EMBEDDED CONVOLUTIONAL FACE FINDER

Sébastien Roux, Franck Mamalet and Christophe Garcia

France Telecom division R&D, 28 Chemin du Vieux Chêne, 38243 Meylan, France
e-mail: <first name>.<last name>@francetelecom.com

ABSTRACT

In this paper, a high-level optimization methodology is applied for the implementation of the well-known Convolutional Face Finder (CFF) algorithm for real-time applications on cellular phone, such as teleconferencing, advanced user interfaces, pictures indexing and security access control. This face detector is based on a feature extraction and classification technique which consists in a pipeline of convolutions and subsampling operations.

Design of embedded systems must find a good trade off between performance and code size due to the limited amount of resource available. We propose a methodology to cope with the main drawbacks of the CFF original implementation like floating-point computation and memory allocation, to allow parallelism exploitation and perform algorithm optimizations. Results show that our embedded face detection system can accurately locate faces with less computational load and memory cost. It runs on a 275MHz Starcore DSP at 9 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 strongest constraints is the limited computational resources. Low memory capacities, low CPU frequency and lack of specialized hardware like floating point unit are some of the major differences between a PC and an embedded platform. Unfortunately, advanced algorithms are usually developed on PC without any implementation restriction in mind. Thus, porting application on power-constrained embedded 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. A survey was published by Yang et al. [1] in 2002. In this paper, face detection techniques are classified in three main categories:

- feature invariant approaches [2],
- template matching methods [3],
- appearance-based methods [4].

A recent technique that belongs to the third category, called Convolutional Face Finder (CFF) has been introduced by Garcia and Delakis [5] which leads to the best performance on standard face databases. The CFF is an image-based neural network approach that allows robust detection, in real world images, of multiple semi-frontal faces of variable size and appearance, rotated up to +/- 20 degrees in image plane and turned up to +/- 60 degrees.

Addressing both face detection performance and implementation on embedded system has been considered in recent years by Tang et al. [6] for cascade adaboost classifiers [7] on ARM based mobile phones. The Adaboost technique was also used in [8] for implementing a hybrid face detector on a TI DSP. Another way to achieve resource constrained implementation is to design dedicated hardware for face detection. In [9], the authors proposed an ASIC implementation of the face detector introduced by Rowley et al [10].

However, in real time embedded implementations one often has to trade off among 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 will show that we have achieved both efficiency and speed objectives (10 images/s) with our CFF implementation.

The remainder of the 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 (Starcore SC140) and RISC (XScale) based platforms are provided in section 4. Finally, conclusions and perspectives are drawn in section 5.

2. CFF ALGORITHM OVERVIEW

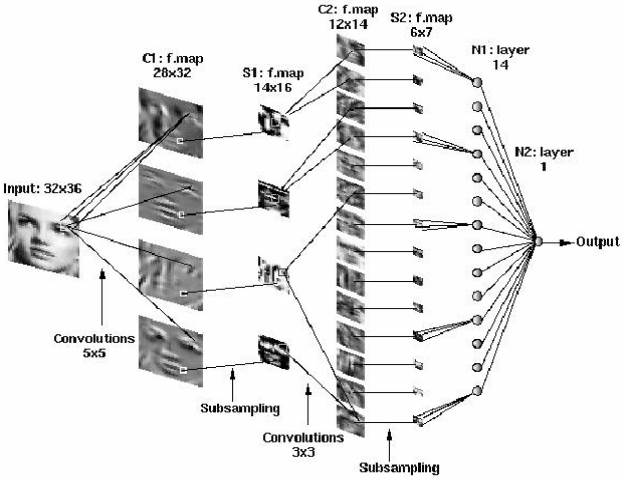

Fig.1. Convolutional Face Finder pipeline.

