Scalable Object Detection Accelerators on FPGAs Using Custom Design Space Exploration

Chen Huang and Frank Vahid*
Dept. of Computer Science and Engineering
University of California, Riverside, USA
{chuang,vahid}@cs.ucr.edu
*also with the Center for Embedded Computer Systems, Univ. of California, Irvine

***ABSTRACT*- We discuss FPGA implementations of object (such as face) detectors in video streams using the accurate Haar-feature based algorithm. Rather than creating one implementation for one FPGA, we develop a method to generate a series of implementations that have different size and performance to target different FPGA devices. The automatic generation was enabled by custom design space exploration on a particular design problem relating to the communication architecture used to support different numbers of image classifiers. The exploration algorithm uses content information in each feature set to optimize and generate a scalable communication architecture. We generated fully-working implementations for Xilinx Virtex5 LX50T, LX110T, and LX155T FPGA devices, using various amounts of available device capacity, leading to speedups ranging from 0.6x to 25x compared to a 3.0 GHz Pentium 4 desktop machine. Automated generators that include custom design space exploration may become more necessary when creating hardware accelerators intended for use across a wide range of existing and future FPGA devices.**

I. INTRODUCTION

Automated object detection, such as face detection, has been studied extensively in recent decades. Methods have been incorporated into products, such as modern cameras that automatically focus by detecting faces, or video surveillance systems that highlight vehicles. Many future applications need fast accurate object detection, including domains such as human computer interfaces, smart rooms, robot vision, and automobile collision warning systems. Future applications may need the object detection be done in real-time (30-60 frames/second) with a high detection ratio. High accuracy requirements demand sophisticated algorithms such as the Haar-feature based object detection algorithm [13] used in this paper. However, current desktop processor implementations of complex object detection algorithms suffer from low detection speeds and high processor resource utilization. A desktop processor implementation of a Haar-feature based face detection algorithm achieves 3.5 frames/second on a Pentium 4 3.0 GHz machine for a low resolution 320x240 video stream, based on our experiments.

The Haar-feature based object detection algorithm has massive potential parallelism, such as the feature values of each sub-window potentially being calculated in parallel. Likewise, the image scaling process in the algorithm can be executed in parallel with data buffer construction. The algorithm also needs to execute the most computation-intensive task, feature value calculation, iteratively for each sub-window. These computation patterns are highly suitable for FPGA implementation. With custom-designed circuits for object detection in the FPGA, real-time object detection is possible without assistance of a processor.

We implemented the Haar-feature based object detection algorithm on a series of Xilinx Virtex5 FPGAs with different performance and resource requirements. We found the bottleneck preventing the design from being scalable was the communication architecture between the data buffer and the classifiers, as the communication architecture consumed the most FPGA resources. In this paper, we focus on the design of efficient communication architectures for different numbers of classifiers using content information of each feature set. We formulated a feature mapping problem and developed a tool to automatically explore and generate different designs that can fit in FPGAs with different capacities. With growing FPGA capacities and the tremendous variety of possible implementations of an algorithm on FPGAs, such custom design space exploration may become a common requirement for implementing applications on FPGAs.

The paper is organized as follows. Section II reviews previous work on object detection algorithms and custom design space exploration. Section III introduces Haar-feature based object detection. Section IV introduces the FPGA implementation details, and Section V discusses the communication architecture and a custom design space exploration tool. Section VI presents experimental results and Section VII concludes.

II. RELATED WORK

Object detection, especially face detection, has been an active research area for many years. A color-based face detection algorithm was proposed by Hsu [5]. The algorithm extracts image areas with skin color. Rowley [11] introduced a neural network based face detection algorithm that can detect rotated faces. Viola and Jones [13] introduced a Haar-feature based object detection algorithm, which has been implemented in Intel's OpenCV [9] image processing library. Our work uses Viola and Jones's algorithm for the FPGA implementation.

Many efforts implement different face/object detection algorithms on FPGAs. Theocharides [12] implemented a neural network based algorithm for face detection. Their implementation was able to detect rotated faces and achieved 75% accuracy. Wei [14] implemented an AdaBoost [2]

algorithm to detect face biometrics on images with 120x120 pixels. Gao [3] re-trained the Haar face features to 16 features per stage, which is more convenient for FPGA implementation. Cho [1] implemented the entire Haar-feature based face detection algorithm on a Xilinx Virtex5 FPGA. They implemented two versions with 1 or 3 classifiers. Their communication architecture between image buffer and classifiers was not designed for scalability. Previous implementations of object detection algorithms on FPGAs focus on implementing one or a few designs for a certain FPGA. However, the design space of the Haar-feature based object detection algorithm is enormous. Our approach formulates a custom exploration problem for the communication architecture aiming to make the object detection accelerator more portable to different FPGAs.

