Accelerating Face Detection on Programmable SoC Using C-Based Synthesis

Nitish Srivastava   Steve Dai   Rajit Manohar   Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{nks45, hd273, rm92, zhiruz}@cornell.edu

Abstract

High-level synthesis (HLS) enables designing at a higher level of abstraction to effectively cope with design complexity of emerging applications on modern programmable system-on-chip (SoC). While HLS continues to evolve with a growing set of algorithms, methodologies, and tools to efficiently map software designs onto optimized hardware architectures, there continues to lack realistic benchmark applications with sufficient complexity and enforceable constraints. In this paper we present a case study of accelerating face detection based on the Viola Jones algorithm on a programmable SoC using a C-based HLS flow. We also share our insights in porting a software-based design into a synthesizable implementation with HLS-specific data structures and optimizations. Our design is able to achieve a frame rate of 30 frames per second which is suitable for realtime applications. Our performance and quality of results are comparable to those of many traditional RTL implementations.

1. Introduction

As the complexity of applications and hardware platforms continues to escalate, high-level synthesis (HLS) emerges as a popular alternative to traditional register-transfer-level (RTL) methods for improving design productivity that is crucial in today's rapidly-evolving technology landscape. By automatically generating digital circuits from behavioral specifications, it is able to significantly reduce design effort while efficiently exploring a large multidimensional design space. Designers can leverage HLS to quickly convert software designs into customized hardware and obtain quality of results competitive to time-consuming manual RTL implementations.

While a growing interest in C-based design has led to the release of a range of commercial and academic HLS tools along with an ever-improving set of design techniques, there continues to be a lack of sufficiently complex software applications with realistic design constraints that can be used to benchmark these tools. Applications in current HLS benchmark suites often only contain small application kernels, which are too simple to effectively reflect the influence of specific optimizations and detail the strengths and limitations of different tools in achieving the desired design constraints. Furthermore, current benchmark applications rarely require hardware-software partitioning to leverage a co-design methodology that takes advantage of the capability of modern tightly-integrated programmable system-on-chips (SoCs).

To address the challenges of providing realistic benchmarks for HLS tools, we identify face detection based on the Viola Jones algorithm [15] as a complex application whose achievable frame rate serves as a realistic performance constraint. Face detection is the task of finding faces within an image at different locations and irrespective of their size. It finds applications in a number of fields from photography to surveillance to robotics. The computationally intensive nature of the Haar feature classifiers in the Viola Jones algorithm makes face detection a suitable candidate for hardware acceleration.

In this paper we present a case study of accelerating a face detection system targeting a programmable SoC, emphasizing the insights from bringing a software design into a synthesizable implementation with specific data structures and optimizations. Our main contributions are twofold:

1. We identify Viola Jones face detection algorithm as a complex and realistic application for benchmarking HLS tools and provide a comprehensive case study to explore the flow from a pure software based implementation to an optimized C++ design suitable for HLS design flow.
2. We optimize our face detection system for performance, at the C/C++ level and synthesize it with a full-system compiler using SDSoC [9] from Xilinx. We show that our C-based design is suitable for real-time face detection applications achieving a frame rate of 30 fps. Our source code is publicly available on the authors' websites.

The rest of the paper is organized as follows: Section 2 examines the related work; Section 3 provides an overview of face detection based on the Viola Jones algorithm; Section 4 describes the baseline implementation; Section 5 discusses various optimizations performed; Section 6 presents performance and area results, followed by conclusions in Section 7.

2. Related Work

There have been many prior studies to evaluate state-of-the-art HLS tools [14, 16, 2]. Most of these works have used simple linear algebra and digital signal processing kernels such as matrix multiplication, FIR etc. CHStone represents a step towards benchmarking HLS with more realistic programs that make extensive use of high-level language features such as structs, pointers, and function calls [7]. However, most designs in CHStone remain small in size and are not necessarily good representatives of complex applications that can be handled by modern HLS tools. (e.g., those from the SoftFloat library [1]). MachSuite is a collection of 19 HLS benchmarks designed to span a variety of application domains that can potentially benefit from hardware acceraleration [13]. These benchmarks are constructed to be kernels instead of complete applications. More recently, Liu et al. have made available an HLS implementation of an H.264 decoder design [11]. The authors have provided insights on porting a complex C reference design on FPGA by applying a set of code optimizations and HLS directives.

