Face Detection on Embedded Systems

Abbas Bigdeli[1], Colin Sim[2], Morteza Biglari-Abhari[2] and Brian C. Lovell[1]

[1]Safeguarding Australia Program,
NICTA, Brisbane, QLD 4000, Australia
{abbas.bigdeli, brian.lovell}@nicta.com.au
[2]Department of Electrical and Computer Engineering
The University of Auckland, Auckland, New Zealand
{csim036, m.abhari}@auckland.ac.nz

Abstract. Over recent years automated face detection and recognition (FDR) have gained significant attention from the commercial and research sectors. This paper presents an embedded face detection solution aimed at addressing the real-time image processing requirements within a wide range of applications. As face detection is a computationally intensive task, an embedded solution would give rise to opportunities for discrete economical devices that could be applied and integrated into a vast majority of applications. This work focuses on the use of FPGAs as the embedded prototyping technology where the thread of execution is carried out on an embedded soft-core processor. Custom instructions have been utilized as a means of applying software/hardware partitioning through which the computational bottlenecks are moved to hardware. A speedup by a factor of 110 was achieved from employing custom instructions and software optimizations.

1 Introduction

The identification and localization of a face or faces from either an image or video stream is a branch of computer vision known as face detection [1, 2]. Face detection has attracted considerable attention over recent years in part due to the wide range of applications in which it forms the preliminary stage. Some of the main application areas include: human-computer interaction, biometrics, content-based image retrieval systems (CBIRS), video conferencing, surveillance systems, and more recently, photography.

The existing visual sensing and computing technologies are at a state where reliable, inexpensive, and accurate solutions for non-intrusive and natural means of human-computer interactions are feasible. Biometrics is an evolving application domain for face detection and is concerned with the use of physiological information to identify and verify a person's identity. In most cases, face recognition algorithms are designed to operate on images assumed to only contain frontal faces [2]. Therefore, face detection is required to first extract faces from an image prior to the recognition step. Examples of commercial biometric systems are BioID [3] and ViiSage[1]. HumanScan is the company that developed BioID; a multimodal system incorporating voice, lip movement and face recognition to

---

[1] www.viisage.com

authenticate a person. This system implements a model-based face detection algorithm based on the Hausdorff distance [4].

Another application area that can clearly benefit from face detection is surveillance systems that would allow easier identification of criminals in public spaces. Shan *et al* [5] presented a robust face recognition system specifically designed for Intelligent CCTV systems. Another video surveillance system which has the capacity to detect faces is that proposed by Kim *et al* [6]. More recently, FujiFilm[2] and Nikon Corporation[3] have incorporated face detection technologies into some of their camera series to automatically improve pictures taken under poor lighting conditions.

The majority of the research work to date has primarily focused on developing novel face detection algorithms and/or improving the efficiency and accuracy of existing algorithms. As a result, most solutions deployed (similar to the examples given above) are typically high-level software programs targeted for general purpose processors that are expensive and usually non-real-time solutions. Since face detection is typically the first step and frequently a bottleneck in most solutions due to the large search space and extensive amount of computationally intensive operations, it is reasonable to suggest an embedded implementation specifically optimized to detect faces. An embedded solution would entail many advantages including 1) low cost, as only a subset of hardware components are required compared to the general computer based solutions, 2) optimization of the face detection algorithms for real-time operations independent of face recognition or other post-processing concerns and 3) integration with other technologies such as security cameras to create smart devices.

Related Work

Now that reliable, accurate, and efficient face detection algorithms are available coupled with advances in embedded technologies; low-cost implementations of robust real-time face detectors can be explored. The most common target technologies are: pure hardware, embedded microprocessors, and configurable hardware.

