Real-time Object Detection in Software with Custom Vector Instructions and Algorithm Changes

Joe Edwards
University of British Columbia
VectorBlox Computing Inc.
jedwards@ece.ubc.ca

Guy G.F. Lemieux
University of British Columbia
VectorBlox Computing Inc.
lemieux@ece.ubc.ca

***Abstract*—Real-time vision applications place stringent performance requirements on embedded systems. To meet performance requirements, embedded systems often require hardware implementations. This approach is unfavorable as hardware development can be difficult to debug, time-consuming, and require extensive skill. This paper presents a case study of accelerating face detection, often part of a complex image processing pipeline, using a software/hardware hybrid approach. As a baseline, the algorithm is initially run on a scalar ARM Cortex-A9 application processor found on a Xilinx Zynq device. Next, using a previously designed vector engine implemented in the FPGA fabric, the algorithm is vectorized, using only standard vector instructions, to achieve a 25× speedup. Then, we accelerate the critical inner loops by adding two hardware-assisted custom vector instructions for an additional 10× speedup, yielding 248× speedup over the initial Cortex-A9 baseline. Collectively, the custom instructions require fewer than 800 lines of VHDL code, including comments and blank lines. Compared to previous hardware-only face detection systems, our work is 1.5 to 6.8 times faster. This approach demonstrates that good performance can be obtained from software-only vectorization, and a small amount of custom hardware can provide a significant acceleration boost.**

I. Introduction

Increasingly sophisticated image processing is required in embedded systems. The algorithms in these image processing pipelines often have both intensive computing requirements and real-time constraints. A high degree of parallelization is required. On a desktop machine, where cost and power are not paramount, real-time performance can often be achieved using SIMD instructions or GPGPU processing. In embedded systems, custom hardware solutions are often required to meet performance targets. FPGAs represent an ideal platform for creating these system-on-chips (SoC), as they can be used for computationally demanding processing as well as sensor integration. To improve productivity and reduce design cost, FPGA users should minimize RTL design.

In a system where many algorithms in the image processing pipeline demand acceleration, the amount of custom hardware quickly grows. This requires larger, more costly FPGAs and increased development effort. Instead, software-based acceleration adds flexibility to the design; with minimum development effort, changes to algorithms and implementations are possible as requirements change.

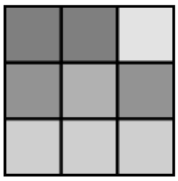
This paper demonstrates a hybrid hardware/software approach to developing a face detection system. It first uses a soft vector accelerator to improve software performance as much as possible. Then, it adds a small amount of hardware, in the form of custom vector instructions, for a final performance boost. The amount of custom hardware is kept small, allowing most of the processing to be specified in software.

Fig. 1: Photo of video output showing 49 of 50 faces detected on a 1080p image in 36ms

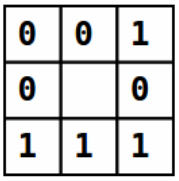
Previous efforts at implementing face detection on FPGAs have produced good results [3], [4], [6]. However, these solutions are single-use, fixed implementations that can be difficult to maintain. Also, the developed hardware is only useful for object detection and no other purpose. In contrast, our approach results in faster detection rates than the hardware systems, yet it is software-driven, meaning the same hardware can easily can be modified to do pre-processing and post-processing such as image enhancement.

In our solution, we also develop a novel approach for representing AdaBoost cascades with reduced bitwidth fixed-point values that match the exact decision-making logic desired, without any chance of overflow or floating-point round-off errors. We solve for these fixed-point values using integer linear programing (ILP). To our knowledge, this approach has never been done before.

Our face detection system produces the results shown in Figure 1. It is fully implemented on a Xilinx ZC706 board (XC7Z045 SoC device) and an Avnet HDMI I/O daughterboard. With suitable settings, it performs real-time face detection on 1080p60 video input, overlaying results live on 1080p60 output. The FPGA fabric contains a streaming HDMI I/O system as well as the soft vector processor. Framebuffers store 32-bit RGB pixels in HPS-attached DRAM. The software



(a) compute brightness



"00101110" == 56

(b) 8-way compare to neighbours

```
if (LUT[56]) {
  VALUE = PASS
} else {
  VALUE = FAIL
}
```

(c) lookup pattern

```
STAGE += VALUE
```

(d) add pass or fail

Fig. 2: Overview of MB-LBP feature detection. A search window traverses an image pyramid, and calculates multiple features at each position. Each feature compares average brightness of the center of a 3×3 window against the 8 neighbours, producing an 8-bit comparison bitfield. This 8-bit LBP value produces either a PASS or FAIL value, which is summed with all features in the same stage. A stage passes, meaning a face might be present, if the sum exceeds a stage-threshold. A face may be detected only if all stages pass.

runs on the ARM Cortex-A9 processor and VectorBlox MXP FPGA-based vector accelerator.