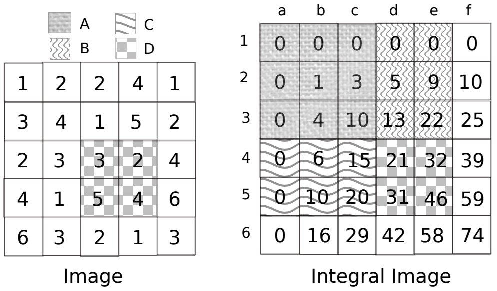
In this work we select Viola Jones face detection algorithm to benchmark FPGA-targeted HLS tool. FPGAs have become an at-

FPGA '17, February 22-24, 2017, Monterey, CA, USA
© 2017 ACM. ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021753

tractive platform for real-time face detection systems. Several prior works have explored the RTL implementation of face detection algorithm on FPGAs. Lai et al. designed a parallel hardware architecture for FPGAs and were able to achieve 143 frames per second (fps) for the VGA image (640×480) [10]. However, the number of classifiers used was very small for real-world applications (52 as compared to 2,913 classifiers in our case), which led to a poor accuracy. Ngo et al. presented an efficient modular architecture for detection of multiple faces in video streams and were able to achieve a frame rate of 30 fps on QVGA (320×240) [12]. However, their results were based on simulations instead of real hardware implementation. Gao and Lu designed an RTL implementation and were able to achieve a frame rate of 98 fps for 16 classifiers in parallel [6], although they had to retrain the Haar classifiers such that each stage includes classifiers in the multiple of 16. He et al. proposed an SoC architecture for face detection using artificial neural networks and achieved a frame rate of 624 fps [4]. However, the number of subwindow sizes used were 11×11, 19×19, and 17×17, which would result in a poor detection accuracy for small faces. Jin et al. have shown the best performance among the reported FPGA-based face detection systems by attaining a frame rate as high as 307 fps for VGA images [8]. But it is worth noting that their algorithm was based on face uncertainty map using local binary pattern transform instead of Viola Jones. Cho et al. implemented the Viola Jones algorithm on a Virtex-5 FPGA and were able to achieve 26 fps for three classifiers processing the image in parallel and 15 fps for a single classifier [3]. This work is closest to our implementation in terms of the overall system architecture, but their implementation was in RTL as opposed to HLS. To our knowledge, we are the first to implement Viola Jones face detection algorithm on FPGA using C-based synthesis, and achieve a frame rate of 30 fps, suitable for many real-time applications.

3. Face Detection Algorithm

Viola Jones face detection algorithm is a widely-used method for real-time object detection. It uses Haar-like features, which are inner products between the image and Haar templates. A face candidate is a rectangular section of the original image. As images may have faces of different sizes, an image pyramid is constructed by downscaling the image by a constant factor. This multiscale representation of image is then searched for all possible 25×25 faces. The inner product of Haar features requires the sum of different rectangular sections of the downscaled image. Integral image is an efficient way to sum up the pixel values within a rectangular region. The value at any location (x,y) of the integral image is the sum of the image pixel value above and to the left of the location (x,y). The Haar features are mainly of two types – two-rectangle feature and three-rectangle feature. The value of two-rectangle feature is the difference between the weighted sum of pixels within two rectangular regions. A three-rectangle feature feature is the weighted sum within the two outside rectangles subtracted from the weighted sum in center rectangle. The weights and size of each feature is generated using AdaBoost machine learning algorithm. The area of any rectangle within the original image can be computed very easily using each corner of the rectangle in the integral image (as shown in Figure 1), where the area of the rectangular section D is computed by adding the diagonal elements e5, c3 and subtracting the off diagonal elements e3 and c5. To test every rectangle for a potential face, a 25×25 sliding window shifts around the whole image after downscaling, with a pixel offset of 1. Each time the window shifts, the image region within the window goes through the cascade classifier, which consists of multiple stages of classifiers (as shown in Figure 2(c)). If the input region fails to pass the threshold of a stage, the cascade classifier immediately rejects the region as a face. If a region pass all stages successfully, it is classified as a candidate of face. The cascade filter can reduce the computation workload by rejecting a region at early stages. To compensate the effect of different lighting conditions, all the images are mean and variance normalized before sending them to the classifier.



1	2	2	4	1
3	4	1	5	2
2	3	3	2	4
4	1	5	4	6
6	3	2	1	3

	a	b	c	d	e	f
1	0	0	0	0	0	0
2	0	1	3	5	9	10
3	0	4	10	13	22	25
4	0	6	15	21	32	39
5	0	10	20	31	46	59
6	0	16	29	42	58	74

Figure 1: Image and its integral image — the value of the integral image at location c3 is the sum of the pixels in rectangle A. The value at location e3 is A+B, at location c5 is A+C, and at location e5 is A+B+C+D. The sum within the rectangle D can be computed as e5+c3-(e3+c5).