Pure hardware systems are typically based on very large scale integrated circuit (VLSI) semiconductor technology implemented as application specific integrated circuits (ASIC). Compared to the other technologies, ASICs have a high operating frequency resulting in better performance, low power consumption, high degree of parallelism, and well established design tools. However, a large amount of development time is required to optimize and implement the designs. Also, due to the fixed nature of this technology the resulting solutions are not flexible and cannot be easily changed, resulting in high development costs and risk. Theocharides *et al* [7] investigated the implementation of a neural network based face detection algorithm in 160 nm VLSI technology based on algorithm proposed by Rowley *et al* in [8, 9], which has a high degree of parallelism.

On the other hand, software programs implemented on general purpose processors (GPP) offer a great deal of flexibility, coupled with very well established design tools that can automatically optimize the designs with little development time and costs. GPPs are ideally suited to applications that are primarily made up of control processing. However, they are disadvantaged because minimal or no special instructions are available to assist with data processing [10]. Digital signal processors (DSP) extend GPPs in the direction of increasing parallelism and providing additional support for applications requiring large amounts of data processing. The drawbacks of microprocessors (both GPPs and DSPs) are

---

[2] www.fujifilmusa.com
[3] www.nikonimagings.com

high power consumption, and inferior performance compared to an ASIC. The performance of the final solution is limited to the selected processor.

Finally, configurable platforms such as field programmable gate arrays (FPGA) combine some of the advantages from both pure hardware and pure software solutions. More specifically, the high parallelism and computational speed of hardware, and the flexibility and short design time of software. By inheriting characteristics from both hardware and software solutions, the design space for FPGAs is extended for better trade-offs between performance and cost. These design trade-offs are far superior to that of pure hardware or software solutions alone. From an efficiency point of view, the performance measures for FPGAs, that is, operating frequency, power consumption, and so on, are generally half way in between the corresponding hardware and software measures.

Several configurable hardware based implementations exist, including that by McCready [11] and Sadri *et al* [12]. McCready specifically designed a novel face detection algorithm for the Transmogrifier-2 (TM-2) configurable platform. The Transmogrifer-2 is a multi-board FPGA based architecture proposed by Lewis *et al* [13]. The algorithm was intentionally designed with minimal mathematical operations that could execute in parallel — engineering effort has been put in to reduce the number of multiplications required. The implemented system required nine boards of the TM-2 system, requiring 31,500 logic cells (LC). The system can process 30 images per second with a detection accuracy of 87%. The hardware implementation is said to be 1,000 times faster than the equivalent software implementation.

On the other hand, Sadri *et al* [12] implemented the neural network based algorithm proposed by Rowley *et al* [8] on the Xilinx Virtex-II Pro XC2VP20 FPGA. Skin color filtering and edge detection is incorporated to reduce the search speed. The solution is partitioned such that all regular operations are implemented in hardware while all irregular control based operations are implemented on Xilinx's embedded hardcore PowerPC processor. This partitioning allows the advantages of both hardware and software to be simultaneously exploited. The system operates at 200 MHz and can process up to nine images per second.

The examples presented illustrate the obvious compromises between accuracy and algorithm robustness versus the amount of resources required. That is, to improve the performance of the face detection algorithms, we must either increase the embedded design complexity, which generally results in higher power consumption and hardware costs, or settle for a lesser solution.

2. PC Based Software Prototype

The initial software prototype of the standard Viola-Jones algorithm was implemented based on the trained classifiers provided in the Open Computer Vision Library (OpenCV)[4]. The particular classifiers used in this implementation are those trained for a base detection window size of 24x24 pixels. The classifiers are trained to detect upright frontal faces. These classifiers were created and trained by Lienhart *et al* who used a total of 8,000 training samples, of which 5,000 are face images and 3,000 non-face images [14]. The accuracy of the implemented Viola-Jones face detection algorithm was validated using a subset of images from the CMU + MIT face databases, and images retrieved from a random web crawl.

---

[4] www.opencv.org

To ensure that the results obtained can be compared with other published sources, detection and false detection events use the definition of Lienhart *et al* [14], as follows. A detection window is said to correctly identify a face if the following criteria are satisfied:

 i) The maximum displacement between the centre of the detected window and the actual face do not exceed 30% of the actual face size.
 ii) The difference between the detected window and actual face size does not exceed 50% of the actual face size.

