Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 21724, 8 pages
doi:10.1155/2007/21724

Research Article

Real-Time Video Convolutional Face Finder on Embedded Platforms

Franck Mamalet, Sébastien Roux, and Christophe Garcia

France Telecom Research and Development Division, 28 Chemin du Vieux Chêne, 38243 Meylan, France

Received 27 April 2006; Revised 19 October 2006; Accepted 26 December 2006

Recommended by Dietmar Dietrich

A high-level optimization methodology is applied for implementing the well-known convolutional face finder (CFF) algorithm for real-time applications on mobile phones, such as teleconferencing, advanced user interfaces, image indexing, and security access control. CFF is based on a feature extraction and classification technique which consists of a pipeline of convolutions and subsampling operations. The design of embedded systems requires a good trade-off between performance and code size due to the limited amount of available resources. The followed methodology copes with the main drawbacks of the original implementation of CFF such as floating-point computation and memory allocation, in order to allow parallelism exploitation and perform algorithm optimizations. Experimental results show that our embedded face detection system can accurately locate faces with less computational load and memory cost. It runs on a 275 MHz Starcore DSP at 35 QCIF images/s with state-of-the-art detection rates and very low false alarm rates.

1. INTRODUCTION

When embedding new services on mobile devices, one of the largest constraints is the limited computational resources. Low memory capacities, low CPU frequency, and lack of specialized hardware such as a floating-point unit are some of the major differences between a PC and an embedded platform. Unfortunately, advanced algorithms are usually developed on a PC without any implementation restriction in mind. Thus, embedding applications on power constraint systems is a challenging task and requires strong algorithmic, memory, and software optimizations.

Advanced user interface, security access control, model-based video coding, image and video indexing are some of the applications that rely on face detection. In recent years, numerous approaches for face detection have been proposed. An interesting survey was published by Yang et al. [1]. Face detection techniques can be classified in three main categories:

(i) feature invariant approaches [2, 3],
(ii) template matching methods [4, 5],
(iii) appearance-based methods [6, 7].

A recent technique belonging to the third category, called convolutional face finder (CFF) has been introduced by Garcia and Delakis [8] which provides the best performance on standard face databases. CFF is an image-based neural network approach that allows robust detection, in real world images, of multiple semifrontal faces of variable size and appearance, rotated up to ±20 degrees in image plane and turned up to ±60 degrees.

Recently, Tang et al. [9] have considered both face detection performance and implementation on embedded systems for cascade AdaBoost classifiers [10] on ARM-based mobile phones. The AdaBoost technique was also used in [11] for implementing a hybrid face detector on a TI DSP. Another way to achieve resource constrained implementation is to design hardware dedicated to face detection. In [12], the authors proposed an ASIC implementation of the face detector introduced by Rowley et al. [13].

However, real-time embedded implementations often require a trade-off between high detection rates, fast run time, and small code size. In most cases, the side effect of embedding a face detector is the reduction of the algorithm efficiency. We achieved both efficiency and speed objectives with our CFF implementation.

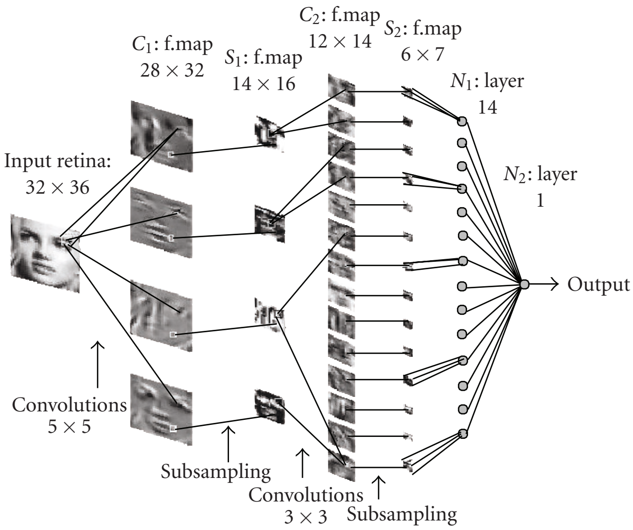

Figure 1: Convolutional face finder pipeline.

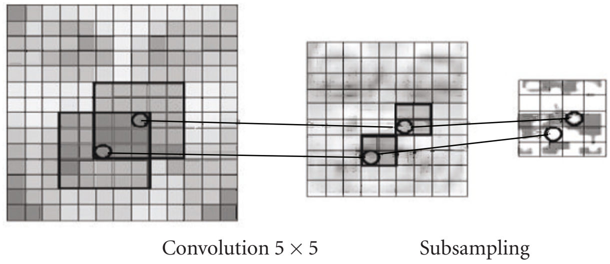

Figure 2: Receptive fields for convolution and subsampling for a feature map of layers C_1-S_1.