4. Baseline Implementation

In order to explore the flow from a software-based design to an optimized C/C++ based design suitable for HLS, we started with an open-source software implementation of Viola Jones face detection algorithm from [5]. The classifier used in this software implementation consisted of 25 stages, 2913 Haar classifiers, and Haar features trained by faces of size 25×25 pixels. We modified the source code to remove unsynthesizable constructs like system calls, heap accesses and recursive functions, to make it suitable for porting onto FPGA. This gave us a naïve hardware implementation of the face detection system with a frame rate $<$ 3 fps.

Figure 2 provides an overview of the entire system used for our design. It consists of a CPU connected to FPGA where a host program is running on the CPU and a face detection accelerator is running on the FPGA. The CPU sends the image in pgm format, where each pixel value is an 8-bit number, to the hardware accelerator. The face detection system implemented on the FPGA processes the entire image to detect all possible faces and returns the coordinates of the rectangles that are detected as faces to the CPU. The CPU then marks the faces by printing rectangles on the image. The hardware implementation consists of three main modules which are detailed in the following sections.

4.1 Image Scaler

This module is responsible for downscaling the image to form an image pyramid. It takes the original image and a scaling factor as inputs and returns the downscaled image using a simple linear interpolation algorithm. The linear interpolation algorithm is implemented using two nested loops iterating over the image height and image width respectively. The inner loop body has shift, multiplication, and assignment operations to perform downscaling. The typical scaling factor used for our design is 1.2

4.2 Integral Image Generator

This module takes a downscaled image from the image scaler and constructs an integral image which is then stored in BRAMs. It consists of nested loops, with outer loop iterating over the height of the image and the inner loop iterating over the width of the image. The inner loop updates integral image pixel values by accumulating the pixels in the same row and to the left of a pixel location in the downscaled image and adding the pixel value at the same location but one row above in the integral image. This module also has

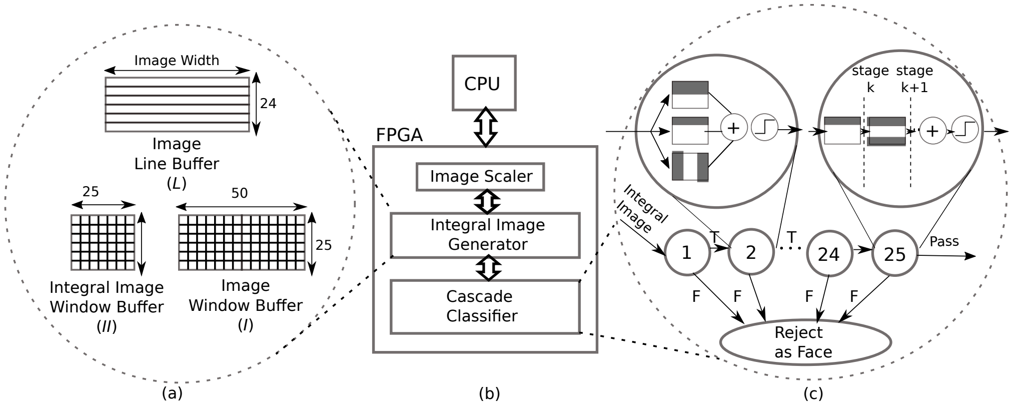


Figure 2: Face detection system — (a) Image line buffer, image window buffer and integral image window buffer for integral image generation, (b) Block diagram of the face detection system consisting of CPU and FPGA, and (c) Cascaded classifier with the classifiers in first 3 stages applied in parallel and in pipeline for rest of the stages

another nested loop which iterates over the rows and columns of the integral image and shifts the origin of the subwindow by one pixel location every iteration. The new subwindow location and the integral image is then sent to the cascaded classifier for further processing.

