synthesis results, and a brief comparison with the special-purpose vision processor presented in [29]. When comparing equal sized input images, the frame rate achieved by the proposed architecture is significantly larger. The associated power consumption and hardware overhead costs cannot be compared, however, the overall simulation results indicate that the proposed architecture can be scaled to significant sizes, and potentially be used in high-performance applications with large input image sizes, or can be designed to consume minimal energy and hardware overheads for small-scale embedded systems.

### *D. Discussion*

Both the FPGA prototype as well as the large scale ASIC implementation have shown great potential for applications with real-time performance requirements, such as real time object detection in vehicular embedded and applications involving multiple camera streams. The system is particularly useful in monitoring populated areas such as airports and transportation terminals, where it can process frames from alternate video streams, regardless of the amount and size of objects found in the input image frames. The scalability of the system and its independence from the training set also provide flexibility to the designer, allowing the designer to determine the most efficient size of the system directly from the application requirements. By merging the IPG with the feature upscaling originally used, the system achieves a fully parametrizable performance-to-cost ratio; if the silicon budget allows it, an increase in the array size will boost the performance (by increasing the degree of parallelism). On the other hand, a smaller array, while slower, costs less and can still satisfy certain performance requirements.

There are some useful conclusions extracted from our simulations with respect to the algorithm. The FPGA implementation shows that the architecture can scale well in smaller, less demanding environments, while maintaining reasonable frame rates. The ASIC implementation on the other hard illustrates the full-throttle operation of the detector, and its suitability for multiple video streams and detection of objects that could appear in different numbers and sizes within an input image frame. Obviously, depending on the budget

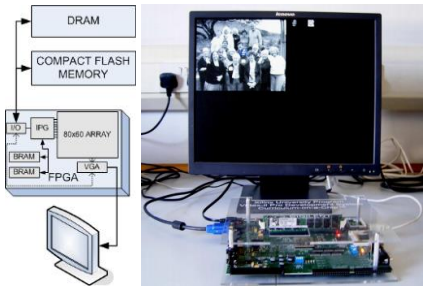


Fig. 11. Experimental Platform – Block Diagram (left) and photo of the experimental platform, using the Virtex II Pro FPGA (right).

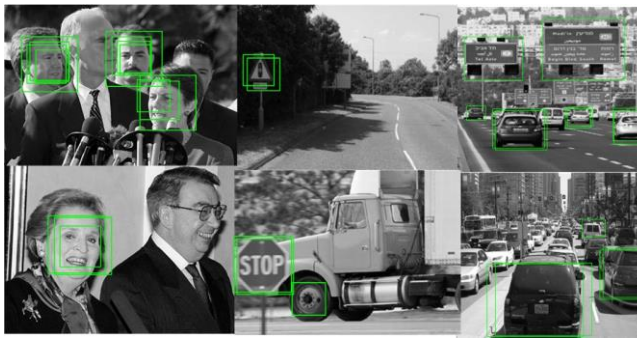


Fig. 12. Output image frames with the detection frames placed on the detected objects of interest.

TABLE V
ASIC IMPLEMENTATION – SIMULATION RESULTS

| Application /Accuracy | Input Image Size | Time to process 10 frames (seconds) | Projected Frame Rate |
|---|---|---|---|
| Face Detection 95- 96 % | 1024x768 | 0.109 | ~91 |
| | 800x600 | 0.098 | ~102 |
| | 640x480 | 0.084 | ~118 |
| | 320x240 | 0.075 | ~133 |
| Road Sign Detection 92- 97 % | 1024x768 | 0.099 | ~101 |
| | 800x600 | 0.093 | ~107 |
| | 640x480 | 0.083 | ~120 |
| | 320x240 | 0.072 | ~139 |
| Vehicle Detection 91- 96 % | 1024x768 | 0.128 | ~78 |
| | 800x600 | 0.116 | ~86 |
| | 640x480 | 0.108 | ~93 |
| | 320x240 | 0.098 | ~102 |

TABLE VI
RELATED WORK IMPLEMENTATION ON ASICS RESULTS COMPARISON

| | **Presented Work** | **Hanai [29]** |
|---|---|---|
| **Technology** | TSMC 65nm CMOS | 90 nm CMOS |
| **FPS (320x240)** | ~133 | 8 |
| **Area (# of transistors)** | 88 million | 2.1 million |
| **Power (mW)** | 2.45 per 320x240 frame | 0.47/fps – 3.72 total |
| **Clock Frequency** | 800 MHz | 54 Mhz |
| **Accuracy** | 95% | 81% |

and application constraints, the designer can select the type of implementation that satisfies the operating conditions and application specifications.