At $\Delta = 1.0$ and $s = 1.25$[5], the detection accuracy for the CMU+MIT image set is 80%. This result is consistent with that presented by Lienhart *et al* [14]. On the other hand, the detection accuracy for the web crawl image set is 92%. The slightly better results obtained in the web crawl image set is possibly due to the following factors:

 i) The images from the web crawl image set had a higher resolution and better quality
 ii) The images' backgrounds were less complex, resulting in less misclassification.

2.1 Implementation Details

In order to standardize the input data for all implementation platforms so that the performance results can be benchmarked, a set of 10 images arbitrarily chosen from a web search is used as input. Each image contains a single frontal face. Multiple images are used so that any variability in the processing time for individual images is better averaged out. All the images are stored as grayscale bitmaps of size 576x720 pixels — image size being arbitrarily chosen. Variability in execution times is primarily attributed to the cascaded nature of the Viola-Jones algorithm. The amount of time required to search an image for a face is closely related to the "complexity" of the image; that is, if an image has large areas that do not contain faces but passes many of the cascaded classifier stages, then more time is required to process the image. Other aspects that contribute to varying the execution times are platform dependent and include: cache and memory access times, pipeline schedule, interrupt mechanisms, and so on.

The software prototype was developed in C and compiled with a GNU GCC compiler under Cygwin. The compiler was set to highest optimization for speed, and the PC was a Pentium 4 processor, 3.20 GHz, with 1.0 GB RAM with Windows XP as the operating system.

2.2 Software Implementation Results

All 10 faces within the 10 images were located with two false positives. Due to the small number of images, the ratio between detection and false detection rates may be skewed. The total execution time for the program to process all 10 images is 19.91 seconds. A breakdown of each function and their contribution to the final time are presented in the **Table 1.**

Not all functions used in the program are in **Table 1**; this is because the profiling process is based on samples taken when the program is running. As a result, functions that execute quickly may not be acknowledged. An arrow before a function name indicates that it is called from another function where the function that called it has one less indentation level. As an example, referring to **Table 1**, `InitialiseClassifierStages`, `UpdateClassifierStages`, and so on are all called from the `FaceDetection` function. As seen in the flat profile, the `RunClassifierCascade` function consumes the majority of the execution time. This function is responsible for checking if a sub-

---

[5] Where s is the scale factor in the search algorithm and, Δ represents the number of pixels to shift the detection window [14]

window contains a face. The function itself is executed in close to no time; the long processing time is due to the accumulation of over 7.5 million calls. The number of calls to RunClassifierCascade is related to the number of search locations which in turn is proportional to the size of the image.

Table 1. Flat profile of the Viola-Jones algorithm.

Function Name	Total Time Taken (sec)	Number of Calls	Average Time (sec)	Percentage of Total Time (%)
CalculateIntegralImage	0.35	10	0.04	1.76
CalculateSqIntegralImage	0.03	10	0	0.15
FaceDetection	19.53	10	1.95	98.09
→ InitialiseClassifierStages	0.02	10	0	0.1
→ UpdateClassifierStages	0.02	150	0	0.1
→ RunClassifierCascade	19.46	7,458,250	0	97.74
→ round	0.01	Unknown	N/A	0.05
→ AddSubWindow	0	110	0	0
→ GetFaceWindows	0	10	0	0
FreeIntegralImage	0	20	0	0
FreeFaceWinList	0	10	0	0
FreeSubWinList	0	10	0	0

2.3 Initial Embedded System and Port of Software Code

A fully functional embedded system based on Altera Nios II softcore processor was created on a Stratix FPGA development board. Overall the maximum system operating frequency is 96.42 MHz. However, due to the limitations of the possible frequencies that can be generated by the PLL module, the maximum feasible operating frequency without violating timing constraints is 80 MHz. From this point onwards unless otherwise stated the actual operating frequency for each system is 80 MHz.