Custom exploration has been used in past work focusing on automatic soft-core generators. Nordin [8] presented a parameterized soft-core generator for the discrete fourier transform (DFT) kernel, which yields implementations over a range of different performance/cost tradeoff points. L'Insalata [7] proposed an environment for automatic generation of fast fourier transform (FFT) and inverse FFT cores, which focused on low circuit complexity. The Xilinx ISE tool [15] provides soft-core generators for on-chip memory, standard bus interfaces, math functions, and much more. These soft-core generators can tune basic parameters such as memory bandwidth and depth. The ISE tool also provides a parameterized MicroBlaze [16] soft-core processor. Previous soft-core generators are often designed for computation kernels or processors with straightforward parameter tuning, while the Haar-feature based object detection in this paper is a more complex application that leads to a larger design space, and gives more optimization opportunities.

III. HAAR-FEATURE BASED OBJECT DETECTION ALGORITHM

A) Algorithm overview

The basic idea of the Haar-feature based object detection algorithm is to detect an object in small sub-windows of an image. For example, to detect a face in an image, the algorithm examines all possible 20x20 sub-windows (called examine windows) in an image, as illustrated in Figure 1.

Suppose the image size is 320x240 pixels and the examine window size is 20x20 pixels. The examine process starts from the top left corner. Then the examine window moves down along the Y axis 1 pixel at a time. When the entire column has been examined, the examine window will move along the X axis by 1 pixel to repeat the process for the next column. The total number of examined windows is (240–20)*(320–20) = 66,000. Desktop implementations often move the examine window 2 pixels at a time; such interleaved scanning is 4 times more efficient, while having a comparable detection quality.

Figure 1: Movement of examine window.

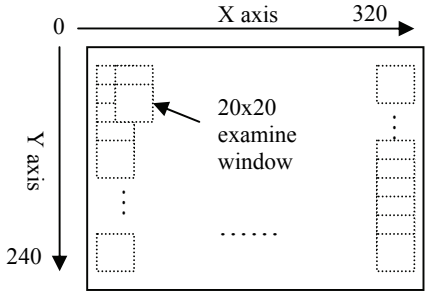
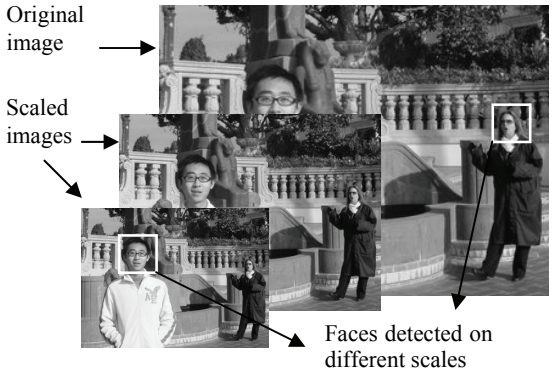


Figure 2: Image scaling example



B) Image scaling

To detect objects of different sizes, the algorithm scales down the image and repeats the search. The image scaling process is illustrated in Figure 2. There are two faces in Figure 2 with different sizes. The 20x20 examine window detects one face in the original image. By scaling down the original image, the algorithm can detect another face with the same 20x20 examine window. The algorithm scales down the image by a constant scale factor until the height or width of the scaled image is smaller than the examine window's width.

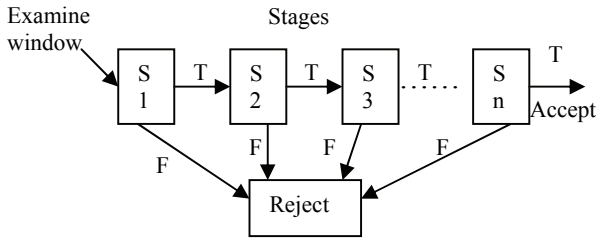
C) Haar-feature and integral image

The most computation intensive part of the detection algorithm is to determine whether the current 20x20 examine window contains an object. The algorithm calculates Haar-feature values to make the decision. Figure 3(a) shows two Haar features within a 20x20 examine window, where each feature contains 2 to 3 small rectangles. Each object type has different Haar-features relating to its shape, e.g., two eye features in a face are shown in Figure 3(a). Details about Haar-features can be found in Viola and Jones [13]. The feature sum equals to the pixel sum (sum of the image pixels' grayscale) in the white rectangles minus the pixel sum of the black rectangles. The feature value is determined by comparing the feature sum to the feature threshold. The feature set and threshold of an object are generated by training a large number of images with the AdaBoost algorithm [2].