The rest of this paper is organized as follows. Section 2 presents background material on object detection and vector processing. Section 3 presents the software-only algorithm changes that produce the initial 25× speedup. Section 4 presents the custom vector instructions that lead to a further 10× speedup. Section 5 presents results and a comparison to prior work, while Section 6 concludes.

II. Background

A. AdaBoost Object Detection

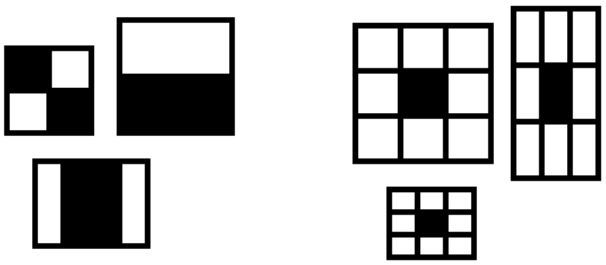
AdaBoost-based object detection, characterized by applying a series of weak filters, was popularized by Viola and Jones' seminal paper [11]. It is employed in a variety of computer vision applications including face-based auto-focusing or safety warnings when pedestrians are detected in the paths of vehicles. OpenCV, a popular open source library for computer vision, provides efficient implementations for both training and detection routines for these classifiers and contains support for various features [2].

An image pyramid is processed so all sizes of an object can be found (scale-invariant). A sliding window is used to check for objects at every location. A cascade of weak classifiers is used to determine if the search window contains the target object. Each weak classifier specifies a feature to check at a given offset within the search window. For each feature, a pass or fail score is summed to produce a stage total. After all features in the stage are evaluated, this stage total is checked against a required stage-threshold. If not met, the search exits early, indicating that no object is found. Exiting early allows for a minimum number of stages to be evaluated, greatly improving runtime. A second key feature introduced by Viola and Jones that greatly aids performance is the use of integral images (sum area tables). Calculating features usually requires summing regions inside the search window. Integral images allow the sum of any rectangular region to be calculated with only four lookups (one at each corner) and three simple arithmetic operations.

1) Haar vs LBP Feature Types: Viola and Jones used Haar features in their work. Haar features use the average intensity of up to 4 rectangular regions and check to see if a specified relationship is present between them. Integral images makes computing these regions efficient. The variance of the search window is also used in the calculation to make features more robust to lighting, and therefore a squared version of the integral image must also be generated.

Alternative feature types have been used successfully and are more amenable to quick calculation. Local binary pattern (LBP) features rely on generating a pattern based on comparing a central region to its eight neighbouring regions. This 8-bit pattern is then checked in a lookup table to see if it is among the patterns that pass. The operations used in calculating LBP features map well to embedded processors, using integer operands and avoiding the division and square-root operations required for calculating Haar features. Although more regions need to be calculated for each feature, each LBP feature is more expressive than Haar, resulting in cascades with fewer features overall.

Liao et al. introduced multi-block LBP (MB-LBP) features, where the block sizes used to generate the LBP pattern may be larger than 1 pixel [7]. Figure 2 demonstrates how MB-LBPs are calculated in an object detection system. Summed area tables can again be used to calculate these regions faster. Multi-block LBP features increased accuracy over single-pixel based LBP patterns. Examples of typical Haar and MB-LBP features are shown in Figure 3 as they would appear inside the search



(a) Haar (b) LBP

Fig. 3: Examples of typical features

window. MB-LBP features are used in our implementation due to the computational advantages over Haar features.

B. Soft Vector Processor

The main reason for selecting a soft vector processor is to minimize the development effort required to accelerate algorithms. Software-based iterations greatly reduce compilation time and enable rapid debugging with familiar software development tools.

For this paper, we adopt a parameterizable and extensible soft vector processor, the VectorBlox MXP [10], shown in Figure 4. A host processor controls the vector engine by sending it sequences of 2-input, single-output, variable-length SIMD instructions. The engine size is configured as the number of 32-bit vector lanes. Subword-SIMD is supported, so 8-bit and 16-bit data types have 4× and 2× more parallelism, respectively.

Vectors are divided into wavefronts that match the combined width of all ALUs. A vector operation proceeds one wavefront at a time, one cycle per wavefront, until the entire vector has been computed.

Both the inputs and the outputs are stored in a configurable scratchpad memory, allowing for an efficient implementation with variable vector lengths and unaligned accesses. A DMA engine moves the vectors between main memory and the internal scratchpad memory. This is handled by the programmer. Double buffering hides data transfer overhead through prefetching. The processor supports instruction pipelining and hazard detection. Long vectors help reduce pipeline stalls created by hazards, keeping the engine fully utilized.

1) Custom Vector Instructions: The VectorBlox MXP allows use of *custom vector instructions* or CVIs [9]. These CVIs can be used by a programmer the same as any other vector instruction, taking full advantage of the vector processor's pipelining and data marshalling. CVIs provide the ability to leverage custom hardware while keeping the design effort to a minimum.