Given that the code implemented on the PC is based on GNU libraries which is the same as the software development in the Nios II environment, it was possible to be directly ported across to the Nios II system without many changes.

2.4 Profile of Embedded Code

Initially, the GNU profiler was used to profile the code which was compiled with maximum speed optimization. The original code runs quite slowly. The lengthy processing time is attributed to the extensive number of locations searched within each image. The amount of searching is directly proportional to the input image size and the step size of the detection window. There are several methods available to help reduce the search space, common approaches being, image differencing and skin color modeling. Image differencing as its name suggests involves taking the difference between image frames. Since humans are generally the only objects that move within a scene, searching can be concentrated on the sections that are changing. On the other hand, the skin color modeling method makes use of statistical models of human skin to filter the input image hence the search for faces can then be restricted to those areas. An alternative method is to reduce the size of the input images using an image interpolation algorithm. Once the faces are found in the smaller image, the location and dimension of the detection windows can be rescaled to the size of the original image.

The most popular techniques for enlarging or shrinking images are nearest neighborhood, bilinear, and bicubic down-sampling. Of these methods, bicubic offers the best results in terms of sharpness and preservation of image details. Every point in the

resulting image is calculated from a weighted average of 16 pixels adjacent to the corresponding pixel in the original image. The bicubic interpolation algorithm can be expressed as the following set of equations:

$$g(x) = \sum_k c_k u(s_k) \qquad u(s) = \begin{cases} A_1|s|^3 + B_1|s|^2 + C_1|S| + D_1 \\ A_2|s|^3 + B_2|s|^2 + C_2|S| + D_2 \\ 0 \end{cases} \qquad \textbf{(1)}$$

Bicubic interpolation function and its kernel [15].

Where g is the interpolation function, c is points in the original image, u is the interpolation kernel, and s is the distance between the pixel of interest and its neighboring pixels. As noted in [15], the accuracy and efficiency of the algorithm lies solely on the interpolation kernel, u. It was empirically found that the smallest possible scale factor without compromising the detection accuracy of the 10 input images is 0.25. All the faces in the original image were detected with no false positives. The performance report after applying bicubic down-sampling is given in **Table 2**.

Table 2. Performance report after applying bicubic down-sampling.

Section	Total Time (sec)	Occurance	Average Time (sec)	Percentage of Total Time (%)
Imresize	84.74	10	8.47	7.46
CalculateIntAndSqIntImages	0.16	10	0.02	0.01
FaceDetection	1050.27	10	105.03	92.52
→ InitialiseClassifierStages	0.19	10	0.02	0.02
→ UpdateClassifierStages	49.14	90	0.55	4.33
→ RunClassifierCascade	1000.78	315,100	0.003	88.16
Average Time Per Image			113.52	

The down-sampling calculations are carried out in the `Imresize` function. As seen in the performance report, the size reduction of the input image has a positive performance affect on all functions that carry out operations on the image. On average, a speedup factor of 16.5 is achieved after down-sampling the input image to a quarter of its original size.

An additional optimization applied was to combine the `CalculateIntegralImage` and `CalculateSqIntegralImage` functions together. This is a logical step given that these functions have a similar structure and operate on the same data. The resulting function name is `CalculateIntAndSqIntImage`, and completes execution in approximately half the time originally required to calculate the integral and square integral images separately. The reduction in time is a result of not having to fetch the same image data from extended memory (SDRAM) twice.

3. Optimization Using Custom Instructions

Configurable custom processors are becoming an ever more popular implementation technology of choice for addressing the demands of complex embedded applications. Unlike traditional hardwired processors that consist of a fixed instruction set from which application code is mapped; configurable processors can be augmented with application specific instructions, implemented as hardware logic to optimize bottlenecks. This lends towards a method for hardware-software partitioning whereby the efficiency of hardware and the flexibility of software are integrated.