Figure 3: Haar-feature and integral image

(a) Haar features (b) integral image

Figure 4: Decision cascade (T=True, F=False)



We utilized an existing Haar-feature set from the OpenCV library [9]. To efficiently calculate the pixel sum of an arbitrary rectangle, the algorithm uses an integral image as an auxiliary data structure. In an integral image, each point stores the pixel sum of a rectangle, starting from the top left corner to this point. With the integral image, calculating the sum of an arbitrary rectangle can be done in constant time, e.g., *Sum(R1) = P4 – P2 – P3 + P1,* as shown in Figure 3(b).

D) Decision cascade

Each object type has a corresponding Haar-feature set of different sizes. For example, the frontal face feature set contains 2135 Haar-features. However, for each sub-window, the object detection algorithm may not need to calculate all feature values, but rather decisions can be cascaded, as illustrated in Figure 4.

The Haar features are divided into several stages. For example, the frontal face has 22 stages. Each stage has 3 to 200 Haar-features. The algorithm calculates the feature value for each feature within one stage and then sums the values to get the *stage sum*. If the stage sum passes the stage threshold, the algorithm continues to the next stage. Otherwise, the algorithm terminates and rejects the current examine window. If an examine window passes all stages, the algorithm accepts the current window meaning that the object is found.

IV. Designed Architecture

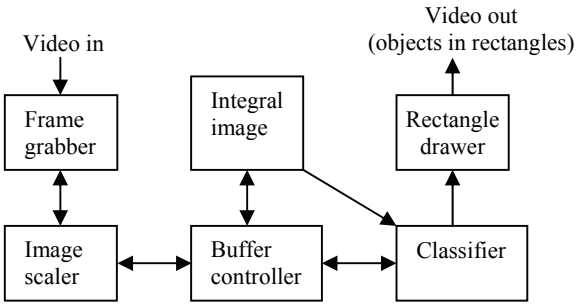
A) Overview

The overall FPGA implementation is illustrated in Figure 5. The frame grabber reads the video input from a standard VGA interface. When an image frame has been grabbed, the image scaler scales down the image by a constant scale factor and notifies the buffer controller when the scale process is done. The buffer controller then constructs an integral image for each examine window in the scaled image. When an integral image has been built, the buffer controller sends a buffer ready signal to the classifier. The classifier reads the values in the integral image and carries out the cascaded decision process. If an object is found, the classifier will notify the rectangle drawer to draw a rectangle around the object.

B) Image scaler

The image scaler scales the image down by a constant scale factor of 1.2. The examine window size is 20x20 pixels and the original image size is 320x240 pixels. The total number of scaling processes is: $Floor(\log_{1.2} 240/20) = 13$. We use the

Figure 5: Overall architecture



bilinear scaling [4] approach in our implementation, which needs 4 memory accesses to the original image per pixel.

C) Integral image

The integral image is the only data source needed by the classifier. The desktop version of the algorithm first calculates and stores the integral image of the entire image for later uses. However, storing the entire integral image is too expensive on an FPGA in terms of both size and performance. Our current design only stores the integral image of the current 20x20 examine window. As the maximum value in the integral image is 20*20*255 (255 for 8-bit grayscale) = 102,000, we use a 20x20 17-bit register file to store the integral image.

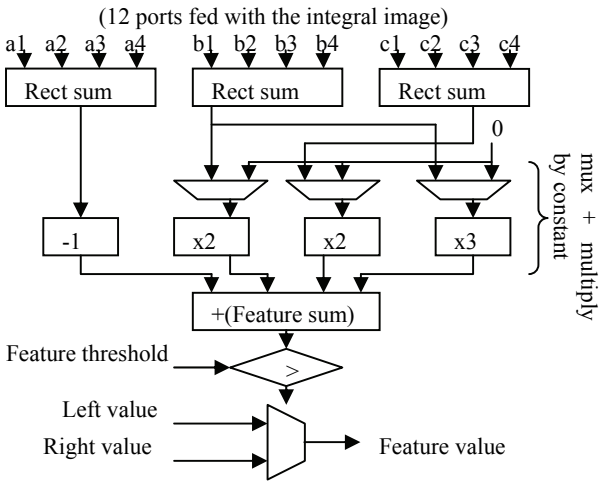
During the object detection process, the examine window moves within the image as in Figure 1. When the examine window moves down by 1 pixel, every entry of the integral image can be updated in parallel. For instance, *Integral image (i, j) = Integral image(i + 1, j) – Integral image(1, j)*, where *i* is the row index and *j* is the column index. To support this parallel updating process, this register file is implemented with LUTs (lookup tables in the FPGA).