2) Wavefront Skipping: A common problem that plagues SIMD parallelism is control flow divergence. Branches in SIMD engines are commonly handled by masked or predicated instructions, but this requires all elements in the vector to be processed even if masked. This wastes performance by occupying potential execution slots with no-operations.

To reduce the impact of control flow divergence, the VectorBlox MXP supports wavefront skipping [8], a form

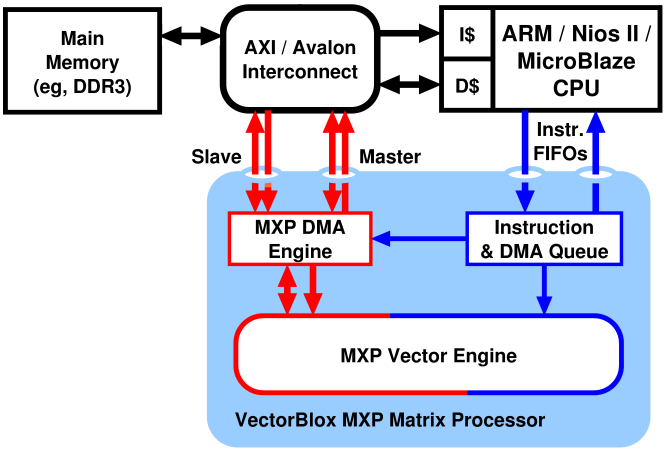


Fig. 4: Overview of the VectorBlox MXP processor

of density time masking. This is beneficial when multiple instructions are executed with the exact same mask. On a first pass, the mask is analyzed to memorize those wavefronts where all elements are masked. On subsequent passes, i.e. during instruction execution, these 'empty' wavefronts can then be skipped entirely and occupy 0 cycles of the ALUs. This results in faster execution as a mask widens to enable fewer and fewer elements. Custom vector instructions in MXP can also be masked.

III. Algorithmic Optimization

In this section, we start with a scalar implementation, and gradually optimize it for performance. We perform an initial vectorization, then improve it using restricted block sizes along with precomputation. Next, we use masked vector operations to further improve performance. Finally, we use an ILP formulation to generate fixed-point values for the cascade computation of each stage, replacing 32-bit floating-point computation, which is susceptible to rounding errors, with robust 8-bit integers. The use of 8-bit calculations does not improve software performance in this section, but they allow for increased parallelism in the custom vector instructions presented in Section IV.

In all cases, we have measured performance and verified results using a Xilinx ZC706 board with 1080p HDMI input and output. With this setup, we can use live camera input, and visually see where faces are tagged on the output display.

A. Initial Vectorization

Our initial scalar implementation was compiled with `gcc -O3`, running bare metal on the ARM Cortex-A9 processor at 667 MHz. An object dump shows that the compiled code contains NEON instructions (ARM's native SIMD extensions), but we did not try to optimize the scalar code to aid NEON auto-vectorization.

Next, manual vectorization for MXP across rows is quickly implemented and verified against the scalar implementation. This initial vectorization gave us an understanding of which functions are amenable to SIMD parallelization, and how performance may scale as more ALUs are added. We did not

vectorize all of the code; we used profiling to identify and vectorize only the most compute-intensive loops.

Vectorization of the innermost loop is done across a row of pixels, thus testing many possible $< x,y >$ starting positions in parallel. The next two outer loops must test all stages, and then test each feature within each stage. These outer loops cannot be SIMD-vectorized because each feature and each stage have unique properties. Furthermore, the *early exit* condition means that as soon as a stage fails, no other stages need to be tested at that position. This performs well in scalar or MIMD implementations, but leads to low utilization in SIMD engines as the early exit positions are masked off within the vector, but they continue to use execution slots until all positions fail or all stages are tested.

After vectorizing, the inner loop that computes each feature in the stage consumes the most runtime (93%). Other components, including RGB to grayscale and downscaling (bilinear interpolation) to create the image pyramid, consume the second-most runtime (7%). We will optimize downscaling later in Section IV. The appending and merging of features to produce the final results consumes minimal runtime ($< 1\%$) when only a handful of faces are present, but it can potentially be vectorized as well.

The innermost loop consists of two parts: computing the LBP pattern at this feature's specified location, and performing a table-lookup with the pattern to determine its contribution towards passing the stage.

Using a 320x240 test image, we perform a dense scan (scaling factor = 1.1, stride = 1) to compare with previous work [3]. The scalar core takes 1.9 seconds to complete. Our initial vectorized version of the code, running on 16 vector lanes, processes the test image in 0.83 seconds, a improvement of nearly 2.3×. The vector processing is done at 166MHz, which is 1/4 of the rate of the scalar engine.

B. Restricting LBP Block Sizes

Multi-block LBP patterns allow any block size to be used for a feature. Using OpenCV's frontal face detector as an example, the distribution of block sizes across all features is shown in Figure 5(a). This graph indicates that smaller block sizes are used more often. Also, blocks typically have similar width and height.