4.3 Cascade Classifier

This module receives an integral image and a subwindow location and passes the region inside the subwindow through the cascaded Haar classifiers as shown in Figure 2(c). For the pre-trained cascade classifier used for our implementation, there are 25 stages, each containing multiple Haar classifiers, ranging from 9 to 211. To implement cascading, this module contains a nested loop with the outer loop iterating over the number of stages and the inner loop iterating over the classifiers in each stage as shown in Figure 3. For each classifier in any stage, 12 values (x-y coordinates for the corner, plus width and height) corresponding to 3 rectangles in the Haar-classifier are read from the integral image subwindow, and the sum of each rectangle is obtained by adding the diagonal elements and subtracting the off diagonal elements of the rectangle. The sum of each rectangle is then multiplied by the corresponding weight and then added together and compared to a threshold value. Depending on whether the classifier sum exceeds the threshold value, one of the two classifier parameters α or β is accumulated into a running sum for that stage. For any stage, if the accumulated value exceeds the stage threshold, then it is considered to pass that stage and next iteration of the stage loop is processed, otherwise the function returns a negative value to the integral image generator, indicating that the given sub-window is not a face. In case the thresholds for all the stages are passed, then the subwindow is considered as a face and the cascade classifier notifies this by returning a positive value to the integral image generator, which then saves the upscaled version of those coordinates into a BRAM, so that they can be streamed out to the CPU when the processing of the entire image has finished. It also performs the normalization of the integral image by computing the mean and the standard deviation of the sub-window.

5. Optimizations

To improve the performance of our baseline implementation, we performed various optimizations as mentioned below:

5.1 Parallel and Pipelined Classifiers

We determined that the nested loop in the cascade classifier (Figure 3) is critical for the performance of the face detection system, as it is applied to all the subwindows in each downscaled image. The nested loop consists of a call to a Haar classifier and the best performance can be achieved when all the loops are completely unrolled and all the classifiers are processed in parallel. This requires a lot of hardware resources, making it infeasible to fit the design on the FPGA. Another approach is to pipeline the inner loop to exploit parallelism and to thus have a single pipelined classifier hardware whose classifier parameters change every cycle. This drastically reduces the amount of hardware resources required, but hurts the overall throughput. We wanted something in between these two approaches. We studied the number of sliding windows passing through each stage and the results obtained are shown in Figure 4. From the figure it can be seen that the number of subwindows passing through the first stage is two orders of magnitude more than the subwindows passing through the fourth and the fifth stages. Also, the subwindows passing through the second and third stages are an order of magnitude more than the ones passing from next two stages. This study clearly indicates that the first three stages are the throughput limiting factors. As the number of sliding windows passing through each stage is not the same due to cascading of the stages, the classifiers in the first three stages can be processed in parallel to increase the throughput at the cost of hardware resources. As the number of classifiers in the first three stages is small 9, 16, 27 respectively, this does not impose much area overhead and significantly improves the performance. For the next few stages, pipelining of the classifiers is more appropriate to save hard-

```
Cascade Classifier:
for (i=0;i < Nstages;i++) {
  for (j=0;j < Nclass[i];j++)
    stagesum += Classifier(II,classifierid,stddev);
  if (stagesum < sthresh[i]) break;
}

Classifier(II,k,stddev) {
  sum0 = (II[r0.y][r0.x]+II[r0.y+r0.h][r0.x+r0.w]
  -II[r0.y+r0.h][r0.x]-II[r0.y][r0.x+r0.w])*w0[k];
  sum1 = (II[r1.y][r1.x]+II[r1.y+r1.h][r1.x+r1.w]
  -II[r1.y+r1.h][r1.x]-II[r0.y][r1.x+r1.w])*w1[k];
  sum2 = (II[r2.y][r2.x]+II[r2.y+r2.h][r2.x+r2.w]
  -II[r2.y+r2.h][r2.x]-II[r2.y][r2.x+r2.w])*w2[k];
  finalsum = sum0+sum1+sum2;
  if (finalsum > cthresh[k]*stddev) return alpha[k];
  else return beta[k];
}
```

Figure 3: Unoptimized code for cascade classifier and a single classifier

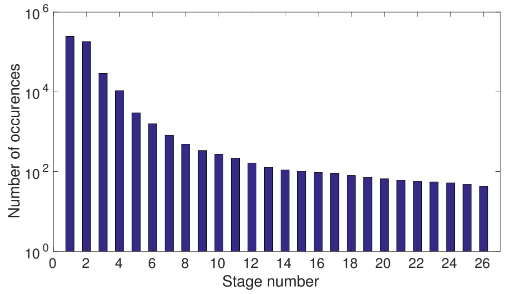


Figure 4: Number of occurrences of each stage on a 320×240 image

ware resources. Figure 2(c) shows an overview of the face detection system and optimized cascaded classifier with first 3 stages having classifiers in parallel and rest of them having pipelined classifiers.

