Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 938256, 9 pages
doi:10.1155/2008/938256

Research Article

Design of a Real-Time Face Detection Parallel Architecture Using High-Level Synthesis

Nicolas Farrugia,[1] Franck Mamalet,[1] Sébastien Roux,[1] Fan Yang,[2] and Michel Paindavoine[2]

[1] *MAchine to machine technologies Tangible Interactions expertiSe on devices Laboratory (MATIS), Orange Labs, 28 Chemin du Vieux Chène, 38243 Meylan, France*
[2] *Laboratory of Electronics Informatics Image (LE2i), Health-STIC Federative Research Institute (IFR100), Burgundy University-Engineer Science Center, 21078 Dijon, France*

Correspondence should be addressed to Franck Mamalet, franck.mamalet@antispamorange-ftgroup.com

Received 10 March 2008; Revised 20 June 2008; Accepted 12 November 2008

Recommended by Dragomir Milojevic

We describe a High-Level Synthesis implementation of a parallel architecture for face detection. The chosen face detection method is the well-known Convolutional Face Finder (CFF) algorithm, which consists of a pipeline of convolution operations. We rely on dataflow modelling of the algorithm and we use a high-level synthesis tool in order to specify the local dataflows of our Processing Element (PE), by describing in C language inter-PE communication, fine scheduling of the successive convolutions, and memory distribution and bandwidth. Using this approach, we explore several implementation alternatives in order to find a compromise between processing speed and area of the PE. We then build a parallel architecture composed of a PE ring and a FIFO memory, which constitutes a generic architecture capable of processing images of different sizes. A ring of 25 PEs running at 80 MHz is able to process 127 QVGA images per second or 35 VGA images per second.

1. INTRODUCTION

Face detection and analysis in images defining a parallel and video streams is an important research field and has many applications in security access control, image indexing, and person identification. New applications on power-constrained devices are foreseen, like video coding in mobile videoconference and intelligent user interfaces.

Many algorithms for face detection have been proposed in the past twenty years [1]. The chosen face detection method is the Convolutional Face Finder (CFF), introduced by Garcia and Delakis in [2]. It 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 semifrontal faces of variable size and appearance, rotated up to ±20 degrees in image plane and turned up to ±60 degrees. In [3], the authors have shown that the CFF algorithm can be implemented efficiently on embedded software platforms, while keeping a good detection rate and a low false alarm rate. Many optimisations were done, leading to significant gains in terms of processing speed and memory requirements, thus enabling face detection on a mobile phone with five QCIF (176×144) frames per second and only 220 KBytes of memory [3]. However, face detection is most often the first step of a face analysis process and will require a faster system.

There have been a few attempts at hardware systems for face detection [4, 5] but the authors report lower detection rates and higher false alarm rates than with the CFF [2], and frame rates up to 50 QVGA (320×240) frames per second. So far, no hardware implementation of the CFF algorithm has been reported.

We, therefore, aim to design the first fast and robust face detection optimised hardware by defining a parallel architecture for the CFF algorithm. In [6], we have described an optimised algorithm architecture matching methodology, consisting of dataflow modelling of the algorithm, parallelism extraction, and complexity analysis. Using this information, we have performed a coarse-grain design space exploration, which enabled us to specify an efficient parallel architecture, consisting of a ring of PE where each PE processes the whole face detection algorithm on a small block of data and communicates a small amount of data to one of its neighbours. In this paper, we present the implementation of such a PE capable of handling the successive convolutions of the CFF algorithm.

In order to ease and accelerate the implementation of complex algorithms, new methodologies have been studied and developed in the past twenty years. High-Level Synthesis (HLS) methods produce designs by specifying them using a high-level language like C. Many approaches and tools for HLS are now available [7–11] and are starting to be mature enough to be used for complex designs. In this paper, we propose a guided HLS approach using the UGH tool [11]. Using this approach, we are able to explore and implement several PE alternatives.

The remainder of the paper is organised as follows. In Section 2, we present the CFF algorithm and previous works which have introduced a dataflow model of the algorithm and a theoretical architecture of PEs efficiently implementing this algorithm. In Section 3, we present our high-level synthesis based exploration which enables us to evaluate several implementation alternatives. We give processing times and synthesis results for four PE alternatives and select one optimal PE. In Section 4, we present a parallel architecture composed of the previous PE, which enables us to process images of different sizes. We, finally, give the performances of this architecture, and we compare our face detection systems with those in the literature. We will then conclude this paper and give our perspectives in Section 5.

2. THE CFF ALGORITHM-OVERVIEW AND PREVIOUS WORKS

2.1. Overview of the CFF algorithm

The Convolutional Face Finder was presented in [2] and relies on convolutional neural networks introduced by LeCun and Bengio [12].

In this paper, we will only consider the core of the face localization process as depicted in Figure 1. The convolutional neural network used to implement the face detector has been previously optimised in [3], and consists of a set of two different kinds of layers.

(i) CSi layers are called convolutional layers and contain a certain number of planes. Each element 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, which corresponds to a pure convolution with a mask applied over the planes in the previous layer. A 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 CSi layer is typically done with horizontal and vertical steps of two pixels which correspond to perform local averaging and subsampling operations. Then, the results are passed through a hyperbolic tangent function, used as an activation function. As a result, the CSi layer also performs a reduction by two of the dimensionality of the input.

(ii) Ni layers are called classification layers and are applied after feature extraction and input dimensionality reduction of CSi layers. These layers correspond to a multilayer perceptron.