Figure 3: Face detection block diagram.

The remainder of this paper is organized as follows: an overview of the convolutional face finder technique is given in Section 2. Section 3 presents the methodology used for embedding such an algorithm. Section 4 details this methodology on the CFF case study. Experimental results for DSP- and RISC-based platforms are provided in Section 4. Finally, conclusions and perspectives are drawn in Section 5.

2. CFF ALGORITHM OVERVIEW

The convolutional face finder was presented in [8] and relies on convolutional neural networks (CNN) introduced and successfully used by LeCun et al. [14]. It consists of a pipeline of convolutions and subsampling operations (see Figure 1). This pipeline performs automatic *feature extraction* in image areas of size 32 × 36, and *classification* of the extracted features, in a single integrated scheme.

The convolutional neural network, shown in Figure 1, consists of a set of three different kinds of layers. Layers C_i are called convolutional layers, which contain a certain number of planes. Layer C_1 is connected to the retina, receiving the image area to classify as face or non-face. Each unit in a plane receives input from a small neighbourhood (biological local receptive field) in the planes of the previous layer. Each plane can be considered as a feature map that has a fixed feature detector corresponding to a pure convolution with a trainable mask, applied over the planes in the previous layer. A trainable bias is added to the results of each convolutional mask. Multiple planes are used in each layer so that multiple features can be detected.

Once a feature has been detected, its exact location is less important. Hence, each convolutional layer C_i is typically followed by another layer S_i that performs local averaging and subsampling operations. More precisely, each layer S_i output data is the result of the average of four input data in C_i, (see Figure 2) multiplied by a trainable coefficient, added to a trainable bias, and passed through a hyperbolic tangent function, used as an activation function. This subsampling operation reduces the dimensionality of the input by two and increases the degrees of invariance to translation, scale, and deformation of the learnt patterns.

In CFF, layers C_1 and C_2 perform convolutions with trainable masks of dimensions 5 × 5 and 3 × 3, respectively. Layer C_1 contains four feature maps and therefore performs four convolutions on the input image. Layers S_1 and C_2 are partially connected. Mixing the outputs of feature maps helps in combining different features, thus in extracting more complex information. Layer C_2 has 14 feature maps. Each of the four subsampled feature maps of S_1 is convolved by two different trainable masks 3 × 3, providing eight feature maps in C_2. The other six feature maps of C_2 are obtained by fusing the results of two convolutions on each possible pair of S_1 feature maps.

Layers N_1 and N_2 contain simple sigmoid neurons. The role of these layers is to perform classification after feature extraction and input dimensionality reduction are performed. In layer N_1, each neuron is fully connected to exactly one feature map of layer S_2. The unique neuron of layer N_2 is fully connected to all the neurons of layer N_1. The output of this neuron is used to classify the input image as face (positive answer) or nonface (negative answer). All parameters (convolution kernels, subsampling coefficients, biases) have been learnt automatically using a modified version of the back-propagation algorithm with momentum, from a large training set of faces [8].

As depicted in Figure 3, the process of face detection with CFF is performed in two steps. The first one can be considered as a coarse detection and returns positive responses to gather face candidate positions. The second one called fine detection performs a refined search in an area around each face candidate found in the first step.

In [8], the authors present both the training methodology to learn the coefficients, and the face localization process when training is completed. In this paper, we will focus on the face localization process. Figure 4 presents in detail the steps of this face localization process.

(i) Coarse detection is processed as follows: CFF is applied on a pyramid of scaled versions of the original image (see Figure 4-1) in order to handle faces of different sizes: each scale produces a map of face candidates (see Figure 4-2) which is fused back to the input image resolution and produces clusters of positive answers (see Figure 4-3). For each cluster, a representative face is computed as the centroid of its candidate face centers and sizes, weighted by their individual network responses.

(ii) Fine detection takes those candidates as input and locally applies CFF on a small pyramid around the face candidate center position (see Figure 4-4). The volume of positive answers is considered in order to take the classification decision, that is, face or non-face (see Figure 4-5). Finally, overlapping candidates are fused to suppress multidetection of the same face.

The acronym CFF will be used either for the detection pipeline or for the entire algorithm.

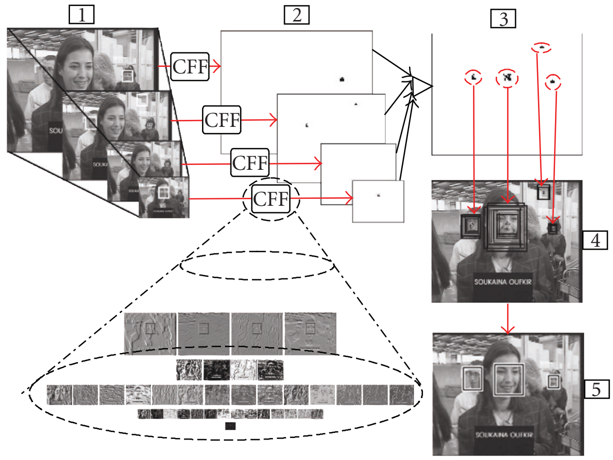

Figure 4: The different steps of the process of face localization.