By restricting the block sizes, two things occur. First, by using only powers of two, the computation better fits SIMD-style execution. Second, we notice that adopting the same block size across many features leads to computational efficiency due to aliasing which allows us to memoize these patterns. Given a specific $< x0,y0 >$ position, each feature uses a different offset, so there is no aliasing and all computations are necessary. However, across two different starting positions, the computation for some feature i may be aliased to a feature j with the same block size. Thus, we can transform the innermost computation, which performs redundant work due to aliasing, into a lookup operation, where the work is done just once as a precomputation. We must precompute an entire image worth of results, once for each block size. Hence, reducing the number of block sizes is of great importance.

Cascade	True +	False +	PPV
haarcascade_frontalface_default.xml	451	55	0.89
haarcascade_frontalface_alt.xml	451	10	0.97
haarcascade_frontalface_alt2.xml	455	9	0.98
lbpcascade_frontalface.xml	396	27	0.94
unrestricted_lbp_frontalface.xml	385	9	0.98
restricted_lbp_frontalface.xml	381	12	0.97
restricted2_lbp_frontalface.xml	386	17	0.96

TABLE I: Accuracy of frontal face cascades on MIT-CMU, showing true/false positives and positive predictive value

When restricting the block width or height to a power of two, we get the distribution shown in Figure 5(b). If we also restrict the width to equal the height, we get the distribution shown in Figure 5(c).

Recent work by Bilaniuk et al. [1] has made similar observations: restricting block sizes to powers of two is better for SIMD engines, and using just three block sizes (1×1, 2×2, and 4×4) makes it more efficient to use precomputation. In their results, these restrictions did not significantly change true positive detection rates, but it did increase false positives from below 1% to above 5%.

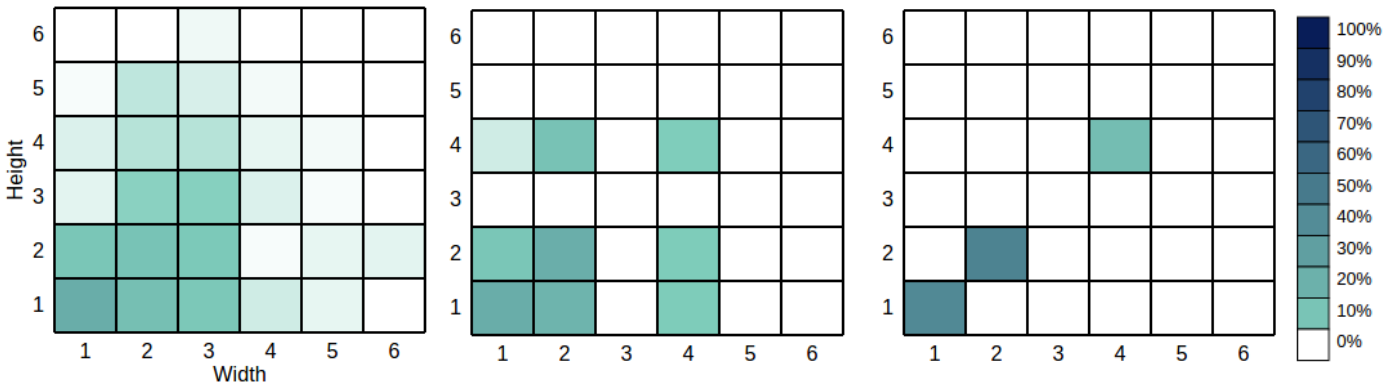
To measure the tradeoff in accuracy versus number of block sizes, we created three versions of the cascade. We used OpenCV's traincascade program to train the new cascade. By modifying `lbpfeatures.cpp`, we can exclude unwanted features sizes from the training process. Three versions were generated with traincascade: (1) an unmodified, unrestricted version, (2) a restricted version, where width and height must be powers of two, (3) and a restricted2 version, where width and height are equal and must be a power of two. This requires minimal changes to the source code.

Each of the cascades produced are valid and can be verified and used outside of our embedded implementation. Like Viola and Jones, we use the MIT-CMU test set (sets A, B, C) to quantify the trade off in accuracy and restricted features. The results for OpenCV's default frontal face cascades and new trained classifiers are presented in Table I. The unrestricted and restricted classifiers perform more accurately than OpenCV's default Haar and LBP frontal face cascades. Like Bilaniuk's result, our false positives also increase by a small amount.

The initial switch to restricted square features with precomputed features produces a speedup of $1.8\times$ over the initial vectorized solution with 16 lanes.