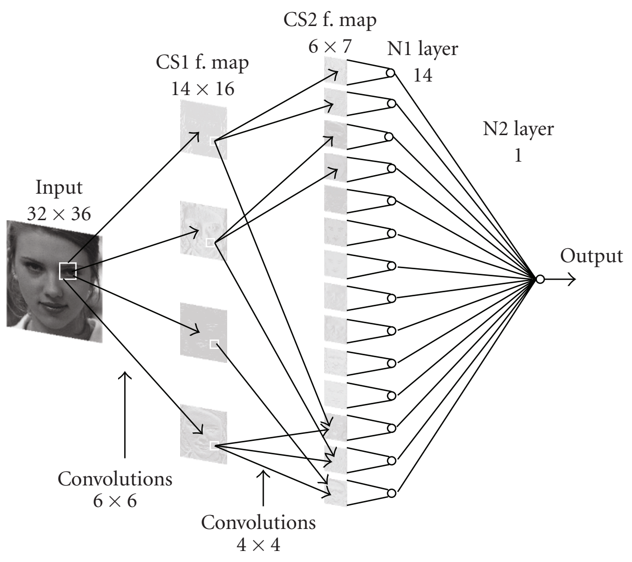


Figure 1: Convolutional Face Finder pipeline.

All parameters (convolution kernels, biases, neuron weights) have been learnt automatically using a modified version of the backpropagation algorithm with momentum [2]. In the remainder of this paper, we will only consider the face localization process when training has been completed.

The detail of CFF layers is given in Figure 1.

(i) CS1 layer performs convolutions with masks of dimension 6×6. It contains four feature maps and therefore performs four convolutions on the input image.

(ii) CS2 layer performs 4×4 convolutions and has fourteen feature maps. Each of the four-subsampled feature maps of CS1 is convolved by two different 4×4 masks, providing the first eight feature maps of CS2. The other six feature maps of CS2 are obtained by fusing the results of two 4×4 convolutions on each possible pair of CS1 feature maps.

(iii) N1 layer contains 14 sigmoid neurons. Each neuron is fully connected to exactly one feature map of the CS2 layer whose size is 6×7.

(iv) N2 layer consists of a unique neuron fully connected to all the neurons of the N1 layer. The output of this neuron is used to classify the input image as face (positive answer) or non-face (negative answer).

2.2. Previous works—dataflow model description

In [6], we have established a design space exploration methodology based on dataflow modelling and parallelism

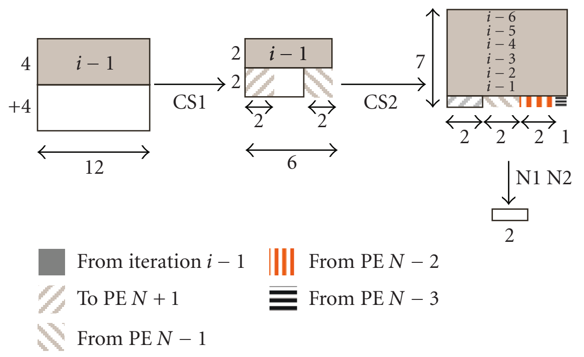


FIGURE 2: Dataflow model for one PE of the CFF algorithm for producing two data of N1 layer.

extraction of the CFF algorithm. This exploration has shown that the maximum parallelisation efficiency is obtained by massive data parallelism exploitation with a ring of PE.

In Figure 2, we detail the corresponding dataflow model for a given path in the CFF algorithm, which involves four successive PEs (N, $N-1$, $N-2$, and $N-3$) and seven successive temporal iterations for the calculation of two N1 layer output data.

Each PE processes the whole algorithm on a block of 8 lines of 12 pixels and transfers a small amount of data (overlapping parts) to one of its neighbours. The several slashed parts in Figure 2 represent the data to be transferred between successive PEs.

To start with, CS1 6×6 convolutions are applied on a 12×8 block of data with horizontal and vertical steps of two pixels, thus producing 2×4 output data. Then, PE N has to send a 2×2 output data to PE $N+1$ and receives a 2×2 block from PE $N-1$. Using data from previous iteration, CS2 is applied on a 4×6 data block which produces two data to be used as input of the N1 layer. These data are sent to the nearest neighbour and each PE gathers five data from three other PEs. With the six previous iterations, two N1 output data and two face detections can be computed.

3. IMPLEMENTATION OF A PROCESSING ELEMENT

In this section, we detail a complete implementation of a PE able to process the whole CFF algorithm. The design of this PE is guided by the results of the DSE/AAA methodology presented in [6]. The coarse-grain PE model previously used in this exploration does not detail neither the internal dataflow needed to process the convolutions nor the way the coefficients for the convolutions are handled.

In this paragraph, we will describe the local dataflow necessary to implement each CFF layer on the same PE. We will see that the requirements of the CFF algorithm in terms of dataflow management make it difficult to do an optimised design in manual VHDL coding.

We will then present a high-level synthesis based implementation which enables us to quickly evaluate several implementations of this complex dataflow. We then give simulation and synthesis results of this implementation.

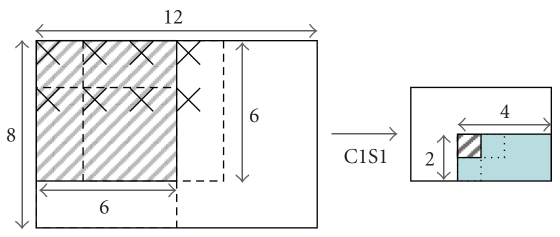


FIGURE 3: Local dataflow of CS1 and kernel coefficient reusing.