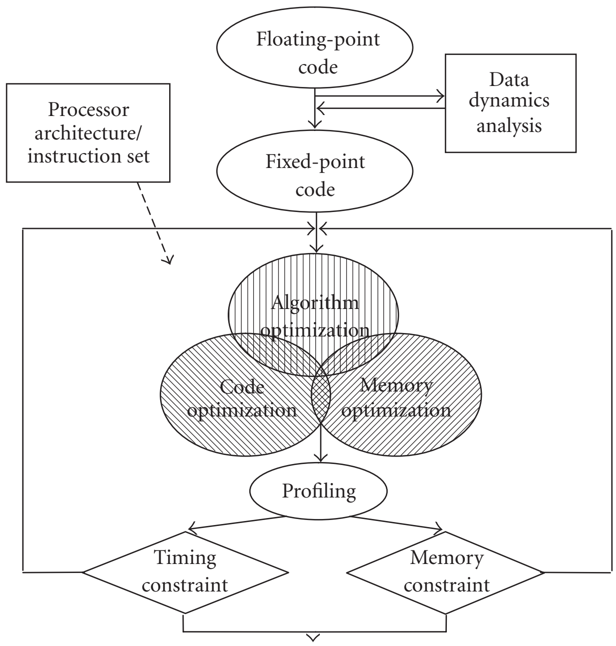

Figure 5: Diagram of followed methodology.

3. PORTING CFF TO EMBEDDED PLATFORMS: MAIN ISSUES AND METHODOLOGY

In order to implement complex algorithms on an embedded target processor, compilers are the tools used to optimize the instructions flow. In the last decade, many research activities have been carried out on instructions flow optimizations [15] and optimizing compilers [16], and some have led to industrial products such as the Metrowerks compiler for SC140 [17, 18]. However, compilers can only cope with the instructions flow optimization and parallelization. Even if these compilers mostly avoid human assembly programming, they only deal with local optimizations and many optimizations still need to be carried out by high-level code rewriting.

Other tools enable us to deal with these high level optimizations. First of all, high level profiling tools such as VTune software [19] are dedicated to pointing out the most consuming parts of the code using on-target time-sampling simulations. Also, memory accesses analysis tools [20] can be used to identify memory access bottlenecks. Some recent works try to propose semiautomatic tools for data transfer and storage optimizations [21]. This work relies on the following methodology which is only driven by high-level code profiling; further investigation will be carried out to automate our optimization process.

Our approach is based on iterations of high-level code optimizations and profiling to focus firstly on the most consuming functions of CPU resources. When dealing with an algorithm such as CFF, the first step towards embedded implementation is to avoid floating-point calculation. This step is called data optimization in [22] and is achieved thanks to a fractional transformation in accordance with data dynamics and processors data path. This also requires a strong verification of the accuracy of these transformations which can otherwise lead to incorrect results. The next steps of the methodology are iterations of a tri-optimization flow (code, memory, and algorithm) controlled by an on-target profiling (see Figure 5).

Profiling tools depend on the target platform: for instance, we use the VTune software [19] on an Xscale-based platform to profile the compiled code directly on target, and global timing information to evaluate the speed-up factor after each optimization iteration.

Table 1: Results of CFF on different test sets for the floating- and fixed-point versions.

	Faces size	36 to 300 pixels high				18 to 300	
	Threshold	10		17		17	
		Detection rate (%)	False alarms	Detection rate (%)	False alarms	Detection rate (%)	False alarms
Floating-point version	CMU	84,89	6	80,12	0	87,99	2
	CINEMA*	87,32	8	82,97	1	82,97	4
	WEB*	87,98	2	83,97	0	91,98	2
	ATT	97,25	0	96,50	0	96,75	0
	Total	89,20	22	85,71	1	90,47	8
Fixed-point version	CMU	86,75	4	81,37	0	88,20	3
	CINEMA*	88,41	6	82,25	3	85,14	9
	WEB*	88,98	1	86,17	1	92,38	5
	ATT	99,25	0	97,50	0	96,50	0
	Total	90,71	15	86,85	4	90,95	17

*CINEMA and WEB are test sets of, respectively, 276 and 499 faces kindly provided by *GarciaandDelakis* [8].