## VI. Conclusion

Object detection is an important step in multiple applications related to computer vision and image processing, and real-time detection is critical in several domains. In this paper, we presented a flexible, parallel architecture for implementation of the AdaBoost object detection algorithm. The architecture combines an image pyramid generation process, along with highly parallel systolic computation, to offer a flexible design that is suitable for several types of applications and budgets. The paper presented two experimental platforms of the architecture, a low-end FPGA implementation and a high-end ASIC implementation, both of which achieved significantly high detection frame rates and accuracy, comparable to their budget, illustrating the flexibility and potential of the architecture.

We anticipate that further optimizations in terms of power consumption will significantly improve the architecture, leaving this as immediate future work. We also plan on exploring system-level optimization algorithms, of determining a systolic array size that best satisfies the performance/cost requirements. Additionally, we plan to include an embedded processor so that the architecture can potentially support on-line training, making it capable for dynamic, autonomous environments and situations. Furthermore, we hope that this architecture will lead to improvements in existing training sets, taking into consideration hardware constraints when training a detector. We also hope that this architecture will be combined with other on-chip implementations of related applications to form a complete high-performance embedded computer vision and image processing hardware platform.

## References


[1] L. Dlagnekov and S.Belongie, "Recognizing Cars," *UCSD CSE Tech Report, no. CS2005-083*, 2005.

[2] F. Moutarde, B. Stanciulescu, and A. Breheret, "Real-time visual detection of vehicles and pedestrians with new efficient adaBoost features," in the *Proc of Workshop on Planning, Perception and Navigation for Intelligent Vehicles (PPNIV) of 2008 International Conference on Intelligent Robots and Systems (IROS'2008)*, Nice, France, 26 Sept. 2008.

[3] Xusheng Tang, Zongying Ou, Tieming Su, Pengfei Zhao, "Cascade AdaBoost Classifiers with Stage Features Optimization for Cellular Phone Embedded Face Detection System," *ICNC*, pp. 688-697, 2005.

[4] Y. Abramson and B. Steux, "Hardware-friendly detection of pedestrians from an on-board camera," in *Proceedings of the IEEE Intelligent Vehicle Symposium (IV'04)*, Parma, Italy, June 2004.

[5] Changyong Yoon, Minkyu Cheon, Euntai Kim, Mignon Park, Heejin Lee, "Real-time road sign detection using Adaboost and Multicandidate," *Proceedings Of The 8th Symposium On Advanced Intelligent Systems ISIS 2007*, 2007, pp 953-956.

[6] P.Viola and M.Jones,"Real-time object detection," *Int. J. Comput. Vis*., vol. 57, no 2, pp 137-154, May 2004.

[7] Y. Freund and R. E. Schapire, "A Short Introduction to Boosting," *Journal of Japanese Society for Artificial Intelligence*, vol. 14, no. 5, pp. 771-780, 1999.

[8] T. Theocharides, N. Vijaykrishnan, M. J. Irwin. "A Parallel Architecture for Hardware Face Detection," *Proc. Of the IEEE Computer Society Annual Symposium on VLSI Design, ISVLSI'06*, Karlsruhe, Germany, pp. 452-453.

[9] M. Hiromoto, H. Sugano, and R. Miyamoto, "Partially Parallel Architecture for Adaboost-Based Detection with Haar-Like Features," *IEEE Trans. on Circuits and Systems for Video Technology*, Vol.19, No.1, Jan. 2009, pp.41-52.

[10] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fpga-based face detection system using haar classifiers," *Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, New York, NY, USA: ACM, 2009, pp. 103-112.