3.1. Convolutions and local dataflow

In the design of a PE, we need to consider the local dataflow of the computation in terms of input data and also in terms of convolution kernel coefficients. Each PE processes CS1, CS2, N1, and N2 layers on an input block of 12×8 pixels, as described in Figure 2. To illustrate the complexity of the local dataflow, we will focus on the CS1 layer whose details are given in Figure 3.

The slashed part on the left side (resp., on the right side) represents the first 6×6 input window (resp., the result of CS1 on this input window). Dotted borders represent the next horizontal and vertical input windows (resp., the next results) considering horizontal and vertical steps of two input pixels. There are eight input windows to compute, depicted in Figure 3 by crosses on their top-left corner. These windows comprise many overlapping data: 17% of the input image block is involved in the computation of six different windows.

Thus, the first coefficient of each kernel is applied to these data, and each coefficient is used eight times. In addition, each window is processed with four different CS1 kernels. This complex local dataflow can also be represented sequentially as a 5-deep "for" loop nest written in C language, as depicted in Figure 4. The inner operation in the loop nest is a Multiplication-Accumulation (MAC) between a kernel coefficient, an input data, and a previous accumulation. Furthermore, as presented in Section 2.1, the CS2 and N1 layers also have very complex local dataflow.

Our main goal is to design a PE which processes the whole algorithm on a window of size 12×8 and communicates with its neighbours as described in the dataflow definition in the Section 2.1. Concerning the local dataflow of the algorithm, there are three main issues to cope with.

(1) PE memory organisation and bandwidth: each feature map has to be stored and read back in a parallel way to enable local task parallelism (e.g., two CS2 feature maps computed in parallel).

(2) Fine scheduling and address generation: a choice of scheduling has to be made to efficiently exploit data locality and convolution regularity, and address sequences have to be computed for each memory.

(3) Inter-PE communication and temporal iterations: overlapping parts between PEs have to be transferred and data between successive iterations have to be stored and read back.

Each issue can be addressed with several implementation solutions. For example, issues one and three involve a choice of the type of memory (FIFO or shared RAM, dual port RAM, register banks, etc.) and its size, as well as the definition of the links between PEs. In addition, there are many fine scheduling choices to exploit different kinds of parallelism, and for each scheduling, complex address generation has to be computed for each memory.

The complexity and interdependency of these issues make it hard to do a manual RTL description for each possible implementation solution. Even if theoretical study using polyhedral models and loop transformation techniques [13] could be done in order to define an optimised fine scheduling of one loop nest, such a study on the whole algorithm is beyond the scope of this paper.

In the next paragraph, we present a High-Level Synthesis (HLS) flow which enables us to explore several fine schedulings, memory organisations and bandwidths, and inter-PE communications, in high-level C language, and to automatically generate a synthesizable PE including datapath, control and address generation. Our objective is to quickly obtain an efficient and functional PE prototype which will be used to implement the parallel dataflow described in Section 2.2.

```
for (i4 = 0; i4<4; i4 ++)                    //filter number
  for (i0 = 0; i0<2; i0 ++)                  // output window row number
    for (i1 = 0; i1<4; i1 ++)                // output window column number
      for (i2 = 0; i2<6; i2 ++)              // coefficient row number
        for (i3 = 0; i3<6; i3 ++)            // coefficient column number

{result[i1][i4][i0]+ = data[2 × i0 + i2][2 × i1 + i3]× coeff[i4][i2][i3] ;}
```

Figure 4: Sequential description of CS1: Loop nest.

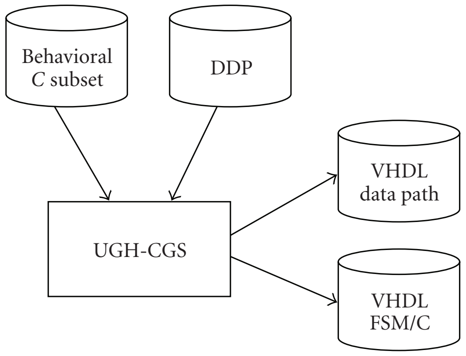


Figure 5: UGH coarse-grain scheduling design flow.

3.2. *PE design using a high-level synthesis tool*

(1) *High-level synthesis*

We have chosen to use an HLS approach in order to accelerate exploration, implementation, and validation of the PE. Many approaches for HLS have been proposed. There are commercial tools [7–9] which provide complete frameworks for hardware synthesis. Such tools are designed to accelerate and ease the development in a hardware project. The formalisms, formats, or platforms used are thus proprietary and often very specific and locked. Therefore, these tools may not be well suited for research purposes. On the contrary, some academic tools are open-source [10, 11], and provide the user with more flexibility and visibility on the tool behaviour. In Section 3.3, we present our implementation using UGH (User-Guided HLS) tool, an open-source HLS tool integrated in a larger framework for SoC design called dysident, and was developed by LIP6 laboratory. This tool is adapted to our study for two main reasons. First, it allows an architecture to be targeted by specifying the available resources before synthesis. This allows us to rely on the previous results of our manual PE implementation [6]. Second, UGH allows us to describe directly in C language all data input/output transfers, address calculation and dataflow management (e.g., ping pong strategies, modulo addressing, multiple data sources, etc.), which are required for our application [11].