The Convolutional Face Finder was presented in [5] and relies on Convolutional Neural Networks introduced and successfully used by LeCun and al. [11]. It consists in a pipeline of convolutions and subsampling operations (Fig. 1). This pipeline performs automatic

feature extraction in image areas of size 32x36, and *classification* of the extracted features, in a single integrated scheme. In [5], the authors present both the training methodology to learn the coefficients using back propagation, and the face localization process when training has been completed. In this paper we will only consider the face localization process. Fig. 2 presents in detail the steps of this face localization process:

- A coarse detection is first performed as follows. The CFF is applied on a pyramid of scaled versions of the original image (Fig. 2-1) in order to handle faces of different sizes: each scale produces a map of faces candidates (Fig. 2-2) which is fused back to the input image resolution and produce clusters of positives answers (Fig. 2-3). For each cluster a representative face is computed as the centroid of its candidate face centers and sizes weighted by their individual network responses (Fig. 2-4).
- Then a fine detection takes those candidates as input and applies locally the CFF on a small pyramid around the face candidate center position. The volume of positive answers is considered to take the classification decision of face or non-face (Fig. 2-5). Finally, overlapping candidates are fused to remove multi detections of the same face.

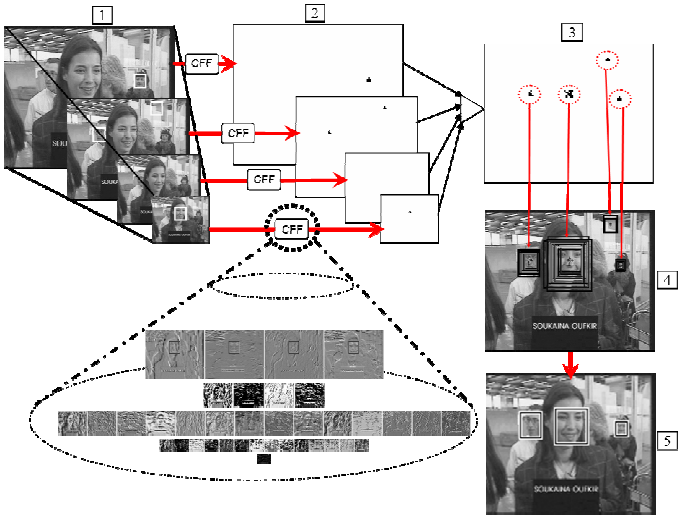

Fig. 2. The different steps of the process of face localization.

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

In order to implement complex algorithm on embedded target processor, compilers are the tools to optimize the instructions flow. In the last decade, many research activities have been carried out in instruction-flow optimizations [12] and optimizing compilers [13], and some have led to industrial products such as the Metrowerks compiler for SC140 [14]. However, compilers can only cope with the instructions flow optimization and parallelization.

Even if these compilers avoid mostly any human assembly programming, many optimizations have to be done by manual high-level code-rewriting.

Our approach is based on iterations of high-level code optimizations and profiling to focus first on the most CPU resource consuming functions. When dealing with an algorithm such as CFF, the first step towards an embedded implementation is to avoid floating point calculation. This step is done thanks to a fractional transformation in accordance with data dynamics and processor data paths. 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 (fig. 3).

Profiling tools depend on the target platform: for instance, we use the VTune software on 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.

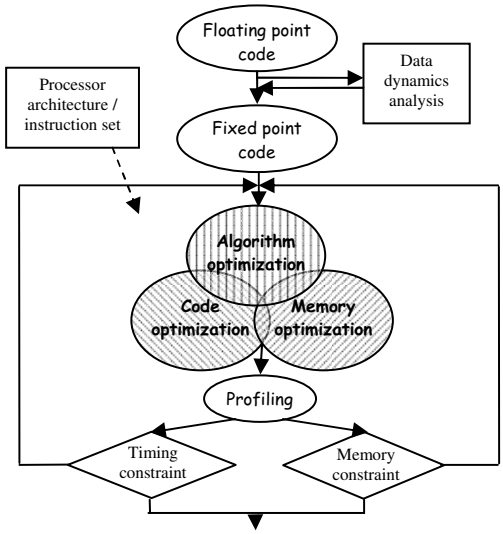

Fig. 3. Diagram of followed methodology.

We will illustrate our methodology on the CFF implementation, which starting point was a floating point arithmetic version and required a memory allocation of 3.8 MBytes to process a QCIF format image (176x144 pixels). The reference complexity analysis of the floating point version of the CFF shows that it requires 3s to compute a single QCIF image on a 624MHz 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

The reference software of the CFF 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 bits DSP Starcore SC140, fractional Q15 arithmetic [15] 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 S1 and S2 pass through a hyperbolic tangent function, 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 stage coefficient in fixed-point arithmetic and compare the results with the floating point version.