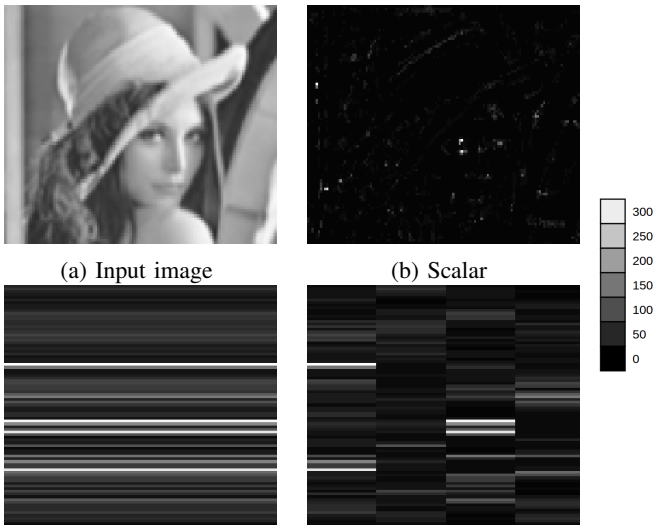
C. Applying Wavefront Skipping

Long vector lengths, often good for performance, can work against the benefit of exiting early. This is seen in Figure 6. The sub-image (b) shows the amount of computation done by a scalar CPU (without SIMD) at each location in the image; bright pixels indicate highly probable locations for a face, whereas black pixels indicate an exit-early condition. Row-based vectorization shown in sub-image (c) shows the extra work done across the entire row, because the whole row must compute until the last pixel is done. Finally, sub-image (d) shows how some work can be skipped when using VectorBlox



(a) Unrestricted (b) Restricted (c) Restricted Square

Fig. 5: Distribution of block sizes of LBP features in the trained classifiers



(a) Input image (b) Scalar

(c) Simple rows (d) Masked rows

Fig. 6: The number of features calculated at every location is shown. The bottom demonstrate parallelizing across a row, with the latter taking advantage of masked instructions

MXP masks: entire wavefronts within a vector that are masked off can be skipped entirely and do not consume execution slots.

After setting up a mask for the vector, we iterate through stages and update the mask after every stage. When all locations within a wavefront exit early, the vector engine skips that wavefront. The speedup from using masked instructions is $6\times$ versus the non-masked version.

D. Reducing Data Size with ILP Formulation

By supporting subword-SIMD, the VectorBlox MXP provides increased performance with smaller data sizes. Our vector code generally uses the minimum possible data size for maximum performance.

In most AdaBoost implementations, to determine whether a stage passes or fails, a series of 32-bit floating-point values (representing pass or fail for each feature) are added together and compared to a 32-bit floating-point threshold. This uses more data bits than necessary, and may even be susceptible to round-off errors since floating-point addition is not commutative or associative.

For increased performance on MXP, this computation can use 8-bit integers instead. Also, integer computation will not be susceptible to round-off errors. However, the number of features per stage must be limited, and the pass/fail values for each feature must be carefully chosen to avoid overflows.

In training, the AdaBoost algorithm determines which features in a stage must pass for the stage to pass. Some features are deemed more important, and given larger weight in the form of increased values in their pass or fail scores. Ultimately, however, this decision logic is encoded into a summation and comparison to a determined threshold.

In our implementation, we use the exact same decision logic to write out a series of constraints for an ILP solver (Microsoft Research's Z3 theorem prover [5]). These constraints allow the ILP solver to assign 8-bit pass and fail score values, p_i and f_i, for each feature i, thus preserving the original logic. We are also able to assign a threshold of 0, so a positive stage total passes and a negative stage score fails.

For example, for a stage with 5 features, there are 2^5 constraints representing all combinations of each feature either passing or failing. Each constraint is given one clause to pass the stage (result ≥ 0) or fail the stage (result < 0). However, a complementary second clause is also necessary to avoid overflows (result $\leq$ 127 or result ≥ -128). An additional $2 \times 5 = 10$ constraints force all of the p_i and f_i values within the 8-bit signed integer range. Z3 is used to solve for these 10 values. An example of these constraints is shown in Figure 7.

One challenge with this approach is that Z3 starts having trouble with stages that have too many features (eg, 15 or more) using these 8-bit constraints. We avoided this in practise by limiting the number of features per stage. Our final trained cascade uses 98 features across 12 stages, with at most 9 features per stage.

Note that the vectorized version presented thus far is bottlenecked on the LUT computations. Hence, there is little performance advantage in switching from 32-bit weights to

```
Subject To
C00a: f0+f1+f2+f3+f4 <=  -1 // fail
C00b: f0+f1+f2+f3+f4 >=-128 // fail
C01a: f0+f1+f2+f3+p4 >=   0 // pass p4
C01b: f0+f1+f2+f3+p4 <= 127 // pass p4
C02a: f0+f1+f2+p3+f4 <=  -1 // fail p3
C02b: f0+f1+f2+p3+f4 >=-128 // fail p3
C03a: f0+f1+f2+p3+p4 >=   0 // pass p3+p4
C03b: f0+f1+f2+p3+p4 <= 127 // pass p3+p4
...
C31a: p0+p1+p2+p3+p4 >=   0 // pass all
C31b: p0+p1+p2+p3+p4 <= 127 // pass all
Bounds
-128 <= f0 <= 127
-128 <= f1 <= 127
...
-128 <= p4 <= 127
```