(2) *UGH design flow*

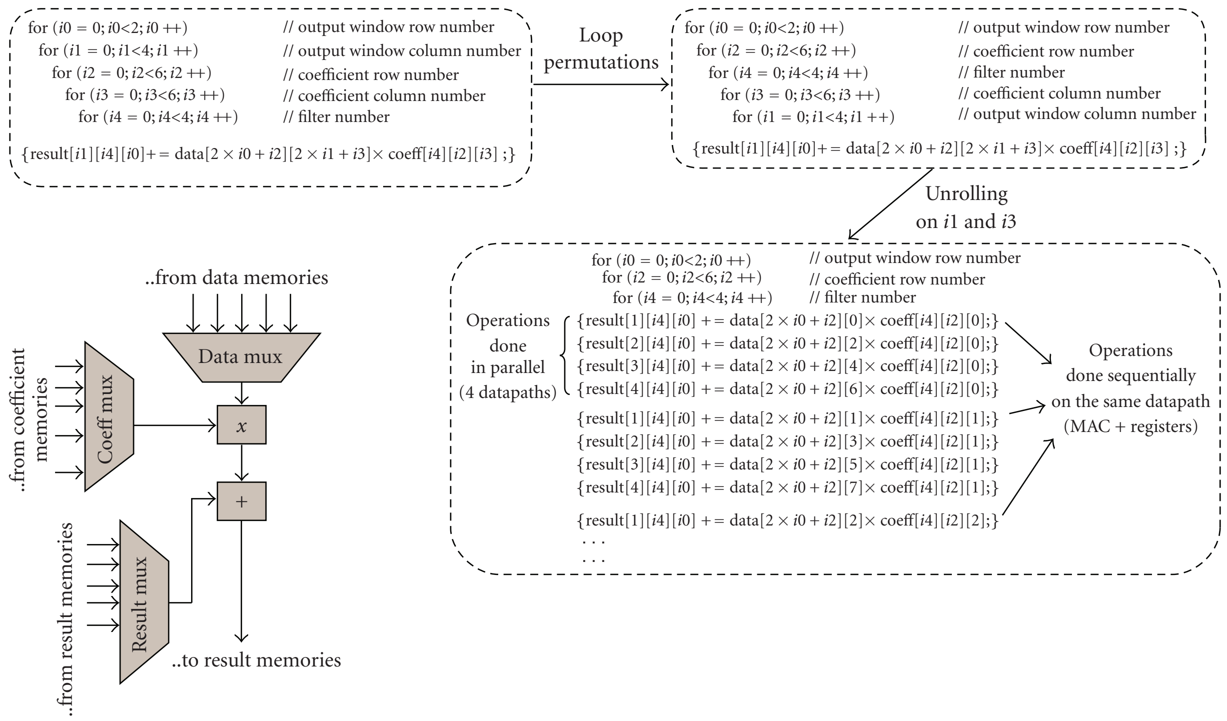
A complete description of the UGH design flow can be found in [11]. In this paper, we will focus on the first part of UGH design flow (Figure 5) which performs a Coarse-Grain Scheduling (CGS) with a behavioural C description of the algorithm and a Draft Data Path (DDP). The C description uses a subset of the C language whose main limitation is the forbidden use of pointers. Special data types are introduced to handle variable bit sizes (e.g., unsigned integer coded on 4 bits). The DDP describes the available resources for the target design (e.g., ALUs, RAMs, multipliers, bitwise logic, etc.) and UGH-CGS binds and schedules the algorithm on the available resources, and generates a VHDL Finite State Machine (FSM) and a VHDL structural datapath which are both synthesisable.

Two particular features of UGH have been much used in our design.

(i) *Automatic parallelisation, datapath, and address generation*: UGH-CGS is able to detect parallelism at the instruction level and schedules it on the available resources which are specified in the DDP file. UGH-CGS is then able to automatically generate the corresponding datapaths and all necessary links (multiplexers and corresponding connections) between the resources, and also all control signals such as memory addresses

(ii) *Flexible communication model*: inputs and outputs can be specified in a direct and flexible way, by simply writing specific "ugh_read" and "ugh_write" operations at any time of the source C code, and by defining the corresponding input or output port in the DDP.

Given that the C description exhibits some parallelism at the instruction level, the first feature makes it possible to quickly test and compare several implementation alternatives



```
for (i0 = 0; i0<2; i0 ++)                    // output window row number
  for (i1 = 0; i1<4; i1 ++)                  // output window column number
    for (i2 = 0; i2<6; i2 ++)                // coefficient row number
      for (i3 = 0; i3<6; i3 ++)              // coefficient column number
        for (i4 = 0; i4<4; i4 ++)            // filter number

{result[i1][i4][i0]+ = data[2 × i0 + i2][2 × i1 + i3]× coeff[i4][i2][i3] ;}
```

```
for (i0 = 0; i0<2; i0 ++)                    // output window row number
  for (i2 = 0; i2<6; i2 ++)                  // coefficient row number
    for (i4 = 0; i4<4; i4 ++)                // filter number
      for (i3 = 0; i3<6; i3 ++)              // coefficient column number
        for (i1 = 0; i1<4; i1 ++)            // output window column number

{result[i1][i4][i0]+ = data[2 × i0 + i2][2 × i1 + i3]× coeff[i4][i2][i3] ;}
```

```
for (i0 = 0; i0<2; i0 ++)      // output window row number
  for (i2 = 0; i2<6; i2 ++)    // coefficient row number
    for (i4 = 0; i4<4; i4 ++)  // filter number
{result[1][i4][i0] + = data[2 × i0 + i2][0]× coeff[i4][i2][0];}
{result[2][i4][i0] + = data[2 × i0 + i2][2]× coeff[i4][i2][0];}
{result[3][i4][i0] + = data[2 × i0 + i2][4]× coeff[i4][i2][0];}
{result[4][i4][i0] + = data[2 × i0 + i2][6]× coeff[i4][i2][0];}
{result[1][i4][i0] + = data[2 × i0 + i2][1]× coeff[i4][i2][1];}
{result[2][i4][i0] + = data[2 × i0 + i2][3]× coeff[i4][i2][1];}
{result[3][i4][i0] + = data[2 × i0 + i2][5]× coeff[i4][i2][1];}
{result[4][i4][i0] + = data[2 × i0 + i2][7]× coeff[i4][i2][1];}

{result[1][i4][i0] + = data[2 × i0 + i2][2]× coeff[i4][i2][2];}
. . .
. . .
```