There are a number of benefits in extending a configurable processor with custom instructions. First, transparency; the added custom instructions will improve the performance of the tasks for which they are designed with minor changes to the original code. Second, rapid development; there is a wide variety of off-the-self configurable cores that could be used as a base for development. Additional instructions could be integrated into the processor core as the need to extend its computational capabilities arises. Finally, low-cost access to domain specific processors; generally the fundamental characteristics of an application area is similar. These characteristics can be summarized as a set of instructions and applied to a variety of similar applications, for example, multimedia applications [16].

Unfortunately, there are two minor drawbacks to using custom instructions. For one, additional hardware is required, although this is becoming less of an issue as embedded technologies become more economical. Secondly, as the custom instructions are directly integrated into the processor's pipeline, the maximum operating frequency may be degraded if the instruction is poorly designed. Adding custom instructions is a proven optimization technique that has been applied to a wide range of embedded applications. Some published examples include, embedded real-time operating systems (RTOS) [17], biometrics [18], and multimedia [19].

3.1 Custom Instruction Design Flow

The design flow for identifying and integrating custom instructions into configurable processors is summarized in **Figure 1**. This is a generic framework that could be applied to any application. Firstly, the software code is profiled to reveal bottlenecks that could be alleviated with the introduction of custom instructions. Once the hardware module for the instruction is implemented and tested, it is added to the processor and the whole system is regenerated. Then the software code is updated to make use of the new instructions. Finally, the functionality of the system is verified to ensure bugs are not introduced with the new instruction. This process is repeated until either the performance requirements or resource limits are met.

3.2 Extending Nios II with Custom Instructions

All of the Altera Nios II processor cores are designed to support up to 256 new custom instructions. The logic for the new instructions is directly connected to the arithmetic logic unit (ALU), as illustrated in **Figure 2**. In the face detection application, this comes does to applying the custom instructions to the Viola-Jones algorithm The Viola-Jones application code which includes the initial software optimizations is profiled using performance counters. As expected, the `FaceDetection` function, more specifically, the `RunClassifierCascade` function is the most time consuming. It should be noted that it is not the `RunClassifierCascade` function itself that is time consuming, but due to the large number of times it is called, the accumulated time is large. Hence, the overall processing time could be improved if either the number of calls to `RunClassifierCascade` or its execution time is reduced. Since reducing the number of calls to `RunClassifierCascade` is going to compromise the accuracy of the face detector, the focus is placed on reducing the execution time of the function itself. Any time savings made in each function call will correspond to a large overall saving.

The main operations carried out in the `RunClassifierCascade` function are addition and multiplication of integer values associated with the calculation of indexes into the input image. There are also floating point multiply and addition operations that are required for image normalization and weighting of the classifier features. Since all the

integer related instructions are already implemented in hardware, it is believed that there are opportunities to improve the performance of the floating point operations through custom instructions.
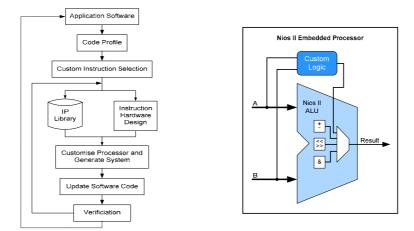

Fig. 1. Custom instructions design flow.

Fig. 2. Connection of custom instruction logic with the Nios II ALU.

Floating point multiply is the first operation of interest as it is used most frequently. Further profiling of the `RunClassifierCascade` function indicates that this operation takes up roughly 559 seconds of the total 1135 seconds, approximately 49% of the total processing time. By default all floating point arithmetic are emulated in software. For these reasons the first operation to be implemented as a custom instruction is the floating point multiply. The next instruction that was implemented was floating point add/subtract Instructions. A summary of the performance reported for the face detector after the addition of the floating point multiply instruction and add/subtract Instruction are presented in **Table 3** and **Table 4** respectively.

Table 3. Performance with the use of the floating point multiply custom instruction.