D) Classifier design

The classifier reads values from the integral image and calculates Haar-feature values. The Haar feature contains two or three rectangles. A datapath for a classifier is illustrated in Figure 6.

Each rectangle pixel sum can be calculated with 4 values

Figure 6: Classifier datapath



(4 corner pixels) from the integral image. The Rect sum component executes the following computation: *Rect sum(a) = a1 – a2 – a3 + a4*. Each rectangle sum is then multiplied by a weight factor. We noticed that there are only three conditions for the weight factors: (-1, 2, 0), (-1, 2, 2), (-1, 3, 0). We can use muxes and multiply by constant components to replace the expensive integer multiplier or DSP block. Then the feature sum of three rectangles is computed and compared with the feature threshold. If the feature sum is greater than the feature threshold, the classifier will return the Right value. Otherwise, the Left value will be returned. The Left and Right values are predefined parameters of each Haar feature. The classifier can calculate the feature value in one cycle (65 MHz) without pipelining. We thus eliminate pipeline registers to reduce the classifier's size.

The cascade decision process described in Section III(D) can be implemented via a stage sum register. The stage sum register stores the sum of all feature values in that stage. If the stage sum is greater than the stage threshold, the algorithm continues on to the next stage. Otherwise, the algorithm stops and returns a false.

V. COMMUNICATION ARCHITECTURE EXPLORATION

This section describes the communication architecture between the integral image and the classifiers. We will first describe a general communication architecture for a single classifier. A more scalable data specialization communication architecture for multiple classifiers will then be discussed. Finally, we will introduce the Haar-feature mapping problem for multiple classifiers and describe a design space exploration algorithm.

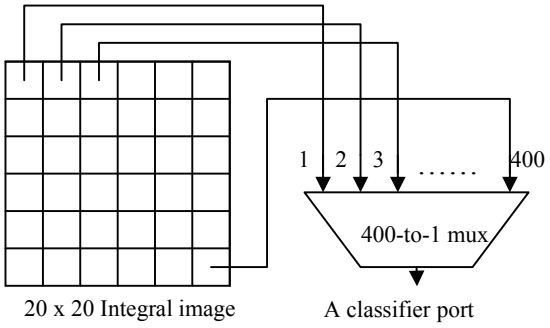
A) General architecture for a single classifier

From the interface between the classifier and integral image shown in Figure 6, note that data should be deliverable from any integral image data entry to any classifier port. Thus, a general communication architecture is shown in Figure 7.

The integral image has 400 entries and each entry of the integral image contains 17 bits. Thus, to access any entry in the integral image matrix, each port of the classifier needs a 17-bit 400-to-1 mux, which consumes about 2300 LUTs in a Xilinx Virtex5 110T FPGA. Since each classifier has 12 ports (Figure 6), one classifier needs twelve 400-to-1 muxes, which takes 27,600 LUTs, or 40% of the total resources of the LX110T FPGA having 69,120 LUTs. For smaller FPGAs that cannot fit all 12 ports, we can put fewer 400-to-1 muxes by using a time multiplexing technique. However, the data bandwidth will decrease with time multiplexing.

Wiring for such muxes is a severe problem. Wire congestion problems worsen when multiple such general communication architectures exist for multiple classifiers accessing the same integral image. As can be seen, simply duplicating this general communication architecture for multiple classifiers is not scalable.

Figure 7: General communication architecture



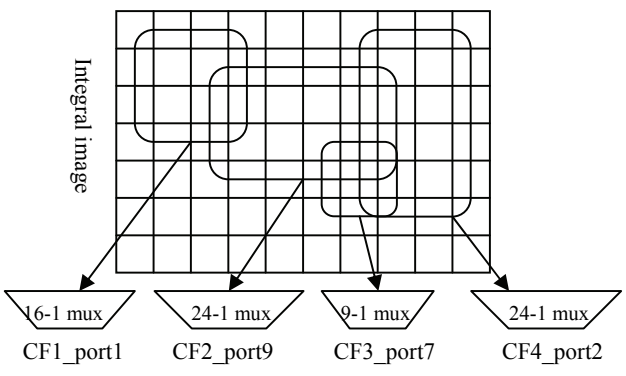
B) Data-specialized architecture for multiple classifiers

Since features within one stage can be calculated in parallel, we can deploy multiple classifiers to increase performance. Instead of using general communication architecture, we can map features to different classifiers. Note that each feature only needs certain entries (feature rectangle location) of the integral image to feed the classifier. By carefully mapping these features to different classifiers, each port of the classifiers may need only a small portion of the entire integral image. In other words, we can group features that access the same integral image entries into the same classifier to make the communication architecture for multiple classifiers more scalable. A feature mapping example is shown in Figure 8.