FIGURE 6: Example of loop transformations done on CS1, interpretation, and an associated datapath.

(e.g., degree of parallelism). The second feature enables us to describe our PE with the same dataflow as in our previous dataflow modelling, by specifying the data acquisition and the successive transfer between PEs as described earlier.

(3) *Data paths and low level parallelisation*

As explained in the previous paragraph, in order to enable UGH-CGS to parallelise computation on the available resources (DDP), a manual loop transformation and unrolling has to be done. For instance, in Figure 6 we present a loop nest permutation and unrolling that enables the exhibition of an inner loop with four independent MACs using the same kernel coefficient. Then, specifying in the DDP file that four multipliers and four adders are available, UGH-CGS is able to automatically generate four appropriate datapaths and all necessary control signals. An example of such a datapath is shown at the bottom-left of Figure 6.

Such transformations have been done for each CFF layer. Table 1 details for each CFF layer an example of loop nest permutation enabling UGH to parallelise the algorithm on four or eight datapaths.

(4) *Memory issues*

UGH memory models are only single or dual port memory. In order to be able to parallelise the computation on several datapaths, multiple memory banks have to be introduced, thus increasing the memory bandwidth (e.g., to read four distinct columns of input data for CS1, we use four memories). This can be easily managed with UGH by dispatching data on different C arrays. UGH generates one memory bank per C array and handles address generation for each memory. In this way, we control the memory access parallelism, by specifying exactly the number and size of the memory needed for our system.

Furthermore, address calculation has to be performed carefully in order to handle seven successive iterations, which can be simply done in C by computing the necessary addresses using increments and modulos (e.g., in CS1, four new lines are acquired and four previous lines are reused in the current iteration).

3.3. PE design case study

(1) *Algorithm description*

Using techniques described above, we describe the full CFF algorithm which consists of CS1, CS2, N1, and N2 layers. Each layer is described using a partially unrolled loop nest with MAC operations. Using an appropriate DDP, UGH successfully generates the FSM and the datapaths implementing the entire algorithm. We also define the appropriate I/O ports in the PE in order to handle input data acquisition, coefficient loading, CS1 and CS2 data transfers, and face detection results output.

(2) *Design exploration and performance analysis*

We use the flexibility offered by UGH to explore several architecture alternatives for the PE. This can be done by

Table 1: Low-level parallelisation of the CFF algorithm.

Layer (convolution size)	Number of windows (data parallelism)	Number of filters (task parallelism)	Total complexity (number of MAC)	Parallelisation scheme on 4 datapaths	Parallelisation scheme on 8 datapaths
CS1 (6×6)	8	4	1152	4 windows	4 windows and 2 filters
CS2 (4×4)	2	20	664	2 windows and 2 filters	2 windows and 4 filters
N1 (6×7)	2	14	1176	2 windows and 2 filters	2 windows and 4 filters
N2 (1×1)	2	14	28	2 windows and 2 filters	2 windows and 4 filters

Table 2: Cycle-time details for each PE version.

Layer	PE_A	PE_B	PE_C	PE_D
	Number of cycles/efficiency			
CS1	1344/0.86	740/0.78	437/0.66	261/0.55
CS2	738/0.87	412/0.81	251/0.66	252/0.33
N1 + N2	1346/0.89	749/0.80	457/0.66	466/0.33

modifying the available resources given in the DDP, and by doing some transformations in the C source code. In order to compute the convolutions, we describe datapaths in the DDP, with MAC blocks performing multiply accumulate operations. Each datapath contains a MAC block and one to four register files. Each register file is composed of four 16 bits registers. We have explored four datapath alternatives to build the following PE versions.

(i) PE_A: 1 datapath only with 1 MAC block, 4 register files.

(ii) PE_B: 2 datapaths, each with 1 MAC block, 2 register files.

(iii) PE_C: 4 datapaths, each with 1 MAC block, 1 register file.

(iv) PE_D: 8 datapaths, each with 1 MAC block, 1 register file.

Figure 7 depicts the architecture of PE_C. Dotted lines and boxes represent everything that is automatically handled by UGH: links, FSM, addresses, and so on. The remaining grey parts are provided in the DDP. TH blocks represent activation functions which are applied after each convolution.