The main constraint of this transformation was to keep 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 for different configurations of the CFF (varying output threshold and minimum faces detection size).

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
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

* CINEMA and WEB are test sets of respectively 276 and 499 faces kindly provided by C.Garcia [5]

The comparison of the floating point and fixed point versions shows up no significant loss in efficiency, and detection rates are equivalent to the previously published ones in [5]. They are even slightly better on part of the selected test sets. What is especially noticeable about CFF efficiency is the low level of false alarms and 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 (or 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 (for instance, the targeted SC140 DSP platform embeds only 512kB of SRam).

In order to reduce this memory allocation without increasing the required amount of computations, a study was made on the data dependency in the algorithm. Fig. 4a shows the amount of data needed in each layer in order to compute a single output of each neuron layer N1. This figure is similar to Fig.1 restricted to one feature map by layer.

Fig. 4b illustrates the differential computation between two neighbouring outputs (south side) of neuron layer N1. Slashed (resp. unslashed) grey parts are unused (resp re-used) previously computed data, whereas dark rectangle are newly computed data.

Since Fig. 4b shows that intermediate computation from previous line has to be kept as input of the layer C2 and N1, the maximum gain in terms of memory footprint is achieved for a line by line processing of the output of N1 layer. Thus, in the final implementation, in order to compute one output line of the layer N1, we use 7 input lines of this layer. These input lines can be computed line by line in layer S2 using two output lines of layer C2. These two output lines require four input lines for the layer C2. Two of these four output lines are common with the previously computed lines, and the two others require four output lines of the layer C1. To end with, these four output lines are computed using eight lines of the input image.

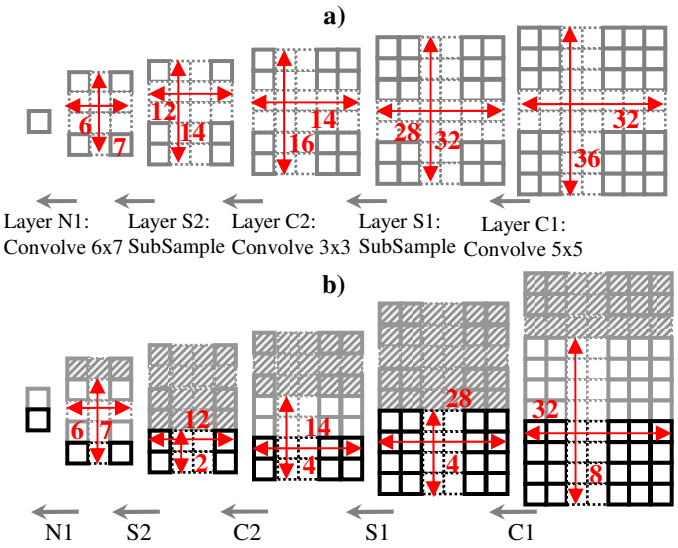

Fig. 4. CFF data flow. a) amount of data needed in each layer, b) differential computation between two neighbouring outputs (south side).

CFF algorithm analysis for the full image processing (resp. the line by line processing) shows that memory allocation is about 10.25*W*H+… (resp. 66*W+…), W and H being 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 image computation) have been made on the reference software leading to a memory footprint of 220kB 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 take advantage of this parallelism is that one needs to satisfy data's alignment constraints: the Move.4F instruction which loads four 16bits-word data is only allowed for an eight bytes aligned pointer and can be generated automatically by the compiler by appropriate C code re-writing and alignment directive use.

Let us analyse the first layer (C1) which is pointed out by the profiling tool as the most complex step of the CFF algorithm: each of the four feature maps of this layer consists in a convolution by a 5x5 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 [25/4] = 7 cycles. The Starcore is able to process two (single or multiple) load instructions by cycle (in parallel with the MACs instructions). Thus, without aligned loads instructions, one convolution would require at least [(25+25+1)/2] = 26 cycles. So, the main goal to optimize such a function is to reduce the number of load and store instructions by using the Move.4F instruction.

However, the 5x5 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 from any use of multiple load instruction on these data.

The proposed solution in order to reduce the number of load instructions per convolution consists in factorizing the coefficients loads for several processing of the 5x5 convolution (multi-sample processing).