We will illustrate our methodology on the CFF implementation, whose starting point was a floating point arithmetic version and required a memory allocation of 3.8 M-Bytes to process a QCIF format image (176 × 144 pixels). The reference complexity analysis of the floating-point version of the CFF shows that it requires 3 seconds to compute a single QCIF image on a 624 MHz Xscale processor. Hereafter, we present in detail each step of this methodology and the achieved performance results.

4. OPTIMIZING THE CFF ALGORITHM

4.1. Fractional transformation

CFF reference software was entirely written using floating-point arithmetic. Mobile-embedded target platforms lack floating-point hardware accelerator for power consumption reasons. Floating-point computations are usually implemented by software, but these are high CPU consuming functions. The first step towards embedding the algorithm is to transform the floating-point computations into fractional ones. Since one of our target platforms was the 16 bit DSP Starcore SC140, fractional Q15 arithmetic [23] was required (Q31 arithmetic may be used when more precision is needed).

The main advantage of the CFF algorithm is that the results of the subsampling layers S_1 and S_2 pass through hyperbolic tangent functions (which limits the data dynamics), thus reducing the risk for common issues of fixed-point computations such as arithmetic dynamic expansion and saturation. A simple methodology was used to normalize and transform each neural network coefficient in fixed-point arithmetic and compare the results with the floating-point version. Each coefficients kernel for each layer is first normalized to prevent accumulation overflow (sum of absolute values strictly lower than one). Each coefficient is then fitted to 16 bits fractional representation. Precision tests are carried out experimentally on standard face databases.

The main constraint of this transformation was to maintain the efficiency of the face detector. The benchmarking was done on different test sets of images, including the CMU test set (the most widely used data set in the literature). Table 1 gives the detection rates of the floating- and fixed-point versions on four test sets for different CFF configurations (varying output volume threshold and minimum allowed face size).

The comparison of the floating-point and fixed-point versions shows no significant loss in efficiency and detection rates are equivalent to the ones previously published in [8]. They are even better on some parts of the selected test sets. What is especially noticeable about the CFF efficiency is the very low level of false alarms, even after the fractional transformation.

4.2. Memory optimization

Due to the computational redundancy in the CFF algorithm, the reference software was processing layer by layer on the whole image (and scaled versions of the original image). This configuration is not suitable for an embedded platform since even for small QCIF images, 3.8 MBytes were allocated (e.g., the targeted SC140 DSP platform embeds only 512 kB of SRam).

In order to reduce this memory allocation without increasing the required amount of computations, a study was carried out on the data dependency in the algorithm. Figure 6(a) shows the amount of data needed in each layer in order to compute a single output of each neuron in layer N_1. This figure is similar to Figure 1 restricted to one feature map by layer.

Figure 6(b) illustrates the differential computation between two neighbouring outputs (south side) of neuron layer N_1. Slashed (resp., unslashed) grey parts are unused (resp., reused) previously computed data, whereas dark rectangles are newly computed data.

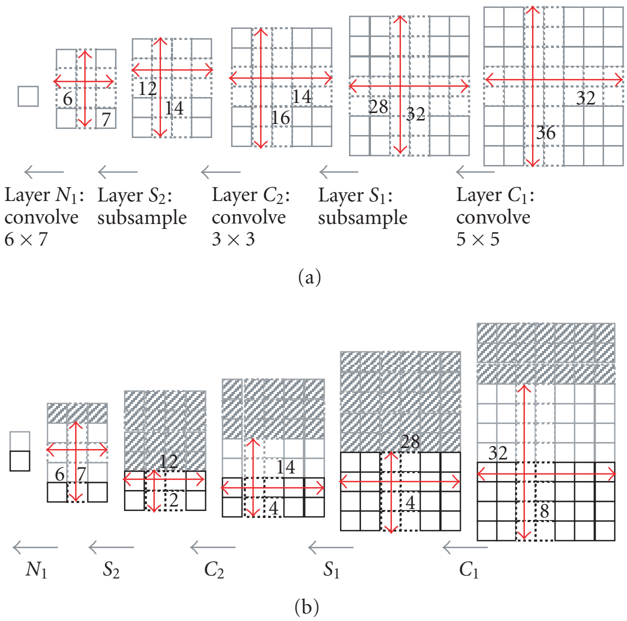

FIGURE 6: CFF data flow. (a) Amount of data needed in each layer, (b) differential computation between two neighbouring outputs (south side).

Since Figure 6(b) shows that intermediate computation from previous lines has to be kept as input of layers C_2 and N_1, the maximum gain in terms of memory footprint is achieved for a line-by-line processing of the output of layer N_1. Thus, in the final implementation, in order to compute one output line of layer N_1, we use 7 input lines of this layer. These input lines can be computed line by line in layer S_2 using two output lines of layer C_2. These two output lines require four input lines for layer C_2. Two of these four output lines are common with the previously computed lines, and the two others require four output lines of layer C_1. These four output lines of layer C_1 are computed using eight lines of the input image.