We then obtain 4 PE versions containing 1, 2, 4, and 8 MAC blocks, respectively. We have validated the VHDL codes generated by UGH using the same test sets as in the behavioural initial C description of the PE and in a VHDL test bench. The processing cycle times for each version are summed up in Table 2, where each layer of the CFF algorithm is detailed. We also point out in this table the efficiency of parallelisation, by computing the theoretical minimum number of cycles T_{th} achievable for each CFF layer (the number of MAC operations given in Table 1 divided by the number of MAC blocks); the efficiency is then equal to $\text{Eff} = T_{\text{th}}/T_{\text{par}}$. We can see in Table 2 that the average efficiency of PE_A is about 0.9. As PE_A contains only 1 MAC, it allows no possible parallelisation. The efficiency of PE_A can be interpreted as follows. 90% of the time is used for computing the MACs and the remaining 10% corresponds to controlling overhead due to the handling of loops and addresses. PE_B and PE_C have average efficiencies of 0.8 and 0.7, respectively, which corresponds to average acceleration factors of 1.6 and 2.8. This efficiency remains high considering the complexity of the algorithm. However, we can point out that for PE_D, there is a significant loss in efficiency which can be interpreted as follows. The CS1 layer is successfully parallelised on 8 MACs but for CS2 and N1 N2, memory bandwidth is not large enough to enable an efficient use of 8 MACs simultaneously. For instance, an efficient parallelisation of N1N2 on 8 MACs would require a simultaneous access to four distinct coefficients and eight distinct data. This could be addressed by using data and coefficient buffer memories to provide more local parallelism and reusing of data.

Synthesis results for each version are given in Table 3 for a Virtex-4 SX35 FPGA. No major modification has been made on the VHDL code generated by UGH. Implementation memory banks are balanced between flip flops, distributed RAM (RAM implemented in logic blocks [14]), and block RAM (embedded static RAMs), depending on the memory bank size and the number of ports needed. DSP48 blocks are used to implement MAC blocks in the datapaths. The maximum frequency of each version of the PE is 80 MHz and is mainly due to unpipelined utilisation of DSP48 blocks and the large number of memories which imply much multiplexing logic. From Table 3, we see that synthesis results are close from one version to another. This is due to the fact that the size of the PEs is mostly because of the size of the control unit (finite state machine) and of the multiplexers, rather than the datapaths which are only MAC blocks and registers. Hence, we can conclude that PE_C is a good compromise as it enables a high acceleration factor (2.7) and requires about the same resources as PE_A and PE_B. Therefore, we choose PE_C as the best implementation for the PE (in the rest of the paper, we will refer to PE instead of PE_C) because it achieves a good tradeoff between size and efficiency.

Table 3: Synthesis results for each PE version on Virtex-4 SX35 FPGA.

	PE_A	PE_B	PE_C	PE_D
	Device occupancy/capacity			
Number of slices	2258/15360	2259/15360	2466/15360	2866/15360
Number of flip flops	363/30720	388/30720	345/30720	457/30720
Number of block rams	15/192	19/192	19/192	19/192
Number of DSP48	5/192	6/192	8/192	12/192
Maximum frequency	80 MHz	80 MHz	80 MHz	80 MHz

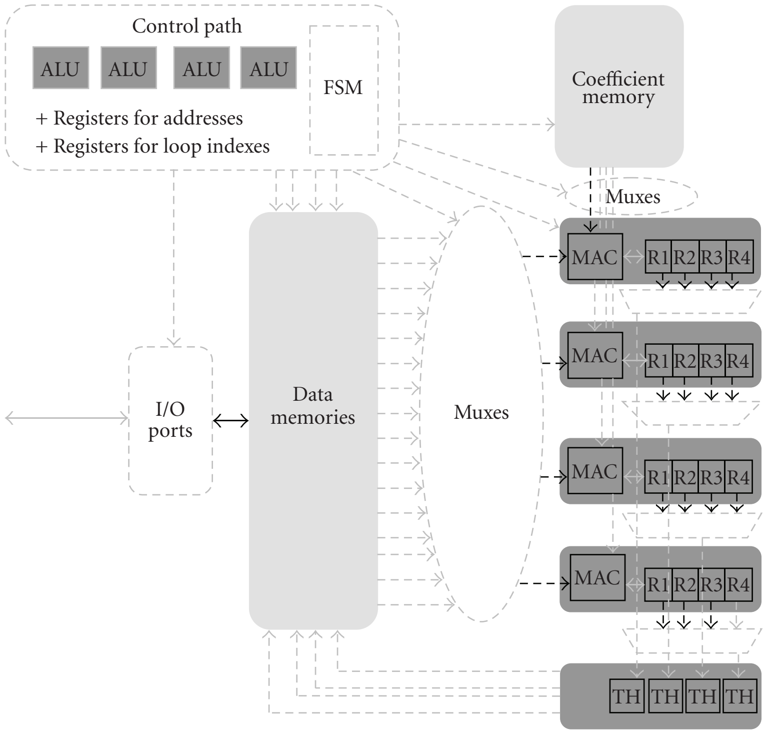


Figure 7: Architecture of a PE with 4 datapaths (PE_C).

4. MULTI-PE PARALLEL SYSTEM

In this section, we present a parallel architecture based on the previously designed PE. In Section 4.1, we describe this architecture and its application for processing a whole image of any size. In Section 4.2, we give its performances, in terms of frame processed per second, for different image sizes, and finally, we compare these performances to other face detection systems.

4.1. Generic parallel architecture

In previous work [6], we have shown that a good tradeoff between efficiency and number N_{PE} of PE is obtained when the input image is divided into $P = N_{\mathrm{PE}}$ blocks of 8 rows of 12 pixels. Each block is processed by one PE, and each PE is connected to two other PEs, thus building a ring architecture. Considering data overlapping, this allows the processing of an image of width $12+(N_{\mathrm{PE}}-1)*8$ and of any height greater or equal to 36 (the minimum number of rows necessary to compute a face detection). We rely on this result to establish a generic architecture using a ring of PEs and a FIFO memory (Figure 8).