Fig. 7: Sample ILP constraints for a stage with 5 features

```
#define VLUT VCUSTOM0
for (f = 0; f < cascade[stage].n; f+=2) {
  // sz = MB-LBP size, w = image_width
  fa = cascade[stage].feats[f];
  fb = cascade[stage].feats[f+1];
  v_a = v_lbp[fa.sz] + fa.dy*w + fa.dx;
  v_b = v_lbp[fb.sz] + fb.dy*w + fb.dx;
  vbx_masked( VVB, VLUT, v_t, v_a, v_b );
  vbx_masked( VVB, VADD, v_s, v_s, v_t );
}
```

Fig. 8: The inner loop using a custom vector instruction

8-bit weights in software at this point. However, in the next section, we will add a custom vector instruction to directly support the LUT operation. At that point, using an 8-bit output weight is far better for reducing hardware and improving performance. This is an example of a hardware-motivated software change.

IV. Custom Vector Instructions

In this section, we accelerate the computation with two key custom vector instructions (CVIs). After the algorithmic refinements above, profiling reveals that the table lookup operation dominates runtime. This operation is an ideal candidate for implementation as a custom vector instruction. Once accelerated, we find the image pyramid (downscaling) and the LBP precomputations become the bottlenecks. The downscaling is resolved by vectorizing the software (details omitted due to space), and the LBP precomputation requires its own custom vector instruction. These two custom vector instructions, for table lookup and LBP precomputation, are described below.

A. LBP Table Lookup Instruction

The LBP table lookup operation is accelerated with the first custom instruction. Algorithmically, this instruction implements the two steps shown in Figure 2(c) and (d), representing the lookup followed by an addition. The input is an 8-bit value corresponding to the LBP pattern, shown as the value 56 in the figure, which is the result of the 8-way LBP comparison. (Further details about the LBP comparison are given in the next section.) The output is an 8-bit value, produced by the second step where two table lookup results are added together.

For each feature, the first step is a table lookup into a 256-entry table with 1 output bit. The output bit selects one of two 8-bit values for the feature (aptly named `PASS` or `FAIL`). Thus, each feature requires 272 bits of storage.

For performance, we can perform two table lookups in parallel using the two 8-bit input operands of the custom instruction. Since the custom instruction can only provide a single output, the second step adds the PASS or FAIL results for these two features into an 8-bit partial stage total named `STAGE`.

To determine whether a stage passes or fails, the results of all features in the stage need to be accumulated. This is done using regular 8-bit vector add instructions, accumulating all partial stage totals into the final stage total. After processing all features, this final stage total is compared to a threshold of zero to determine whether the stage passes.

As mentioned, the custom instruction invokes two table lookups in parallel in each lane. This is done by reading two different 8-bit LBP patterns and presenting them to the CVI as operands A and B, respectively. This requires a 544-bit wide memory. However, since all 8-bit vector lanes are processing the same feature, this memory is shared across the entire vector engine. Back-to-back custom instructions that perform table lookups automatically increment a feature counter, which is used to address into the 544-bit wide memory. This memory must be large enough for all features across all stages; the starting address is determined by the stage number. This face detection cascade contains a total of 12 stages and 98 features. With 2 features per row, and 8 stages having an odd number of features, a total of 53 rows of memory are required (less than 32kB in total). The memory itself is initialized using another custom instruction (details omitted due to space).

Four of these dual-8-bit-lookups can fit into a 32-bit lane of a custom vector instruction. For example, in a 16-lane configuration, 128 table lookups are done every cycle.

Without the CVI, this table lookup requires approximately 15 regular vector instructions, several of which operate on 32-bit operands. The majority of the runtime was originally spent computing and checking the features of each stage. Even after restricting LBP block sizes and transforming this work into a precomputation, the LBP feature lookup table operation and stage pass/fail computation still occupies 57% of the runtime. Two of these table lookups (30 regular vector instructions) are reduced into a single custom vector instruction (CVI). Since the CVI operates only on 8-bit data, more parallelism is available than regular 32-bit vector instructions.

Due to the ILP formulation, the summation of 8-bit values is guaranteed to be sufficient without risk of overflow or rounding. This allows each 8-bit input pattern to produce an 8-bit output.

The final code of the innermost loop is presented in Figure 8. The scalar housekeeping is done in parallel with two vector operations: vector table-lookup, and vector add.

Since the LUT contents are not hardcoded, but loaded at runtime, this system can be used to detect different types of objects beyond faces. It can also be used in detection chains, e.g. first detect a face, then detect eyes within a face.

Adding the custom LBP LUT instruction to the masked vectorized algorithm results in an additional 4.2× speedup.