Table 2 represents the memory allocation analysis for the full image processing and the line-by-line processing. Each stage of output memory is parameterized by W and H, the width and height of the input image.

For a QCIF image, the gain in memory footprint is about 21. Other memory allocation optimizations (e.g., on-scaled images computation) have been made on the reference software leading to a memory footprint of 220 kB compared to the 3.8 Mbytes of the original version.

4.3. *Code optimization: parallelism exploitation*

One of our target-embedded platforms is a Starcore SC140 DSP which has 4 ALUs and multiplier capabilities. This processor is able to load eight 16 bits words and to compute 4 multiplication-accumulations (MACs) in one cycle. The main limitation to taking advantage of this parallelism is that the data's alignment constraints need to be satisfied: the Move.4F instruction [24] which loads four 16 bits-word data

TABLE 2: Memory allocation for full image and line-by-line processing.

Layer	Number of branches	Full image processing	Line-by-line processing
C_1 output	4	$(W-4)^*(H-4)$	$(W-4)^*4$
S_1 Output	4	$(W-4)/2^*(H-4)/2$	$(W-4)/2^*4$
C_2 Output	14	$(W-8)/2^*(H-8)/2$	$(W-8)/2^*2$
S_2 Output	14	$(W-8)/4^*(H-8)/4$	$(W-8)/4^*7$
N_1 Output	14	$(W-28)/4^*(H-32)/4$	$(W-28)/4^*1$
Total	—	$10.25^*W^*H+\cdots$	$66^*W+\cdots$

is only allowed for an eight-bytes aligned pointer and can be generated automatically by the compiler by appropriate C code rewriting and alignment directive use.

Let us analyze the first layer (C_1) which the profiling tool points out as being the most complex step of the CFF algorithm: each of the four feature maps of this layer consists of a convolution by a 5×5 kernel. Without any parallelization one convolution requires 25 data loads, 25 coefficient loads, 25 MACs instructions, and one store instruction. Since the Starcore is able to compute four MACs in one cycle, the theoretical minimum cycle count for processing 25 MACs (without load and store count) is $\lceil 25/4 \rceil = 7$ cycles. Without aligned load instructions, the Starcore is able to process two 16 bits load instructions by cycle (in parallel with the MACs instructions). Thus, due to the number of load and store instructions, one convolution would require at least $\lceil (25+25+1)/2 \rceil = 26$ cycles. The main goal in order to optimize such a function is to reduce the number of load and store instructions by using the Move.4F instruction.

Input data and coefficients are 16 bits words. Assuming that the first element is 8 bytes aligned, the Starcore should be able to load 4 data and/or coefficients in a single cycle. But, the 5×5 convolution processing is done on any image of the pyramid whose width is not necessarily multiple of 4. Thus if the first top-left pixel in the image is 8 bytes aligned, the first pixel on the second line will probably not be aligned preventing any use of multiple load instruction on these data. On the other hand, using aligned loads on coefficients would imply dividing the 5×5 kernel matrix into several matrices 4×4, 1×4, and 5×1, making the convolution processing more complex.

In order to reduce the number of load instructions per convolution, the proposed solution consists of factorizing the coefficients loads in order to process the 5×5 convolution several times (multisample processing).

Figure 7 presents the factorization process. Convolutions are done by 25 iterations on the whole block of pixels. At each iteration, groups of four multiplication-accumulations with a single coefficient are performed. This requires a temporal store and load of intermediate processing (e.g., $c[0,0] \cdot x[0,0], \ldots$), but, since this intermediate matrix can be 8 bytes aligned, four intermediate computations can be loaded or saved in a single instruction. Table 3 sums up the amount of load and store instructions needed for the

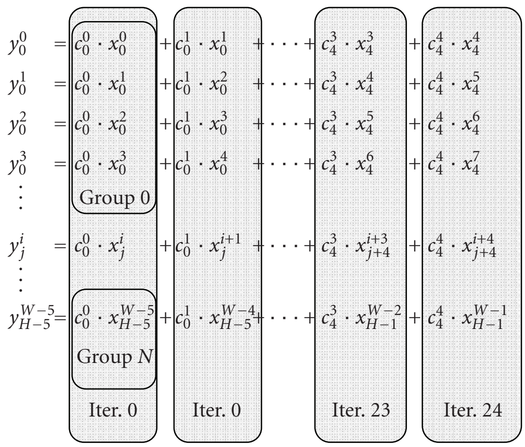

Figure 7: Parallelism exploitation: parallelization process.

Table 3: Load and store count for the 5 × 5 convolution of a block of size $S = (W-4)*(H-4)$.