We divide the input image into vertical strips of width $12 + (N_{\mathrm{PE}} - 1)*8$ and process each strip by dividing it into blocks of 12×8 as described above. The FIFO is connected to the first and the last PEs of the ring and is used to provide the overlapping data from the previous strip and to write the overlapping data for the next one (grey blocks in the top of Figure 8). We have successfully simulated a 4 PE ring with FIFO. Synthesis estimates show that a ring of 25 PE fits in a Virtex-5 LX 330 device, the largest FPGA available.

4.2. Performances

In this architecture, each PE works synchronously. At each vertical iteration, four new input lines are loaded in the PEs and if we consider that this load is done in pipeline with the

TABLE 4: Comparison of hardware face detection implementation.

Implementation	Frame size	Frame rate	Clock freq. (MHz)
CFF embedded software [3]	176 × 144	10	624
Kianzad et al. [15]	320 × 240	12	125
McCready [16]	320 × 240	30	12,5
Theocharides et al. [4]	320 × 240	52	500
Nguyen et al. [5]	320 × 240	42	12,5
Hori et al. [17]	320 × 240	30	100
CFF 4 PE ring	320 × 240	**30**	**80**
CFF 25 PE ring	320 × 240	**127**	**80**
CFF 25 PE ring	640 × 480	**35**	**80**

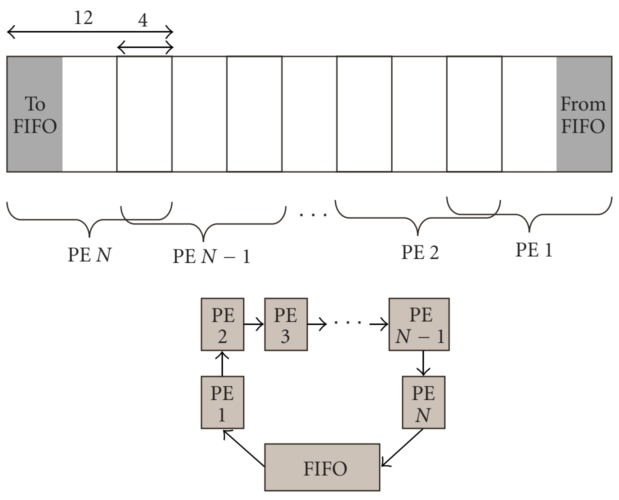


FIGURE 8: Generic ring and a FIFO architecture.

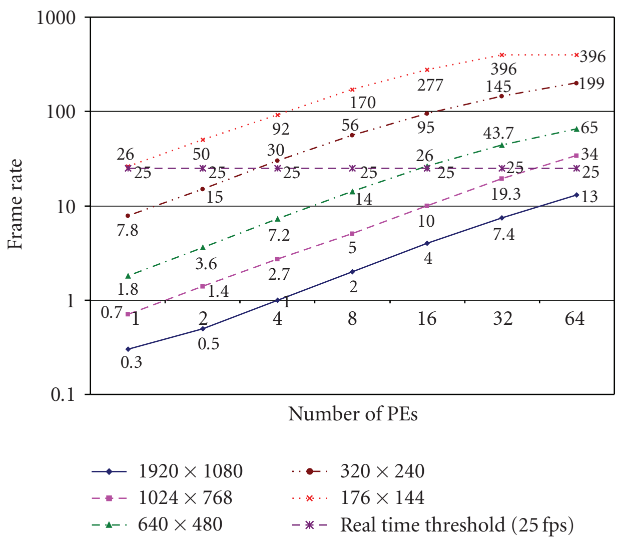


FIGURE 9: Performances of PE ring architecture at 80 MHz.

algorithm computation, we can give the processing time of the algorithm on a complete image of size width L and height M using the following formulae.

(i) T_{CFF}: time to process the CFF algorithm on one 8×12 block.

(ii) Number of vertical iterations: $N_{\text{it}} = (M - 4)/4$.

(iii) Number of vertical strips per PE: $N_{\text{str}} = (L - 4)/(8 \times N_{\text{PE}})$.

(iv) Processing time of a strip: $T_{\text{str}} = N_{\text{it}} * T_{\text{CFF}}$.

(v) Total processing time: $T_{\text{tot}} = N_{\text{str}} * T_{\text{str}}$.

In order to detect faces of different sizes, an image pyramid is built to apply the CFF algorithm to each image (see [3]). This pyramid is constituted of a set of subimages obtained from the initial image by applying a reduction factor of 1.2 to each dimension. We can obtain an estimation of the total processing time of the entire pyramid by adding together the processing times of each subimage.

In Figure 9, we sum up the performances (in frames per second) of this parallel architecture on several image sizes and for 1 to 64 PEs (to improve image readability both axes have logarithmic scales). We also represent the real-time threshold of 25 fps on this figure. An architecture consisting of four PEs fits in a Virtex-4 SX35 FPGA and performs real-time face detection at 30 QVGA images per second. When considering a larger system with 25 PEs, we can process 127 QVGA images per second, which leaves enough time to consider other face analysis tasks in real-time following face detection.