[11] Y. Wei, X. Bing, and C. Chareonsak, "Fpga implementation of adaboost algorithm for detection of face biometrics," in *Biomedical Circuits and Systems, 2004 IEEE International Workshop on*, 2004, pp. S1/6-17-20.

[12] Y. Shi, F. Zhao, and Z. Zhang, "Hardware implementation of adaboost algorithm and verification," *22nd International Conference on Advanced Information Networking and Applications - Workshops, AINAW 2008*, 2008, pp. 343-346.

[13] Schapire, R. E., "The boosting approach to machine learning: An overview," In *MSRI Workshop on Nonlinear Estimation and Classification*, 2002, pp. 1134-227.

[14] T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin, W. Wolf, "Embedded Hardware Face Detection," *In the proceedings of the 17th International Conference on VLSI Design*, Mumbai, India, January 2004.

[15] N.Ranganathan, "VLSI Algorithms and Architectures," IEEE Computer Society Press, Los Alamitos, California, USA, 1993.

[16] R. McCready, "Real-Time Face Detection on a Configurable Hardware System," *International Symposium on Field Programmable Gate Arrays*, 2000, Monterey, California, United States.

[17] Erik Hjelmås, Boon Kee Low, "Face Detection: A Survey," *Computer Vision and Image Understanding*, Vol. 83, No. 3, September 2001, pp.236-274.

[18] Open Source Computer Vision Library, http://www.intel.com/technology/computing/opencv/index.htm and http://sourceforge.net/projects/opencvlibrary/files/, June 2009.

[19] A. Price, J. Pyke, D. Ashiri and T. Cornall, "Real time object detection for an unmanned aerial vehicle using an FPGA based vision system," *in* the *Proc. of the 2006 IEEE International Conference on Robotics and Automation*, May 2006, pp. 2854-2859.

[20] P. Wieslaw, "Vehicle Detection Algorithm for FPGA Based Implementation," in *Advances in Soft Computing, Computer Recognition System 3*, Vol. 57/2009, 2009, Springer Berlin, pp. 585-592.

[21] M. Kolsch and M. Turk, "Robust hand detection," in the Proc. of the International Conference on Automatic Face and Gesture Recognition, Seoul, Korea, 2004, pp. 614-619.

[22] S. Mahlknecht, R. Oberhammer and G. Novak," A real-time image recognition system for tiny autonomous mobile robots," *in the Proc. of the 10th IEEE Symposium of Real-Time and Embedded Technology and Applications*, May 2004, pp. 324-330.

[23] V. Lohweg, C. Diederichs and D. Muller, "Algorithms for Hardware-Based Pattern Recognition," *EURASIP Journal on Applied Signal Processing*, Vol.12, 2003, pp. 1912-1920.

[24] K. Khattab, J. Dubois and J. Miteran, "Cascade Boosting Based Object Detection from High Level Description to Hardware Implementation," *EURASIP Journal on Embedded Systems*, vol. 2009, 2009, pp. 1687-3955.

[25] Y. Ming-Hsuan, D.J. Kriegman, N. Ahuja, "Detecting Faces in Images: A Survey," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 24, No1., January 2002, pp. 34-58.

[26] C. P. Papageorgiou, M. Oren, T. Poggio, "A General Framework for Object Detection," *in* the *Proc. of the Sixth International Conference on Computer Vision (ICCV'98)*, 1998, pp. 555-562.

[27] Xilinx University Program, http://www.xilinx.com/univ/, Jan. 2009.

[28] H.-C. Lai, M. Savvides, and T. Chen, "Proposed FPGA hardware architecture for high frame rate (>100 fps) face detection using feature cascade classifiers," *First IEEE International Conference on Biometrics: Theory, Applications, and Systems, Sept.* 2007, pp. 1-6.

[29] Y. Hanai, Y. Hori, J. Nishimura and T. Kuroda, "A Versatile Recognition Processor Employing Haar-Like Feature and Cascaded Classifier," *ISSCC'09*, Dig. Tech. Papers, pp.148-149, Feb. 2009.

[30] The CACTI Toolset http://research.compaq.com/wrl/people/jouppi/CACTI.html