	Initial version	Modified version
Nb load coef. instructions	$25*S$	25
Nb load data instructions	$25*S$	$25*S$
Nb load/store instructions for intermediate results	0	$25*S/4 + 25*S/4$
Final store instructions	S	0
Total	$51*S$	$25 + 37.5*S$

5 × 5 convolution of an image of size $W*H$ in function of $S = (W-4)*(H-4)$, the number of convolutions.

When processing four output lines of layer C_1 as depicted in the previous paragraph, the gain in terms of load/store instructions is for a QCIF image ($W = 176$, $H = 8$):

$$\frac{51*S - 25 - 37.5*S}{51*S} = \frac{27}{102} - \frac{25}{51*4*(W-4)} = 26,4\%. \tag{1}$$

We achieve the same gain in terms of the number of cycles by convolution with this factorized version (the inner loop takes 3 cycles to compute 4 MACs, compared to the 4 cycles required by the original version).

This optimization may also be applied on processors using SIMD instructions such as WMMX instructions on an Xscale-embedded processor. The efficiency of this optimization on these processors has not yet been evaluated.

4.4. Algorithm optimization

In this section, we present two examples of algorithmic optimization applied on the CFF which lead to a great increase in performance.

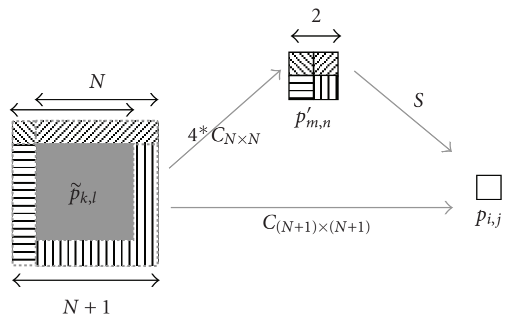

Figure 8: Convolution and subsampling fusion process.

Table 4: Instruction counts for sequential and fused versions.

	Number of Mac instructions
$C_{N\times N} + S$	$4*N2 + 4$
$C_{(N+1)\times(N+1)}$	$(N+1)^2$
Gain	$(3*N^2 - 2*N + 3)/(4*N^2 + 4)$
Gain ($N = 5$)	65%
Gain ($N = 3$)	60%

4.4.1. Convolution and subsampling fusion

When considering the data dependency (see Figure 6), we can see that, at each subsampling layer, there is no overlapping between input data to produce two neighbor subsampled elements.

The output element value $p_{i,j}$ (cf. Figure 8) of a C_i-S_i ($i = \{1,2\}$) couple can be expressed as follows:

$$\begin{aligned} p_{i,j} &= \alpha*\left(p'_{2i,2j} + p'_{2i,2j+1} + p'_{2i+1,2j} + p'_{2i+1,2j+1}\right), \\ p'_{m,n} &= \sum_{k=0}^{N}\sum_{l=0}^{N} c_{k,l}*\tilde{p}_{m+k,n+l}, \\ p_{i,j} &= \alpha*\left(\sum_{k=0}^{N}\sum_{l=0}^{N} c_{k,l}*\tilde{p}_{2i+k,2j+l} + \cdots \right. \\ &\qquad \left. + \sum_{k=0}^{N}\sum_{l=0}^{N} c_{k,l}*\tilde{p}_{2i+1+k,2j+1+l}\right) \\ &= \sum_{k=0}^{N+1}\sum_{l=0}^{N+1} \tilde{c}_{k,l}*\tilde{p}_{k,l}, \end{aligned} \tag{2}$$

where N is the convolution size, α is the subsampling coefficient, $c_{k,l}$ are the convolution coefficients, $p'_{i,j}$ are the inputs of the subsampling, $\tilde{p}_{k,l}$ are the inputs of the convolution, $\tilde{c}_{k,l}$ are the weighted sums of one-to-four $c_{k,l}$ coefficients.

So, we propose fusing each N by N convolution ($C_{N\times N}$) followed by subsampling (S) into a $(N+1)$ by $(N+1)$ convolution ($C_{(N+1)\times(N+1)}$) (see Figure 8).

Table 4 gives the computational and memory access complexities for each version. The gain achieved by this algorithmic optimization is huge in terms of the computational cost, 65% (resp., 60%) is obtained for the first layer C_1-S_1 (resp., second layer C_2-S_2) of the CFF.

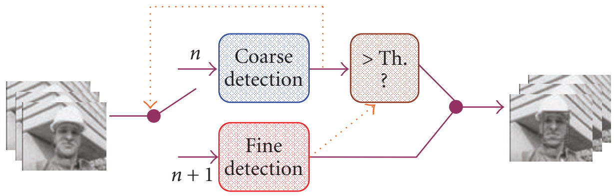

Figure 9: Functional diagram of the CFF video.