B. LBP Pattern Computation

After the table lookup is sped up with the first custom vector instruction, computation of the LBP patterns becomes the bottleneck. This is the process of computing the 8-bit comparison result for the center block in each 3×3 windowed region of the source image. This must be repeated for 1×1, 2×2 and 4×4 block sizes, producing three separate 8-bit arrays as output. Due to feature reuse at different offsets, this can be done once as a precomputation rather than on-the-fly as needed. Restricting the variety of LBP features (in terms of block sizes used) increases the frequency of reuse, and improves the advantage of the precomputation.

Since regular ALU operations have two inputs and one output, computing LBP patterns with 9-inputs and 1-output for a 3×3 window is not straight-forward. The fan-in is larger still for the larger block sizes.

To solve this, we use a stateful pipeline that processes columns of data, one vertical stripe at a time. The output stripe width is a set of 8-bit values that match the wavefront width of the SIMD execution unit. The output values on the left or right edges of the stripe must read pixels past the left edge or right edge. To accomplish this, the custom instruction is supplied with two overlapping rows as input operands: one operand includes the pixels past the left edge of the output stripe, while the other operand is offset enough to include pixels past the right edge of the output stripe. The custom vector instruction iterates one row at a time down the image. The custom instruction has three modes, one for each of the block sizes, and therefore requires three passes, before moving over to the next column. When starting the next column, a 2D DMA is performed to convert the vertical stripe in the source image into a packed image array in the internal scratchpad.

The custom vector operation can be separated into two stages. The first stage reduces (adds) the rows and columns of byte-sized image data according to the LBP block size. The second stage compares the center and 8 neighbours rows, separated by a specific stride, to produce the 8-bit LBP patterns.

Adding the CVI to precompute the LBP patterns results in an additional 2.4× speedup.

V. Results

A. Experimental setup

FPGA builds use Vivado 14.2. All designs are synthesized for 200MHz, with the reported worst-case negative slack used to compute Fmax.

Processing time is reported for one 320x240 image, producing an image pyramid with scale factor of 1.1, and stride of 1 to produce a dense scan. These default settings match those of Brousseau [3]. Generally, however, 1080p60 video can be consumed and produced, and processing can be sped up with larger scale factors and strides.

A full face-detection system was implemented with a 1080p HDMI video camera, FPGA board, and display. A photo of the display output is shown in Figure 1, where a test image is presented to the camera using an iPhone. In this case, 49 of 50 faces are detected in 36ms, despite the small face sizes and the angle of presentation.

B. Performance

Table II compares our performance, in frames per second, to prior work. Our performance is 1.5× to 6.8× faster than these pure hardware implementations. The last row of this table is a fixed-width SIMD CPU implemented as an ASIC. In these cases, we were able to set the image resolution, scaling factor, and stride to match the prior work.

The contributions of each optimization to performance are shown in Figure 9. Wavefront skipping and the LUT custom vector instruction boost performance the most. With all optimizations in place, diminishing returns with respect to vector length become apparent, showing a limit to our SIMD approach. For additional performance gains, a multi-core implementation would be required.

The FPGA resources required for the base system (containing only HDMI I/O) and vector engine are shown in Table III. As a rough comparison, the hardware face detection system developed by Brousseau [3] runs at 125MHz on a Stratix IV 530 (the largest available) and uses similar area to our largest configuration, but our work is 2.6 times faster, offers a complete end-to-end (camera to display) system, and the face detection is written in software.

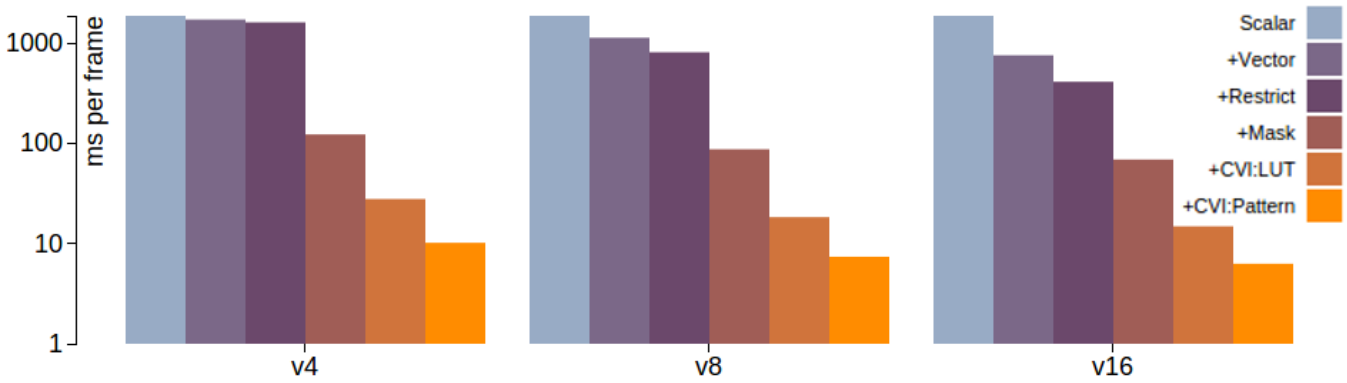
Prior Work	Feature Type	Platform	Image Res.	Prior (fps)	This (fps)	Speedup
Brousseau [3]	Haar	FPGA	320x240	50	159	2.6
Cho [4]	Haar	FPGA	320x240	61	159	2.2
			640x480	16	41	2.1
Gao [6]	Haar	FPGA	256x192	98	175	1.5
Bilaniuk [1]	LBP	ASIC	640x480	5	41	6.8