Section	Total Time (sec)	Occurance	Average Time (sec)	Percentage of Total Time (%)
Imresize	56.5	10	5.65	9
CalculateIntAndSqIntImages	0.16	10	0.02	0.03
FaceDetection	571.17	10	57.12	90.98
→ InitialiseClassifierStages	0.19	10	0.02	0.03
→ UpdateClassifierStages	36.62	90	0.41	5.83
→ RunClassifierCascade	534.03	315,100	0.002	85.06
Average Time Per Image			62.79	

Table 4. Performance with the use of the floating point addition/subtraction custom instruction.

Section	Total Time (sec)	Occurance	Average Time (sec)	Percentage of Total Time (%)
Imresize	57.46	10	5.75	12.73
CalculateIntAndSqIntImages	0.16	10	0.02	0.04
FaceDetection	393.61	10	39.36	87.23
→ InitialiseClassifierStages	0.19	10	0.02	0.04
→ UpdateClassifierStages	34.78	90	0.39	7.71
→ RunClassifierCascade	358.43	315,100	0.001	79.43
Average Time Per Image			45.13	

3.4 Optimization of the `Imresize` Function

The next most time consuming function to focus on is `Imresize`. Since the bicubic resizing algorithm used in the `Imresize` function is not specific to the Viola-Jones algorithm and could also be applied to many other applications, it is an ideal candidate for optimization. As it stands, the `Imresize` function accounts for 13% of the total time.

By examining the bicubic interpolation algorithm more closely, it becomes evident that the coefficients calculated by the interpolation kernel can be fixed constants. More importantly, the exact value of these constants is not essential [20]. A graphical illustration for computing an interpolated value is depicted in **Figure 3**. In essence, the two dimensional convolution like calculations described in Equation 1 can be decomposed to five sets of one dimensional operations. The coefficients, *a0*, *a1*, *a2*, and *a3*, as seen in **Figure 3** all add to one; hence, the intermediate sums will never exceed the largest pixel value of 255. Also, the intermediate results after multiplication are truncated to 8-bit integer values.
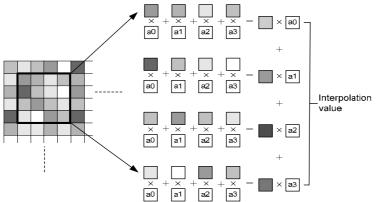

Fig. 3. Graphical description for calculating an interpolation value.

A series of experiments were conducted using the original implementation of the bicubic interpolation algorithm to look at the behavior of *a0*, *a1*, *a2* and *a3* for a variety of different scale factors. The following observations were made:

- *a0* and *a3* have the same values
- *a2* is roughly four times larger than *a0* and *a3* while *a1* is roughly two times larger than *a0* and *a3*
- when the coefficients are set to the same value and are close to zero, the resulting image is very dark (basically black)
- when the coefficients are set to the same value and close to one, the resulting image is extremely noisy

Based on these observations, the chosen values for *a0*, *a1*, *a2*, and *a3* are 0.125, 0.5, 0.25, and 0.125, respectively. These coefficients are chosen because they correspond to dividing the pixel values by 2, 4, and 8, that is, shifting of the pixel values to the left by 1, 2, and 3 bits, respectively. The images produced based on the new implementation of the `Imresize` function using these coefficients is identical to the original implementation and that from Matlab — Matlab's implementation is used as an additional confirmation step. The `Imresize` function that utilizes the coefficients, *a0*, *a1*, *a2*, and *a3* is referred to as the "new implementation", while the implementation that makes use of the bicubic kernel is referred to as the "original implementation".

(a)   (b)   (c)

Fig. 4. Examples of images resized to a quarter of their original size. The methods used are: (a) original implementation, (b) new implementation, and (c) Matlab's bicubic resize function.

The performance of the original and new implementation of the `Imresize` function is summarized in **Table 5**; the values obtained are averaged across all 10 face images. The new implementation is 115 times faster than the original implementation.