There are four classifiers in Figure 8, and each classifier is assigned with a number of different features. Note that each classifier's port only needs values from a portion of the integral image. The figure shows each portion as (rounded) rectangles, but a portion can assume any shape. Since each classifier accesses only a portion, each can use a mux smaller than 400-to-1. For instance, classifier1's port1 needs a 16-to-1 mux, while classifier3's port7 needs a 9-to-1 mux. In general, the more classifiers we have, the fewer features are needed to be assigned to each classifier, which leads to smaller muxes and fewer wires. The data-specialized architecture uses content information within each feature (the feature rectangles' location) to optimize the communication architecture, yielding a custom design for a certain feature set. Detecting a different object (such as a vehicle rather than a face) requires redesigning the communication architecture.

The feature storage is also different in the specialized architecture. In the general architecture, the feature rectangle positions are stored in BRAM. In the specialized architecture,

Figure 8: Data specialization architecture



the mux selection values for each step are stored.

C) Custom exploration for feature mapping

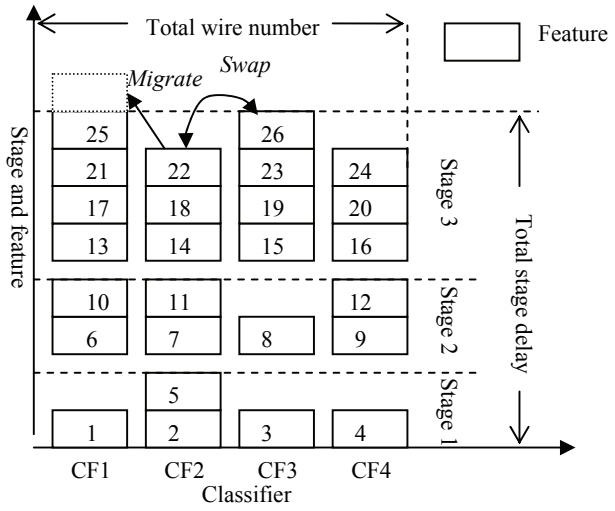
Since the feature mapping determines the communication architecture in the specialized architecture, a good feature mapping can greatly reduce the mux size and number of wires. We developed a custom exploration tool for the feature mapping problem. The tool searches the design space of possible communication architectures for different feature sets. The tool can also be applied for different numbers of classifiers, to tradeoff design size and performance.

The search space of the feature mapping problem is enormous. An object usually has more than 1000 Haar features, which are divided into several stages. The algorithm needs to map Haar features within each stage into different classifiers. A simple feature mapping example is illustrated in Figure 9. The example maps 26 features of 3 stages into 4 classifiers. Since each feature can be mapped to any one classifier and all classifiers are functionally equivalent, the total possible number of equivalent mappings is $m^n/m!$, where m is the number of classifiers and n is the number of features. The total number of possible mappings grows exponentially with the number of features. Thus, a brute force search solution is not feasible for the feature mapping problem.

We applied a simulated annealing [10] heuristic for the feature mapping problem. The heuristic's objective is to minimize the total number of wires of all the classifiers and to minimize the total delay of all classification stages. The two objects are illustrated in Figure 9. The total stage delay is the sum of the maximal delay of each stage. Since each classifier may be assigned with different numbers of features, the stage classification time is determined by the classifier with the most features. The total stage delay reflects the performance. Different feature mappings will result in different wire numbers and corresponding mux sizes for each classifier port. The total number of wires reflects the size of the entire communication architecture. Thus the heuristic optimizes the cost function: *Cost = total number of wires * total stage delay*.