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 done. This requires a temporal store and load of intermediate processing, but, since this intermediate matrix can be 8 bytes aligned, four intermediate computations can be loaded or saved in a single instruction. As a result, the amount of load and store instructions needed for this modified version is 25+37.5*S compared to the 51*S instructions required for the initial version (where S=(W-4)*(H-4)).

So, when processing four output line of the layer C1 as depicted in the previous paragraph, the gain in terms of load/store instructions is 26.4 % for a QCIF image (W = 176, H = 8).

Since the SC140 compiler achieves the best instruction flow parallelization, we get the same gain in term of number of cycles by convolution with this factorized version.

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

4.4. Performance results

Table 2 summarizes the speed up factor obtained on a QCIF video test sequence (120 first frames of the Mpeg Foreman sequence) after several others iterations of the optimization methodology.

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

TABLE 2: CFF processing speed

	Xscale PXA27x @ 624MHz	Starcore SC140 @ 275MHz	Pentium IV @ 3.2GHz
Floating point reference	0.3 fr/s	-	10 fr/s
Fixed point and optimized version	4.5 fr/s	9 fr/s	32 fr/s

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. Indeed, we obtain a speed-up factor of 14 on an Xscale PXA27x based platform, and video processing at 9 QCIF fr/s on a Starcore DSP. Efficiency is maintained high, with 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 need precise face detection but also fine facial feature localization. The first step depicted in this paper was the real time implementation of this face detector by software optimizations. The second step is to detect facial features, and we are now working on the implementation of a facial feature position extractor based on the same principles which is called C3F for Convolutional Face Feature Finder [16].

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.

6. REFERENCES

[1] M. Yang, D. Kriegman, and N. Ahuja, “Detecting Faces in Images: A Survey,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 24, no. 1, pp. 34-58, Jan. 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] I. Craw, D. Tock, and A. Bennett, “Finding Face Features,” Proc. Second European Conf. Computer Vision, pp. 92-96, 1992.

[4] K.-K. Sung and T. Poggio, “Example-Based Learning for View-Based Human Face Detection,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 20, no. 1, pp. 39-51, Jan. 1998.

[5] C. Garcia and M. Delakis, “Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 26, no. 11, pp. 1408-1423, Nov. 2004.

[6] X. Tang, Z. Ou, T. Su and P. Zhao, “Cascade AdaBoost Classifiers with Stage Features Optimization for Cellular Phone Embedded Face Detection System,”. ICNC (3) 2005, pp. 688-697.

[7] P. Viola and M. Jones, “Rapid Object Detection Using a Boosted Cascade of Simple Features,” Proc. Int’l Conf. Computer Vision and Pattern Recognition, vol. 1, pp. 511-518, 2001.

[8] J.-B. Kim, Y. H. Sung, S.-C. Kee, “A Fast and Robust Face Detection based on Module Switching Network, ” Int. Conf. on Automatic Face and Gesture Recognition, pp 409-414, May 2004.

[9] T. Theocharides, et al, “Embedded Hardware Face Detection”, Int. Conf. on VLSI Design, pp. 133-137, Jan. 2004.

[10] H. Rowley, S. Baluja, and T. Kanade, “Neural Network-Based Face Detection,” IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 20, no. 1, pp. 23-38, Jan. 1998.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-Based Learning Applied to Document Recognition,” Proc. IEEE, vol. 86, no. 11, pp. 2278-2324, 1998.

[12] V. Tiwari, S. Malik, and A.Wolfe, “Compilation techniques for low energy: An overview,” Proc. of the Symposium on Low Power Electronics, 1994.

[13] S. Muchnick, “Advanced Compiler Design & Implementation.,” Morgan Kauufmann Publishers, 1997.

[14] V. Palanciuc, D. Bade, E. Flamand, C. Ilasa, “Spill Code Reduction Technique for EPIC architectures application in the Metrowerks StarCore C compiler ,” Workshop on EPIC, 2001.

[15] A. Bateman and I. Paterson-Stephens, The DSP Handbook, Algorithms, Applications and Design Techniques, Prentice Hall, pp. 128-130, 2002.

[16] S. Duffner, C. Garcia, “A Connexionist Approach for Robust and Precise Facial Feature Detection in. Complex Scenes,” Int. Symp. on Image and Signal Processing and Analysis (ISPA), 2005.