Table 5. Performance summary of the original and new implementation of the `Imresize`.

Section	Average Clock Cycles	Average Time (sec)
Original Imresize	459,693,061	5.75
New Imresize	3,949,404	0.05

4. The Effect of Data and Instruction Caches on Performance

It is also important to investigate what the effects data and instruction caching behavior and size have on performance. Initially, the Nios II processor is configured with the default data cache settings, that is, an on-chip memory size of 16 KB with a data cache line size of 4 bytes. According to the Nios II core documentations, if the line size is greater than 4 bytes, data retrieval from extended memory (SDRAM in our case) is pipelined; hence reducing the impact of data transfer latency. A new system with exactly the same configuration but with the data cache line size increased to 32 bytes is generated. When the face detector and face detector beta programs are ran on this system, the total execution times are 466.23 and 394.18 seconds, respectively – the difference between the two execution times are less. These results give a positive indication that the processor's caches have an influence on performance.

The next experiment is to look at the effects of altering the size of the data cache. A series of eight systems with all the possible data cache sizes are generated. All these systems have a data cache line size of 32 bytes and the instruction cache size is fixed to 4 KB. **Table 6** summarizes these results. The usage of other resources such as DSP blocks, PLLs, and pins remain the same. There are minor fluctuations in the amount of LEs used and operating frequencies, but this is likely due to the variability in optimizations by the synthesis and fitting tools. Logically, the total amount of memory bits utilized linearly scales with the data cache size. As seen in **Table 6**, the size of the data cache does have an affect on the performance of the programs, particularly in the size range from 0.5 to 16 KB. Also, the performance continues to improve with a larger data cache.

A similar system but with data and instruction cache sizes of 64 KB (the largest cache sizes possible) were also generated on a Stratix EP1S40 development board (with roughly four times more resources than EP1S10) to confirm that no further improvements were possible with an instruction cache larger than 16 KB. Similarly other computational functions including Divide, Compare, and Round were added as custom instructors and similar speed-ups were observed.

Table 6. Summary of the resource usage and executions times for varying data cache sizes.

Data Cache Size (KB)	LE (%)	Total Memory Bits (%)	Face Detector (sec)	Face Detector Beta (sec)
0.5	66	5.65	589.26	483.07
1	68	6.13	555.3	452.11
2	66	7.07	527.24	433.59
4	66	8.96	507.48	421.03
8	66	12.71	481.64	406.13
16	68	20.19	466.23	394.18
32	68	35.1	458.8	389.05
64	68	64.8	455.79	386.89

5. Conclusion

This paper investigated the effects of replacing software bottleneck operations of a Face-Detection System based on Viola-Jones algorithm with custom instructions on performance. **Table 7** presents a summary of the new instructions implemented along with a measure of their efficiency — in-order for comparisons to be made fairly, the floating point multiply custom instruction is re-synthesized without the use of DSP blocks.

Table 7. Speedup, resource usage, and efficiency measure for each custom instruction.

Floating Point Operation	Resource (LE)	Speedup	Speedup/Area (10^{-3})
Multiply	1,019	18	18
Add/Sub	806	21	26
Divide	1,061	11	10
Compare	77	16	208
Round	354	37	105

These results indicate that the floating point compare custom instruction is by far the most efficient in terms of speedup to area, even though it has a low overall speedup factor when integrated with the face detector application. On the other hand, even though the floating point multiply instruction has a reasonably low speedup to area ratio, when used in the face detector application the speedup for this instruction is high, in part because it is one of the most commonly used operations.

As the Viola-Jones face detection algorithm is primarily dominated by control operations and calculations involving 32-bit integer or floating point numbers, very little benefit is likely to result from the movement of larger functions to hardware.