To improve performance, we also stored the classifier values for the first three stages in registers, instead of BRAMs. The parallelization of the classifiers was done by making explicit function calls to each classifier, instead of doing it in a loop, as HLS tool we used schedules independent function calls outside the loop in parallel. Storing the classifier values in registers was done by hardcoding the constant values in the C code as shown in Figure 5. Hardcoding the classifier values reduces the need for BRAM accesses and gives the compiler more freedom to perform various optimizations. Some of these optimizations are shown in Figure 5, where multiplications by constants in the unoptimized code are replaced by shift and add operations in the optimized code produced by compiler. These kinds of optimizations, if lying on the critical path of the design as in our case, can help improve the performance of the design.

(a) Unoptimized Code

```
HardCodedClassifierk(II,variance){
  sum0=(II[6][5]+II[16][19]-II[16][5]-II[6][19])*-4096;
  sum1=(II[6][11]+II[15][13]-II[15][11]-II[6][13])*12288;
  sum2=0;
  if (sum0+sum1+sum2 > 58*variance) return 292;
  else return -89;
}
```

(b) Compiler Optimized Code

```
HardCodedClassifierk(II,variance){
  sum0=(-II[6][5]-II[16][19]+II[16][5]+II[6][19]);
  sum0=sum0 << 12;
  sum1=(II[6][11]+II[15][13]-II[15][11]-II[6][13]);
  sum1=(sum1 << 13) + (sum1 << 12);
  sum2=0;
  if (sum0+sum1+sum2 > 58*variance) return 292;
  else return -89;
}
```

Figure 5: Hardcoded classifier: (a) Unoptimized code with multiplication by constants (b) Compiler optimized code where multiplication is replaced by shift and add operations

5.2 Fast Integral Image Window Formation

As the coordinates of the classifiers are read from the integral image which is stored in BRAM, it requires 12 cycles to read the values of all the rectangle coordinates of a classifier. The classifiers are applied in the cascade classifier loop (Figure 3) which is critical for performance of the design. Reading classifier coordinates from BRAM imposes a resource constraint on the BRAM ports and prevents the tool pipeline the inner loop with the initiation interval of 1. To achieve an II of 1 in the inner loop, all the pixels of the integral image need to be stored in registers instead of BRAM. To store the integral image for the entire 320×240 image in registers, it would require 1,920,000 1-bit registers, while the FPGA board that we are using has only 437,200 1-bit registers. Hence, if the integral image for the whole image is calculated all at once, it can only be stored in BRAMs. To address this issue, we used the integral image formation method mentioned in [3]. With this method, instead of producing the entire integral image at once, our design produces an integral image subwindow every clock cycle and stores it in an array of registers. As the integral image subwindow is stored in registers, it allows all the classifier coordinates to be read in parallel.

Here we describe the algorithm used for integral image generation. The integral image generator is provided with a 24×320 image line buffer, 25×50 image window buffer, 25×25 integral image window buffer, 25×50 square image window buffer and 2×2 square integral image window buffer as shown in Figure 2(a). For each incoming pixel with coordinate (x, y) representing the origin of the sliding window, the image line buffer performs a shift operation as in (1), where n is the row size of image line buffer, $p(x, y)$ is the incoming pixel value, and $L(x, y)$ represents a pixel in the image line buffer.

$$\begin{aligned} &L(x, (n-2)-k) := L(x, (n-2)-(k-1)), \\ &L(x, n-2) := p(x, y) \text{ where } 1 \leq k \leq n-2 \end{aligned} \tag{1}$$

If each row of the line buffer can be stored in different BRAMs, then it is possible to perform all these operations in parallel. The image window buffer I is a two dimensional array of registers which stores the pixel values from the image line buffer L and the current pixel value $p(x, y)$. The purpose of image window buffer is to store the necessary pixels for integral image window formation. For each incoming pixel $p(x, y)$, the image window buffer performs the following operations:

$$\begin{aligned} &I(i, j) := I(i+1, j), \text{where } 0 \leq i \leq m-2, 0 \leq j \leq n-1 \\ &I(m-1, j) := L(x, j), \text{where } 0 \leq j \leq n-2 \\ &I(m-1, n-1) := p(x, y) \\ &I(i, j) := I(i+1, j) + I(i+1, j-1), \text{where } i+j = m-1, \\ &\qquad 0 \leq i \leq m-1, 1 \leq j \leq n-1, m = 2n, \end{aligned} \tag{2}$$