The simulated annealing based design space exploration heuristic has two operations for generating neighbor solutions: Swap and Migrate, illustrated in Figure 9. Two classifiers exchange one of their features within one stage by the Swap operation. For instance, exchanging features 22 and 26 within stage 3 is a Swap operation. The Swap operation is a balanced operation, which will not change the total stage delay. The Migrate operation moves a feature from one classifier to another within a stage. For example, moving feature 22 from CF2 to CF1 is a Migrate operation. The Migrate operation is an unbalanced operation, which may change the total stage delay. The simulated annealing search heuristic is as follows:

Figure 9: Example feature mapping and neighbor generation



Step 1: Generate a random initial solution and calculate current cost: *CostC*. Define current temperature: $T = CostC$. Define best cost: $CostB = CostC$.
Step 2: Generate N neighbors by Swap operation and store the best Swap neighbor.
Step 3: Generate N neighbors by Migrate operation and store the best Migrate neighbor.
Step 4: Compare the best neighbors from Step 2 and Step 3 and choose the neighbor with smaller cost *CostN*. If $CostN < CostB$, then $CostB = CostN$ and store the best solution.
Step 5: Define: $D = CostN - CostC$. If $D < 0$, accept the neighbor, else use possibility $exp(-D/T)$ to accept it.
Step 6: Decrease T ($T=T*0.999$ in our experiments) and go back to Step2 until the ending condition is satisfied (100,000 iterations in our experiments).

We run the feature mapping exploration algorithm for 2, 4, 8, and 16 classifiers. The algorithm requires about 30 minutes per run on a 3.0 GHz Pentium 4 machine, for a total of 2 hours. The exploration tool automatically generates synthesizable VHDL code for the communication portion. The tool first allocates muxes of different sizes for each classifier port and then schedules the mux selection values according to the feature calculation order. The mux selection values for each step are stored into BRAM. The synthesis and implementation time for each version on the FPGA takes 1 to 3 hours. Since the exploration algorithm only needs to run once for each version, we chose simulated annealing parameters to yield many iterations and hence good results.

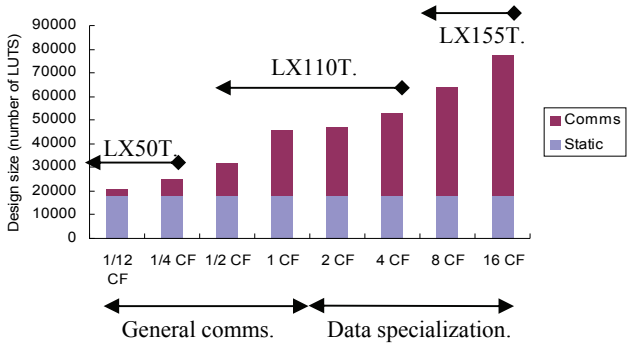
VI. Experimental results

We implemented the Haar-feature based object detection algorithm for frontal face detection and for eye detection. The Haar-feature sets and corresponding parameters for those objects come from the OpenCV library [9]. The architectures were fully implemented including real-time video capture, FPGA processing, and display of detected objects on an LCD monitor. A demonstration video is available [6].

A) Design scalability and mapping to different FPGA devices

We created the exploration tool to automatically output synthesizable VHDL code for the communication portion for different numbers of classifiers and manually wrote synthesizable VHDL for rest of the architecture. We implemented 8 different FPGA versions for face detection: 1/12, 1/4, 1/2, 1, 2, 4, 8, and 16 classifiers (CF). 1/12 CF means we only implemented one 400-to-1 mux, using time

Figure 10: FPGA resource utilization



multiplexing. 1/12, 1/4, 1/2 and 1 classifier versions each utilize the general communication architecture. The 2, 4, 8 and 16 CF versions each uses the data-specializatized architecture. The FPGA resource utilization (for frontal face) in LUTs is shown in Figure 10.

The FPGA resource utilization can be divided into two parts: Static components and Communication components. The communication components include muxes, wires between the integral image and classifiers and the classifier data-paths. The static components include video input/output interface, image scaler, integral image buffer, buffer controller, and others components except the communication components. Figure 10 shows that the communication components consume most FPGA resources for larger designs. Note also that the communication components' sizes increase linearly with the number of classifiers for the general communication architecture. The communication components' sizes for the data specialized architecture grows much slower. For instance, the communication components' size of 16 CF is only about 2 times of the communication size of 2 CF, as opposed to growing linearly with CF number when using the general communication architecture. The data-specialized architecture utilizes the content information of each feature to reduce the number of wires and mux size, which is more scalable than the general communication architecture. The FPGA resource utilizations for eye detection exhibited similar results and are omitted here for space.

We can map these designs to different FPGA devices. For instance, the Xilinx Virtex5 LX50T FPGA with 29,000 LUTs can implement 1/12 or 1/4 CF designs. The 155T FPGA with 97,000 LUTs can implement 8 and 16 CF designs. The 110T FPGA with 69,000 LUTs can implement mid-sized designs.