An inadvertent result revealed through this investigation is that both the size and behavior of the caches, specifically the instruction cache, has a significant affect on the software performance. Experiments have shown that the total execution time may noticeably fluctuate depending on the code or instruction cache size. The implication of this result is that, it is difficult to determine the effectiveness of the optimizations applied — even with custom instructions; since changes to the software code results in a change to the code size and hence caching behavior. Lastly, it has been shown that incremental changes to the software code can add up to substantial reductions in the total execution time. However, the extent and effectiveness of these optimizations is largely attributed to the designer's experience.

References

1. M. H. Yang, D. J. Kriegman, and N. Ahuja: Detecting faces in images: a survey. In IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 24 (2002) 34-58
2. E. Hjelmas and B. K. Low: Face detection: a survey. In Computer Vision and Image Understanding, vol. 83, (2001) 236-274
3. R. W. Frischholz and U. Dieckmann: BiolD: a multimodal biometric identification system. In Computer, vol. 33 (2000) 64-68
4. D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge: Comparing images using the Hausdorff distance. In IEEE Transactions on Pattern Analysis and Ma-chine Intelligence, vol. 15 (1993) 850-863
5. Ting Shan, Brian C. Lovell, Shaokang, Chen and Abbas Bigdeli: Reliable Face Recognition for Intelligent CCTV. In Proc. of Safeguarding Australia 2006- The 5th Homeland Security Summit & Exposition (2006) 356-364
6. T.-K. Kim, S.-U. Lee, J.-H. Lee, S.-C. Kee, and S.-R. Kim: Integrated approach of multiple face detection for video surveillance. In Proc. of Int. Conf. on Pattern Recognition, vol. 2 (2002) 394-397
7. T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin, and W. Wolf: Embedded hardware face detection. In Proc. of the 17th Int. Conf. on VLSI Design (2004) 133-138
8. H. A. Rowley, S. Baluja, and T. Kanade: Neural network-based face detection. In IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 20 (1998) 23-38
9. H. A. Rowley, S. Baluja, and T. Kanade: Rotation invariant neural network-based face detection. In IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (1998) 38-44
10. B. D. T. Inc.: Using General-Purpose Processors for Signal Processing. In ARM Developers' Conf. (2004)
11. R. McCready: Real-Time Face Detection on a Configurable Hardware System. In Proc. of The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications (2000) 157-162
12. M. S. Sadri, N. Shams, M. Rahmaty, I. Hosseini, R. Changiz, S. Mortazavian, S. Kheradmand, and R. Jafari: An FPGA Based Fast Face Detector. In Global Signal Processing Expo and Conf. (2004)
13. D. M. Lewis, D. R. Galloway, M. Van Ierssel, J. Rose, and P. Chow: The Transmogrifier-2: a 1 million gate rapid-prototyping system. In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6 (1997) 188-198
14. R. Lienhart, A. Kuranov, and V. Pisarevsky: Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection. In DAGM, 25th Pattern Recognition Symposium (2003) 297-304
15. R. Keys. Cubic convolution interpolation for digital image processing. In IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 29 (1981) 1153-1160
16. A. Bigdeli, M. Biglari-Abhari, S. H. S. Leung, and K. I. K. Wang: Multimedia extensions for a reconfigurable processor. In Proc. of 2004 International Symposium on Intelligent Multimedia, Video and Speech Processing, (2004) 426-429
17. T. F. Oliver, S. Mohammed, N. M. Krishna, and D. L. Maskell: Accelerating an embedded RTOS in a SoPC platform. In Proc. of TENCON Conference, vol. 4 (2004) 415-418
18. H. Tsutsui, T. Masuzaki, T. Izumi, T. Onoye, and Y. Nakamura: High speed JPEG2000 encoder by configurable processor. In Proc. of Asia-Pacific Conf. on Circuits and Systems, vol. 1 (2002) 45-50
19. Z. GuangWei and L. Xiang: An efficient approach to custom instruction set generation. In IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (2005) 547-550
20. W. H. Press, W. T. Vetterling, S. A. Teukolsky, and B. P. Flannery: Numerical recipes in C: the art of scientific computing. Second ed. Cambridge University Press, New York (1992)