Furthermore, the merging of convolution and subsampling coefficients avoids multiple fractional arithmetic rounding, enabling slight improvements of benchmark results (e.g., on CMU test set +0, 2% on face detection rate and 2 false alarms versus 4).

4.4.2. Tracking adaptation

The CFF algorithm was first dedicated to still-image indexing, and thus, the process considers only an input image. One of our aims was to adapt this algorithm to video indexing.

Since the algorithm was organized in two stages; one coarse detection on the whole image and a second one in a finer pyramid centered at each candidate's face location, we have used this second stage for tracking the detected faces in successive frames.

The proposed video-based CFF algorithm can be seen as an intra and inter processing of images by analogy with image coding (H26x or Mpeg, [25]). One image among N is computed by the coarse detection (intra detection), and we do only apply the fine detection on the following images at each candidates' face location given by the previous image (inter detection). Fine detection using a local pyramid enables us to cope with the face size variation (zoom), and the window search area being 20 pixels around the previous face center enables us to handle most face motion issues.

In order to avoid the false detection alarms on "Intra" images, an adaptive volume threshold has been introduced downstream to the coarse detection. This threshold is adapted using an infinite impulse response filter whenever an Intra detection and its following Inter detection have contradictory answers. Figure 9 gives a functional description of the video adaptation.

Since profiling on coarse and fine detection was balanced (54% for coarse detection and 46% for fine detection under a Vtune profiling on Xscale PXA27x processor), we may at least foresee a speed-up factor of two. But, simulations and profiling on several platforms point out that this video-based CFF is about 3 times faster than the image-based one (for $N = 6$). This is mainly due to false detections of the coarse detection being removed by the first iteration of the fine detection, and so no longer tracked on the following images.

4.5. Performance results

Table 5 summarizes the speed-up factor obtained on a QCIF video test sequence (120 first frames of the Mpeg Foreman sequence) for each kind of optimizations done on the face detector.

Table 5: CFF and CFF video processing speed.

	Xscale PXA27x @ 624 MHz	Starcore SC140 @ 275 MHz	Pentium IV @ 3.2 GHz
Floating-point reference	0.3 fr/s	—	10 fr/s
Fixed-point version	4.5 fr/s	7 fr/s	32 fr/s
Code optimization	—	9 fr/s	—
Algorithm optimization (4.4.1)	6.5 fr/s	13 fr/s	58 fr/s
Tracking adaptation (4.4.2)	16.5 fr/s	35 fr/s	180 fr/s

A speed-up factor of 55 is obtained between the original floating-point version and the fixed-point face tracker on the Xscale platform enabling face-tracking-based services on mobile terminals. Without tracking adaptation, the improvement is still huge (22 times). Real-time face detection is also achieved on highly parallel DSP architecture. Table 5 also points out the strong impact of algorithm optimization on the application performance. Optimizations carried out for embedded platforms are also useful on a PC target able to process real time TV format video streams.

As a comparison with other embedded robust face detection implementations, we consider the works presented in [9, 11] that both propose AdaBoost-optimized solutions (based on the Viola and Jones approach [10]), respectively, on an ARM926 processor and a TI TMS320C6205. The Viola and Jones method is known to be less efficient than CFF [8], with a good detection rate of 76.1% with 10 false alarms for the CMU test set. The number of frames per second achieved by these implementations is, respectively, about 4 and 3 Hz for QVGA-like video format which is comparable to our frame-by-frame implementation of the CFF processing. However, the tracking adaptation enables us to 3 times outperform these frame rates.

Furthermore, as depicted before, the memory footprint has been reduced from 3.8 MBytes to 220 kBytes by the memory optimization step.

5. CONCLUSION AND PERSPECTIVES

In this paper, we have presented the implementation of a state-of-the-art face detector on two kinds of programmable embedded platforms. We have shown that both high detection rates and fast processing are achieved by applying our optimization flow methodology. Memory and code restructuring in conjunction with algorithm adaptation lead to significant improvement. This study proves that CNN algorithms are well suited for embedded implementation since they stand up to fractional transformations and they offer good opportunities for memory and algorithm optimizations. Indeed, we obtain a speed-up factor of 55 on an Xscale-PXA27x-based platform and real-time video processing (up to 35 QCIF fr/s) on a Starcore DSP. High efficiency is maintained, with a detection rate of 87% on the CMU test set and only 4 false alarms.

One of our final objectives is to provide an embedded face recognition system for biometrics applications. Usually, face-based identification systems require precise face and facial feature localization and also fine facial feature positioning. The first step depicted in this paper was the real-time implementation of this face detector by software optimizations. The second step is to precisely locate facial features, and we are now working on the implementation of a facial feature detector based on the same principles which is called C3F for convolutional face feature finder [26].

Furthermore, this study points out that the pipeline of convolutional and subsampling filters denotes high intrinsic and hidden parallelisms which will be exploited in future works with dedicated hardware implementation of CFF and C3F.