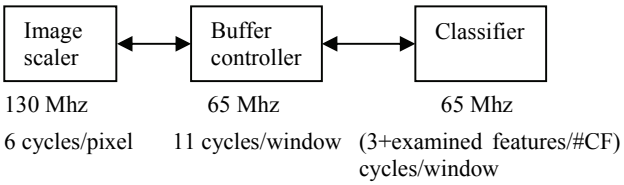
B) Timing analysis of major components

The major components (image scaler, buffer controller, and classifier) that determine design performance can execute in parallel, but they are synchronized with each other. The timing information of each component is shown in Figure 11.

The design uses a 65 MHz global clock for the buffer controller and classifier components. The image scaler runs with a 130 MHz clock. We analyzed the necessary operating cycles for each component to execute one image frame.

Since the scaling factor is 1.2, the scaled image's size is only $1/1.2^2 = 0.694$ of the original image. The total number of

Figure 11: Timing info of components



pixels the scaler needs to calculate for a 320x240 image is: $320 * 240 * (0.694 + 0.694^2 + 0.694^3 + \ldots) \approx 174{,}181$. The calculation time for 1 pixel is 6 cycles. The total scaling cycle number per frame is 522,545 (normalized to 65 MHz clock).

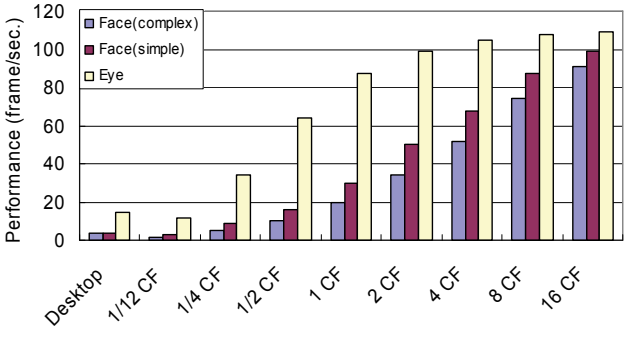
The current implementation of the integral image moves along the Y axis 1 pixel/window and moves along the X axis 4 pixels/window. This design is convenient for BRAMs with 4 bytes bandwidth. The buffer controller builds 1/4 of all possible sub-windows and is computationally comparable to the interleaved desktop version. The total number of sub-windows is $((320 - 20) * (240 - 20) / 4) * (1 + 0.694 + 0.694^2 + \ldots) \approx 53{,}922$. The total number of cycles of the buffer controller is 593,142 with 11 cycles/window speed. Note that the scaler and buffer controllers have comparable cycle numbers per frame. The design has some overlapping execution optimizations among components (omitted in the paper) that can almost hide the execution time of the scaler.

The classifier's execution time for one examine window depends on how many features the classifier calculates (cascade decision process) plus the 3-cycle pipeline filling time. For example, the classifier needs to execute 6 to 2138 cycles to determine a face in a window (1 classifier design). The overall detection time/window is *max (11, classification time)*, where 11 is the integral image construction time. If we ignore the classification time, the system performance upper bound is determined by the buffer controller, which is 65M / 0.593M $\approx$ 110 frames/sec.

C) Performance analysis

We compared the performance in terms of frames/second of different implementations for face and eye detection. A desktop version on a Pentium4 3.0 Ghz machine with 4 GB memory is compared to different FPGA implementations. The desktop version utilized the same fixed-point version of the object detection algorithm as the FPGA implementation for a fair comparison. The results are presented in Figure 12. The

Figure 12: Performance comparison of different implementations



results are measured by detecting a static 320x240 image repeatedly, because the VGA input rate is 60 Hz, which could become a bottleneck for some implementations. Two face images are tested in the experiments. The simple image has only one face, while the complex image has 12 faces. The eye image contains one eye.

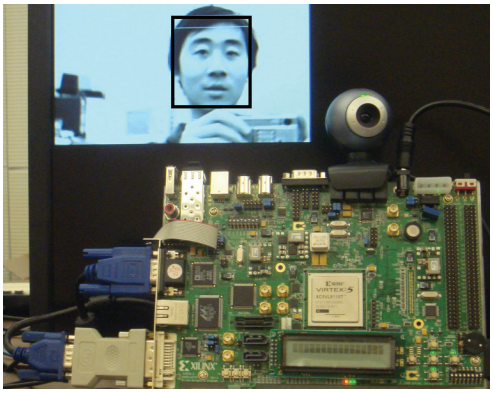
Note that eye detection is faster than face detection, because the eye only has 1066 features while the face has 2135 features. The average classification time for an eye is less than a face. The simple face image detection speed is on average 25% faster than the complex face image, because a successful face detection needs to pass all stages, which takes more cycles.