TABLE II: Results comparison

VI. Conclusions and Future Work

In this paper, we achieve a speedup of 248× over an ARM Cortex-A9 software implementation of LBP-based face detection. This gain can be decomposed into 25× speedup using software-only transformations that vectorize the application so it can run on a soft vector engine with 16 parallel 32-bit ALUs. Furthermore, an additional 10× speedup was updated by implementing just two custom vector instructions (CVIs): one for quickly computing the 8-way comparison, producing the 8-bit LBP, and another for the table-lookup operation.

Our hybrid software/hardware system is 1.6 to 6.8 times faster than previously published face detection systems implemented purely in hardware. Furthermore, our system uses only about 800 lines of VHDL code. Keeping the majority of the system in software makes overall development easier, permits easy integration with other processing algorithms,



Vector Lanes	Software Only				With Custom Instructions	
		Cumulative optimizations				
	Scalar	+Vector	+Restrict	+Masked	+CVI:LUT	+CVI:Pattern
4	1,885.4 (1.0×)	1,731.8	1,646.6	125.2 (15.1×)	28.0	10.3 (183.9×)
8	1,885.4 (1.0×)	1,150.4	823.6	88.5 (21.3×)	21.0	8.5 (222.9×)
16	1,885.4 (1.0×)	831.4	452.5	75.6 (24.9×)	18.0	7.6 (248.4×)

Fig. 9: Performance in milliseconds (speedup) on 320x240 image pyramid, 1.1 scale factor, unit stride

	# FF	# LUTs	Mem. (LUTs)	Mem. (BRAM)	# DSP	Fmax (MHz)
ZC706 Max.	437,200	218,600	70,400	545	900	-
video only	8,873	7,028	514	12	8	236

# Lanes	CVIs					
4	no	16,843	16,437	869	54.5	36	197
	yes	25,679	25,237	1,194	88.5	36	199
8	no	22,235	23,223	1,123	47	64	198
	yes	37,856	39,377	1,651	81	64	175
16	no	36,165	36,672	1,672	48	120	183
	yes	65,430	68,444	2,629	82	120	166

TABLE III: Resource usage and Fmax

and generally makes the platform more flexible and easier to maintain. It also allows the same hardware to be re-used for other types of image processing.

A novel contribution made in this work is the ILP formulation used to replace 32-bit floating-point or fixed-point computation with 8-bit integers. This simplifies hardware, produces a speedup, and provides an accuracy guarantee. With block sizes restricted to be square powers of two, we transformed the LBP computation into a precomputation problem. Although this has been reported before on fixed-width SIMD systems [1], we found that it works for variable-length vector systems as well.

A multi-core system should to be explored to further increase performance. Each core can work on a tile of an image, or at a separate level in the image pyramid.

We have only assessed our algorithmic changes using frontal face detection. For generality, we should validate with detection of other objects.

Acknowledgment

The authors would like to thank Aaron Severance and Joel Vandergriendt for their assistance and feedback, Xilinx for donating research licenses, and NSERC for funding this work.

References

[1] O. Bilaniuk, E. Fazl-Ersi, R. Laganiere, C. Xu, D. Laroche, and C. Moulder. Fast LBP face detection on low-power SIMD architectures. In *IEEE Computer Vision and Pattern Recognition Workshops*, pages 630–636, 2014.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] B. Brousseau and J. Rose. An energy-efficient, fast FPGA hardware architecture for OpenCV-compatible object detection. In *ICFPT*, pages 166–173, 2012.

[4] J. Cho, B. Benson, S. Mirzaei, and R. Kastner. Parallelized architecture of multiple classifiers for face detection. In *IEEE ASAP*, pages 75–82, 2009.

[5] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Alg. for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[6] C. Gao and S.-L. Lu. Novel FPGA based Haar classifier face detection algorithm acceleration. In *FPL*, pages 373–378, 2008.

[7] S. Liao, X. Zhu, Z. Lei, L. Zhang, and S. Z. Li. Learning multi-scale block local binary patterns for face recognition. *Advances in Biometrics*, pages 828–837, 2007.

[8] A. Severance, J. Edwards, and G. Lemieux. Wavefront skipping using BRAMs for conditional algorithms on vector processors. In *FPGA*, pages 171–180, 2015.

[9] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft vector processors with streaming pipelines. In *FPGA*, pages 117–126, 2014.

[10] A. Severance and G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *CODES+ISSS*, pages 1–10, 2013.

[11] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Computer Vision and Pattern Recognition*, volume 1, pages 511–518, 2001.