The integral image window buffer II is used for classification of a face, and stores the integral pixel values moving from the image window buffer. For each incoming pixel $p(x, y)$, the integral image window performs the following operation:

$$\begin{aligned} &II(u, v) := II(u, v) + I(u+1, v) - I(0, v), \\ &\text{where } 0 \leq u \leq n-1, 0 \leq v \leq n-1 \end{aligned} \tag{3}$$

Similar operations are performed for the square integral image SII, except we store $square(p(x, y))$ instead of $p(x, y)$ in the square image window buffer. The square integral image is used to calculate the variance used to normalize the pixel values in the subwindow to handle lighting conditions.

The HLS tool we use provides synthesizable data structures for conveniently instantiating window and line buffers. However, the available methods do not allow add operations while performing left or right shift operations, as required in (2). Therefore, we implemented our own window buffers and line buffers using two dimensional arrays and partitioning them in dimension 0 and 1 respectively. When partitioned in dimension 0, HLS tool infer the arrays as array of registers and for dimension 1 as multiple BRAMs. The operations in (1), (2) and (3) if not coded properly, can end up with dependencies which may restrict the tool to schedule them in single cycle. As seen from the equations, to compute the current value of II, the value of I in the previous cycle is required; and to compute current value of I, the value L in the previous cycle is required. We adopted a methodology where the equations in (1), (2) and (3) are coded in reverse order as shown in Figure 6, to avoid any read-after-write dependency. Loops were unrolled to schedule all the operations in single cycle.

```
/* Integral Image Window Buffer */
for (u=0;u < WINDOW_SIZE;u++)
  #pragma HLS unroll
  for (v=0;v < WINDOW_SIZE;v++)
    #pragma HLS unroll
    II[u][v]=II[u][v]+(I[u][v+1]-I[u][0]);

/* Image Window Buffer */
for(j=0;j < 2*WINDOW_SIZE-1;j++)
  #pragma HLS unroll
  for(i=0;i < WINDOW_SIZE;i++)
    #pragma HLS unroll
    if( i+j != 2*WINDOW_SIZE-1 ) I[i][j] = I[i][j+1];
    else if (i > 0) I[i][j]=I[i][j+1]+I[i-1][j+1];

for(i=0;i < WINDOW_SIZE-1;i++)
  #pragma HLS unroll
  I[i][2*WINDOW_SIZE-1]=L[i][x];

I[WINDOW_SIZE-1][2*WINDOW_SIZE-1] = IMG[y][x];

/* Image Line Buffer */
for(k=0;k < WINDOW_SIZE-2;k++)
  #pragma HLS unroll
  L[k][x]=L[k+1][x];

L[WINDOW_SIZE-2][x] = IMG1[y][x];
```

Figure 6: Code for integral image formation avoiding RAW dependencies and unrolling the loops for single cycle updates

5.3 Integral Image Banking

To read 12 coordinates from the integral image window simultaneously, it requires twelve 625×1 18-bit MUXes. As this many MUXes require more than 170K LUTs, the HLS tool has a hard time generating and pipelining these MUXes. Even if the tool is able to generate them, they are not able to pipeline them efficiently, resulting in timing violations during the routing phase of the design. As the number of LUTs required is huge, this also adds a lot of area overhead and routing congestion. Using a 625×1 MUX means that for any rectangle coordinate in a classifier the value can be read from anywhere in the integral image. We profiled all the classifiers to see how many pixel locations does each of the 12 coordinates require and realized that a rectangle coordinate comes from a blob of pixels in the integral image and does not use all the 625 pixels. We leverage this information to bank the integral image into 28 banks, such that any of the 12 coordinates of a classifier do not lie in the same bank. Figure 7(a) shows an integral image window buffer with a giant 625×1 MUX to read a single coordinate from the integral image. Figure 7 (b) shows a two-level hierarchy of MUXes, where the first level of MUXes select the values from different banks, and the MUX in the second level selects a bank. As all the coordinates lie in different banks, different offsets are used as select signals for different MUXes in the first layer to read all the 12 coordinates from the integral image (the MUX selects are set to 0 for 16 banks that are not needed). The MUXes in the first level are shared between all the 12 coordinates; only the second layer is replicated. This integral image banking helped us reduce the LUT utilization from 179,712 to 16,722 and also allowed the tool to place and route the design quite easily.