The desktop face detection speed is 3.5 frames/second, while the desktop eye detection speed reaches 15 frames/sec. The performance of desktop version is between 1/12 CF and 1/4 CF FPGA version. The largest 16 CF FPGA implementation is about 25 times faster than the desktop implementation for face detection and 7.3 times faster for eye detection. The detection results of the FPGA implementations are the same as the fixed-point desktop version.

Comparing the performance of different FPGA implementations, we note that performance increases almost linearly with the classifier number from 1/12 CF to 2 CF. For larger designs from 4 CF to 16 CF, the performance increases more slowly and the frame/second cannot surpass 110 frame/sec, because the buffer controller becomes a bottleneck.

Compared to Cho's [1] implementation of the same algorithm with 320x240 pixels on the same FPGA, the major difference is that we utilized custom exploration for the communication architecture, while their implementation only considered a general communication architecture. Our 1 CF version achieves similar performance using 28% less FPGA LUTs compare to their 1 classifier version (45,713 vs. 64,143). Compared to their triple classifier version, our 16 CF version, made possible by our scalable communication architecture having extensive exploration to reduce mux/wiring size, is about 4x faster, while using 8% less FPGA LUTs (77,059 vs. 84,232). The working platform is illustrated in Figure 13; a rectangle indicating a detected face can be observed.

Figure 13: The working platform boxes detected faces in real-time.



VII. Conclusions

We showed how to effectively implement Haar-feature based object detection accelerators on a modern series of FPGAs. As the design's communication architecture was the largest consumer of FPGA resources, we focused on designing a scalable communication architecture between the design's data buffer and its classifiers. We created a custom design space exploration algorithm based on simulated annealing to reduce the size of the communication architecture by using feature content information. The exploration was specific to Haar-based object recognition and not something that could be expected to appear in a general high-level or system-level synthesis tool. Other parameters of the Haar-based design could have also been explored. An IP (intellectual property) soft core for object detection utilizing a static or possibly parameterized VHDL or Verilog description would not cover the tremendous difference among generated designs described in the paper. As such, custom generators, including custom design space exploration, may become increasingly necessary for complex applications to be useful across a range of FPGA devices.

Acknowledgments

This work was supported in part by NSF CNS-1016792.

References

[1] Cho, J., Mirzaei, S., Oberg, J., and Kastner, R. 2009. Fpga-based face detection system using Haar classifiers. FPGA 2009.

[2] Collins, M., Schapire, R. E. and Singer, Y. 2002. Logistic Regression, AdaBoost and Bregman Distances. Mach. Learn. 48, 1-3 (Sep. 2002), 253-285.

[3] Gao, C., and Lu, S. Novel FPGA based Haar classifier face detection algorithm acceleration. FPL 2008.

[4] Gribbon, K. T. and Bailey, D. G. 2004. A Novel Approach to Real-time Bilinear Interpolation. In Proceedings of the Second IEEE international Workshop on Electronic Design, Test and Applications

[5] Hsu, R.L., Abdel-Mottaleb, M., Jain, A.K. Face Detection in Color Images. IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 696-706, May 2002.

[6] Huang, C. Object detection demo video, 2011, http://www.youtube.com/watch?v=gkQVanU5P5U.

[7] L'Insalata, N. E., Saponara, S., Fanucci, L., Terreni, P. Automatic Generation of Low-Complexity FFT/IFFT Cores for Multi-Band OFDM Systems. DSD 2007.

[8] Nordin, G., Milder, P. A., Hoe, J. C., and Püschel, M. Automatic generation of customized discrete fourier transform IPs. DAC 2005.

[9] OpenCV. opencv.willowgarage.com/

[10] Quinlan, J. R. 1993 C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc.

[11] Rowley, H., Baluja, S., Kanade, T. Neural Network-Based Face Detection. CVPR 1996.

[12] Theocharides, T., Link, G., Vijaykrishnan, N., Irwin, N.J., Wolf, W. Embedded Hardware Face Detection. 17th International Conference on VLSI Design, 2004.

[13] Viola, P., Jones, M. Rapid Object Detection using a Boosted Cascade of Simple Features. CVPR 2001.

[14] Wei, Y., Bing, X. and Chareonsak, C. FPGA implementation of AdaBoost algorithm for detection of face biometrics. BioCAS 2004.

[15] Xilinx ISE. http://www.xilinx.com/tools/webpack.htm

[16] Xilinx MicroBlaze. http://www.xilinx.com/tools/microblaze.htm