Table 4 presents a comparison between the main hardware implementations of face detection found in literature. All these implementations are done on FPGA hardware and therefore run at relatively low clock frequencies (except [4] which is an ASIC implementation). We can see that our system compares well with the others in terms of frame rate, depending on the number of PEs implemented. The main drawback of the other implementations is usually a significant loss in overall detection rates or even a lack of detection performance measure. most works do not give results of detection accuracy of their hardware implementation. Contrary to this, our system has the same detection performances as the original CFF software implementation [3], therefore it is very robust in terms of detection accuracy and has a very low false alarm rate [2].

Area comparison is not straightforward since the FPGA targets of the considered implementations are different. In [16], the authors report an area of 89856 Logic Elements and approximately 442 Kbits of memory on a system which contains eighteen 10k100 Altera FPGAs. In [5], the authors report an area of 15050 Logic Elements and 268 Kb of memory on a Altera Stratix FPGA. Our 4-PE ring on Virtex-4 uses approximately 8802 Xilinx Slices and about 1272 Kb of embedded memory. We cannot make a direct comparison between these implementations but a rough estimate can be given by considering that one Altera Logic Element is the equivalent of one to two Xilinx Virtex-4 Slices. Hence, our system has a slightly higher hardware cost in terms of logic and uses much more embedded memory. This is due to the large number of coefficients and temporary results which have to be stored in the FPGA. However, as said earlier, our system is capable of a very robust face detection with complex backgrounds.

5. CONCLUSIONS—PERSPECTIVES

We have implemented a parallel architecture for face detection composed of Processing Elements based on the CFF algorithm. Using a high-level synthesis approach, we were able to explore several PE architecture alternatives. We selected a PE with four datapaths exploiting efficiently local parallelism of the algorithm. This PE has been successfully simulated and synthesised on a Virtex-4 SX35 FPGA, and occupies approximately 16% of the device, and is capable of running at a maximum frequency of 80 MHz. We have then presented a ring of PE with a FIFO memory, which constitutes a generic parallel and scalable architecture able to process images of variable sizes. Such an architecture with four PEs can process up to 30 QVGA images per second. An architecture with 25 PE achieves real-time face detection of VGA images.

We are currently working on optimising this PE in order to be able to implement more PE in our targeted FPGAs, and to increase its maximum clock frequency. In future work, we will investigate the generalisation of such modelling and architectures for other algorithms based on convolutional neural networks. We will also investigate the opportunity of using source-to-source transformations in order to enhance the user-guided approach proposed by UGH, by providing a semiautomated tool which transforms the C code before inputting UGH; such techniques can be used to ease memory partitioning and loop unrolling.

ACKNOWLEDGMENTS

The authors would like to thank Clément Quinson for his help in starting up with UGH, as well as Frédéric Pétrot and Ivan Augé of the UGH team for their help with specific points on the UGH tool, and their encouragement on this work.

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] 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.

[3] S. Roux, F. Mamalet, and C. Garcia, "Embedded Convolutional Face Finder," in *Proceedings of IEEE International Conference on Multimedia and Expo (ICME '06)*, pp. 285–288, Toronto, Canada, July 2006.

[4] 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 (VLSID '04)*, pp. 133–138, Mumbai, India, January 2004.

[5] D. Nguyen, D. Halupka, P. Aarabi, and A. Sheikholeslami, "Real-time face detection and lip feature extraction using field-programmable gate arrays," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 36, no. 4, pp. 902–912, 2006.

[6] N. Farrugia, F. Mamalet, S. Roux, F. Yang, and M. Paindavoine, "A parallel face detection system implemented on FPGA," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '07)*, pp. 3704–3707, New Orleans, La, USA, May 2007.

[7] V. Kathail, S. Aditya, R. Schreiber, B. Ramakrishna Rau, D. C. Cronquist, and M. Sivaraman, "PICO: automatically designing custom computers," *Computer*, vol. 35, no. 9, pp. 39–47, 2002.

[8] S. McCloud, "Using a catapult c-based flow to speed implementation and increase flexibility," Tech. Rep., Mentor Graphics, Wilsonville, Ore, USA, 2003.

[9] "Dk design suite datasheet," Agility Design System Inc., http://agilityds.com/literature/dk_datasheet_01000_hq_screen.pdf.

[10] P. Coussy, G. Corre, P. Bomel, E. Senn, and E. Martin, "High-level synthesis under I/O timing and memory constraints," in *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS '05)*, vol. 1, pp. 680–683, Kobe, Japan, May 2005.

[11] I. Augé, F. Pétrot, F. Donnet, and P. Gomez, "Platform-based design from parallel C specifications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1811–1826, 2005.

[12] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time-series," in *The Handbook of Brain Theory and Neural Networks*, pp. 255–258, MIT Press, Cambridge, Mass, USA, 1998.

[13] S. Girbal, N. Vasilache, C. Bastoul, et al., "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 261–317, 2006.

[14] "Virtex-4 user guide, chap. 5: Configurable logic blocks (clbs)," June 2008, http://www.xilinx.com/.

[15] V. Kianzad, S. Saha, J. Schlessman, et al., "An architectural level design methodology for embedded face detection," in *Proceedings of the 3rd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '05)*, pp. 136–141, Jersey City, NJ, USA, September 2005.

[16] R. McCready, "Real-time face detection on a configurable hardware system," in *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL '00)*, pp. 157–162, Villach, Austria, August 2000.

[17] Y. Hori, K. Shimizu, Y. Nakamura, and T. Kuroda, "A real-time multi face detection technique using positive-negative lines-of-face template," in *Proceedings of the 17th International Conference on Pattern Recognition (ICPR '04)*, vol. 1, pp. 765–768, Cambridge, UK, August 2004.