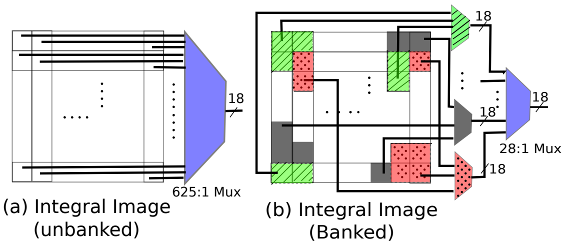


Figure 7: Integral image banking — (a) shows an unbanked integral image and a 625×1 MUX to read one coordinate from image. (b) shows an integral image banked into 28 banks, MUXes for 12 of these 28 banks read all the 12 coordinates and the MUX in the next layer is used to choose one of the banks for a coordinate

5.4 Square Root Approximation

As the first few stages constitute the throughput limiting factors, any logic before the cascaded classifier can bottleneck performance. To eliminate lighting effects, each subwindow has to be normalized. This requires the calculation of mean and standard deviation for every subwindow. As standard deviation is the square root of variance, we identified square root calculation as another throughput limiting factor because it takes 16 cycles on FPGA and is calculated for each subwindow. It is hard to parallelize due to inter-iteration dependencies in the loops. A better approach to achieve single cycle performance was to store the square root values in BRAMs and use direct look up for standard deviation calculations. As the number of bits required to represent the variance of a 25×25 window of 8 bit entries is 26, the square root look up requires a storage of $2^{26} \times 13$ bits. This translates to 47,331 BRAMs (each with 18Kb), which obviously would not fit on-chip. Hence we adopted a different approach where we treated variance as the sum of its most significant 10 bits left shifted by 16 and the lower 16 bits. The square root for the 16 bit numbers was stored in 32 BRAMs by declaring a statically initialized one dimensional array of 65536 elements. The square root calculation was performed by taking the square root of the upper and the lower halves of the number and left shifting the first result by 8 and adding them. We also made a separate case for the values which will be affected a lot by this approximation, and used upper 18 bits and lower 8 bits for them. We measured the percentage error for the approximated square root for all the 2^{26} bit numbers, and the error was less than 1% for 96% of the cases and less than 2% for 99.9% of the cases. For all our experiments, this 1-2% error did not effect the accuracy of face detection algorithm.

6. Experimental Results

The hardware-software setup used for our design consists of an ARM CPU and Zynq-7000 XC7Z045 FPGA available on a ZC706 board. We used Xilinx SDSoC 2016.1 to partition the application into software and hardware sections and automatically generate the data motion network between CPU and FPGA. SDSoC internally invokes Vivado HLS 2016.1 to synthesize the RTL from the design implementation in C. A high frame rate is very important for real time applications. Because the performance of our face detection system depends on the number of faces in the image, we measured the performance for different number of faces for both the software and hardware implementations. The software performance was measured on ARM Cortex-A9 present on ZC706 board.

Table 1 shows the performance of the implemented face detection system for 1, 2, 4 and 8 faces. It can be seen that the performance of the HLS-based face detection system is 8-9X higher than that of the software implementation in all four cases. The HLS design is able to achieve a frame rate of more than 30 fps for 1 to 4 faces, which is suitable for real-time application. Figure 8 shows the improvement in the frame rate for different optimizations. All the optimizations are performed along with fast integral image generation and integral image banking. From the figure it can be seen that baseline performance ranges from 1–3 fps, and all three optimizations (pipelined classifiers, parallel classifiers, and square root approximation) contribute almost equally to increasing the frame rate of the face detection system.

Table 1: Performance of our proposed face detection system with 320×240 resolution images for both hardware and software

# of faces	Software classifier	Hardware classifier
1	206 ms 4.8 fps	30 ms 33.4 fps
2	232 ms 4.3 fps	31 ms 32.1 fps
4	250 ms 4.0 fps	32 ms 31.3 fps
8	371 ms 2.7 fps	38 ms 26.3 fps

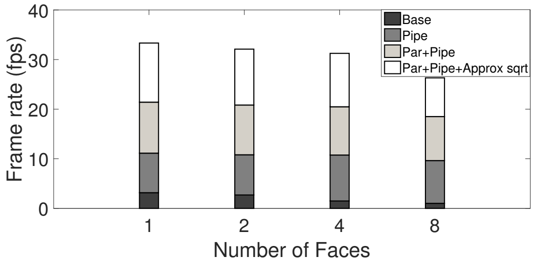


Figure 8: Frame rate improvement corresponding to various optimizations for 1, 2, 4 and 8 faces in 320×240 image