REFERENCES

[1] M.-H. Yang, D. J. Kriegman, and N. Ahuja, "Detecting faces in images: a survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 1, pp. 34–58, 2002.

[2] K. C. Yow and R. Cipolla, "Feature-based human face detection," *Image and Vision Computing*, vol. 15, no. 9, pp. 713–735, 1997.

[3] C.-C. Lin and W.-C. Lin, "Extracting facial features by an inhibitory mechanism based on gradient distributions," *Pattern Recognition*, vol. 29, no. 12, pp. 2079–2101, 1996.

[4] I. Craw, D. Tock, and A. Bennett, "Finding face features," in *Proceedings of the 2nd European Conference on Computer Vision (ECCV '92)*, pp. 92–96, Santa Margherita Ligure, Italy, May 1992.

[5] A. Lanitis, C. J. Taylor, and T. F. Cootes, "Automatic face identification system using flexible appearance models," *Image and Vision Computing*, vol. 13, no. 5, pp. 393–401, 1995.

[6] B. Moghaddam and A. Pentland, "Probabilistic visual learning for object representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 7, pp. 696–710, 1997.

[7] K.-K. Sung and T. Poggio, "Example-based learning for view-based human face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 39–51, 1998.

[8] C. Garcia and M. Delakis, "Convolutional face finder: a neural architecture for fast and robust face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 11, pp. 1408–1423, 2004.

[9] X. Tang, Z. Ou, T. Su, and P. Zhao, "Cascade AdaBoost classifiers with stage features optimization for cellular phone embedded face detection system," in *Proceedings of the 1st International Conference on Natural Computation (ICNC '05)*, pp. 688–697, Changsha, China, August 2005.

[10] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, pp. 511–518, Kauai, Hawaii, USA, December 2001.

[11] J.-B. Kim, Y. H. Sung, and S.-C. Kee, "A fast and robust face detection based on module switching network," in *Proceedings of the 6th IEEE International Conference on Automatic Face and Gesture Recognition (FGR '04)*, pp. 409–414, Seoul, Korea, May 2004.

[12] T. Theocharides, G. Link, N. Vijaykrishnan, M. J. Irwin, and W. Wolf, "Embedded hardware face detection," in *Proceedings of the 17th IEEE International Conference on VLSI Design*, pp. 133–138, Mumbai, India, January 2004.

[13] H. A. Rowley, S. Baluja, and T. Kanade, "Neural network-based face detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 1, pp. 23–38, 1998.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[15] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: an overview," in *Proceedings of IEEE Symposium on Low Power Electronics*, pp. 38–39, San Diego, Calif, USA, October 1994.

[16] S. Muchnick, *Advanced Compiler Design & Implementation*, Morgan Kaufmann, San Francisco, Calif, USA, 1997.

[17] J. C. Bauer, E. Closse, E. Flamand, M. Poize, J. Pulou, and P. Penier, "SAXO: a retargetable optimized compiler for DSPs," in *Proceedings of the 8th International Conference on Signal Processing Applications & Technology (ICSPAT '97)*, San Diego, Calif, USA, September 1997.

[18] V. Palanciuc, D. Badea, C. Ilas, and E. Flamand, "A spill code reduction technique for EPIC architectures," in *Proceedings of the 1st Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-1 '01)*, Austin, Tex, USA, September 2001.

[19] INTEL PCA Optimization guide, http://www.appzone.intel.com/pcadn/.

[20] T. Van Achteren, G. Deconinck, F. Catthoor, and R. Lauwereins, "Data reuse exploration techniques for loop-dominated applications," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '02)*, pp. 428–435, Paris, France, March 2002.

[21] F. Catthoor, K. Danckaert, C. Kulkarni, et al., *Data Access and Storage Management for Embedded Programmable Processors*, Kluwer Academic, Boston, Mass, USA, 2002.

[22] T. Simunic, L. Benini, G. De Micheli, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in *Proceedings of the 13th International Symposium on System Synthesis (ISSS '00)*, pp. 193–198, Madrid, Spain, September 2000.

[23] A. Bateman and I. Paterson-Stephens, *The DSP Handbook, Algorithms, Applications and Design Techniques*, Prentice-Hall, Upper Saddle River, NJ, USA, 2002.

[24] "SC140 DSP Core Reference Manual Second Revision," Motorola Corporation, 2001.

[25] MPEG-4 visual version 1, "Coding of audio-visual objects—Part 2: visual," ISO/IEC JTC1 14 496-2, 1999.

[26] S. Duffner and C. Garcia, "A connexionist approach for robust and precise facial feature detection in complex scenes," in *Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis (ISPA '05)*, pp. 316–321, Zagreb, Croatia, September 2005.