Table 2 shows the resource utilization of the face detection system. Most of the BRAM utilization comes from the storage of the original and the downscaled image in the on-chip BRAMs, each of which consumes 64 BRAMs. The rest of the BRAM utilization comes from the storage of different classifier parameters (weights, rectangle co-ordinates, thresholds, etc.). As we want single cycle access to all these parameters, any two classifier parameters that may be needed at the same time are stored in different BRAMs. The main factors of LUT consumption are the MUXes in integral image banking, logic for the integral image generation and arithmetic expressions, contributing 24%, 50% and 26% respectively toward the LUT utilization. Register utilization mainly comes from integral image generation (57%) and temporary storage (43%) in various modules and classifiers.

Table 2: Resource Utilization of our proposed face detection system with 320×240 resolution images

Logic	Total Used	Total Available	Utilization
LUT	62,522	218,600	28.6%
Registers	81,135	437,200	18.56%
DSP48E	111	900	12.33%
BRAM 18K	157	545	28.81%

7. Conclusions

In this paper, we provide insights for bringing a software design of Viola Jones face detection algorithm into a synthesizable C-based design. We describe various optimizations performed to achieve a frame rate suitable for real-time application. We acknowledge the strength of SDSoC and Vivado HLS in automatically generating the data motion network and pipelining complex loop structures. We also discuss the shortcomings of the tool in restricting the designer from having fine control over the design and synthesis process. We conclude that more advanced benchmarking is needed to create HLS tools with out-of-box quality-of-results competitive to traditional RTL implementations.

Acknowledgement

This research was supported in part by NSF Awards #1065307, #1337240, #1453378, and a research gift from Xilinx, Inc.

References

[1] SoftFloat, http://www.jhauser.us/arithmetic/SoftFloat.html.

[2] E. Casseau and B. L. Gal. High-Level Synthesis for the Design of FPGA-Based Signal Processing Systems. *Int'l Symp. on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2009.

[3] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner. FPGA-Based Face Detection System Using Haar Classifiers. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2009.

[4] H. Chun, A. Papakonstantinou, and D. Chen. A Novel SoC Architecture on FPGA for Ultra-Fast Face Detection. *Int'l Conf. on Computer Design (ICCAD)*, 2009.

[5] F. Comaschi. https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection.

[6] C. Gao and S. L. Lu. Novel FPGA Based Haar Classifier Face Detection Algorithm Acceleration. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2008.

[7] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii. Chstone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Int'l Symp. on Circuits and Systems (ISCAS)*, 2008.

[8] S. Jin, D. Kim, T. T. Nguyen, D. Kim, M. Kim, and J. W. Jeon. Design and Implementation of a Pipelined Datapath for High-Speed Face Detection Using FPGA. *IEEE Trans. on Industrial Informatics (IEEE T IND INFORM)*, 2015.

[9] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo. SDSoC: A Higher-Level Programming Environment for Zynq SoC and Ultrascale+ MPSoC. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[10] H. Lai, M. Savvides, and T. Chen. Proposed FPGA Hardware Architecture for High-Frame Rate (>>100 fps) Face Detection Using Feature Cascade Classifiers. *Int'l Conf. on Biometrics: Theory, Applications, and Systems (BTAS)*, 2007.

[11] X. Liu, Y. Chen, T. Nguyen, S. Gurumani, K. Rupnow, and D. Chen. High-Level Synthesis of Complex Applications: An H.264 Video Decoder. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[12] H. T. Ngo, R. C. Tompkins, J. Foytik, and V. K. Asari. An Area Efficient Modular Architecture for Real-Time Detection of Multiple Faces in Video Stream. *Int'l Conf. on Information, Communications and Signal Processing (ICICS)*, 2007.

[13] B. Reagen, R. Adolf, Y. Shao, G. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[14] S. Skalicky, C. Wood, M. ukowiak, and M. Ryan. High-Level Synthesis: Where Are We? A Case Study on Matrix Multiplication. *Int'l Conf. on Reconfigurable Computing and FPGAs (ReConFig)*, 2013.

[15] P. Viola and M. J. Jones. Robust real-time face detection. *Int'l Journal of Computer Vision (IJCV)*, 2004.

[16] F. Winterstein, S. Bayliss, and G. A. Constantinides. High-Level Synthesis of Dynamic Data Structures: A Case Study Using Vivado HLS. *Int'l Conf. on Field Programmable Technology (FPT)*, 2013.