Intelligent cameras and embedded reconfigurable computing: a case-study on motion detection

Claudio Mucci, Luca Vanzolini, Antonio Deledda
ARCES - University of Bologna
Viale Pepoli 3/2, Bologna, Italy

Fabio Campi
STMicroelectronics
Viale Olivetti 2, Agrate Brianza (MI)

Gérard Gaillat
THALES Optronique S.A.
78283 Guyancourt Cedex, France

***Abstract*— Image processing for intelligent cameras like those used in video surveillance applications implies computational demanding algorithms activated in function of non predictable events, such as the content of the image or user requests. For such applications, hardwired acceleration must be restricted to a minimum subset of kernels, due to the increasing NREs when application update become necessary. Embedded reconfigurable processors, coupling in the same computing engine a general-purpose embedded processor and field-programmable fabrics, provide an appealing trade-off point between pure software and dedicated hardware acceleration. As a case-study, this paper presents the implementation of a set of image processing operators utilized for motion detection on the DREAM adaptive DSP. With respect to pure software solutions, the proposed implementation achieves a performance improvement of 2-3 orders of magnitude, while retaining the same degree of programmability and the same economical perspectives from the end-user point of view of processor-based approaches.**

I. Introduction

Image processing systems for intelligent cameras are becoming very important for a wide spectrum of vision applications, like video surveillance, computer vision, object tracking, face or iris recognition. Even if this task may appear trivial for humans, for a computing device detecting an alien presence moving in a surveilled area or distinguishing a rose from a tulip are very complex application. Most of the computation time is spent extrapolating specific features from the images, in order to reduce data sizes while retaining most relevant informations. In many applications, the image processing flow may depend on real-time events. As an example, motion detection could activate a procedure for object/human recognition. Traditional implementation of this kind of image processing algorithms consists in utilizing standard processors as control engines and FPGAs as programmable accelerators. In fact, motion detection systems normally require average-low implementation costs, and such an approach is acceptable only for simple processing systems with a limited number of operating modes. For example, some operators could be rarely used, but being very computationally demanding they could require hardware acceleration. In this case, the operator would then be often inactive while occupying a large share of the technology resource. A relevant feature of reconfigurable processor such as the one presented in this paper is to provide very fast and efficient dynamic reconfiguration, allowing the user to extensively exploit time multiplexing over a given set of silicon resources. For intelligent cameras reconfigurable processors provide an appealing alternative. They allow intensive run-time re-use of the acceleration logic, properly configurated to the specific (and possibly event-driven) required task.

In this paper, a motion detection algorithm requiring more than 10 different basic operation kernels is considered as a case-study in order to show the possibilities offered by reconfigurable computing, and to analyze the different design trade-offs. A key point of this work is that impressive performance improvements with respect to a software solution (2-3 order of magnitude) were achieved after (roughly) 1 month of work mostly performed by an unexperienced MSc student. In the authors opinion, this proves how reconfigurable processor are becoming an accessible and cost-effective solution open to the wide community of software and DSP programmers rather than a niche solution hardly exploitable in common applications but useful only for hardware designers challenging massively parallel applications.

II. The DREAM Adaptive DSP Architecture and Programming Approach

Many commercial SoC solutions in the field of image processing or wireless telecommunication (e.g. ST Nomadik, TI OMAP, Philips Nexperia) feature today architectures based on a main control processor (commonly an ARM processor), accelarated by a constellation of dedicated circuits or Application Specific DSPs. From an economical perspective, this trend is very promising also for the incoming years, and in this scenario we propose the DREAM adaptive DSP, developed in the Morpheus project timeframe [2]. The DREAM [1] adaptive DSP is reconfigurable processor designed for integration in state-of-art SoCs in order to provide additional functionalities and increased flexibily with respect to Application-Specific cores. The architecture is sketched in Figure 1, which highlight the three basic concepts of DREAM:

- a **reconfigurable datapath** charged to speed-up the computationally intensive parts of the application.
- a **programmable high-bandwidth memory architecture** which allows to exchange data to/from the reconfigurable device with up to 16x32 bit/cycle

The work presented in this paper is done within the MORPHEUS project (IST FP6, project no. 027342), which is sponsored by the European Commission under the 6th Framework program.

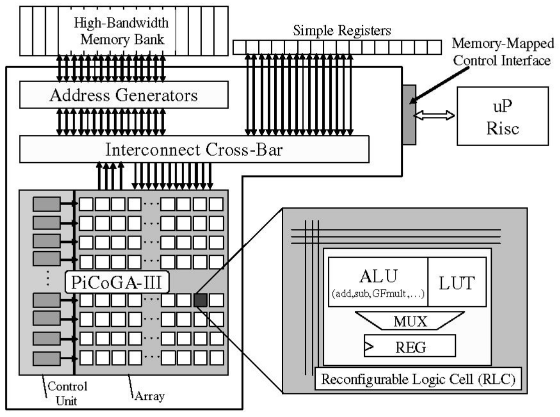

Fig. 1. DREAM simplified architecture

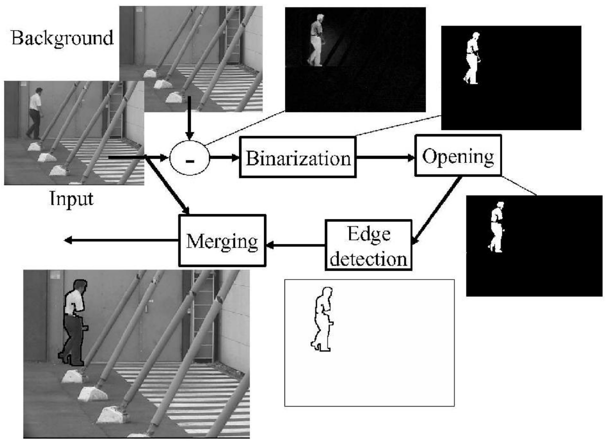

Fig. 2. Simplified motion detection algorithm overview

- a **32-bit RISC processor** which locally handles the computation, thus allowing to build kernels with local controls, reducing at the same time the control overhead on the main processor of the SoC.

In the case of DREAM, the reconfigurable datapath is implemented by the Pipelined Configurable Gate Array (PiCoGA), a 4-context array of 16x24 reconfigurable logic cells (RLC). The array performs computation according to a dataflow paradigm. It is programmed utilizing a simplified C-syntax termed Griffy-C [3] that is utilized to describe Data Flow Graphs hiding to the user technology and implementation details. The aim of this design choice is to allows software programmers to develop application on the device without specific hardware design skills. The basic idea is to feature a standard ARM-based SoC infrastructure providing connection to on-chip or off-chip memories and IO units that handles system-level data communication while the DREAM accelerator (as well as other ASIC/DSP/ASSP units) challenges computationally intensive kernels. For this reason data flowing through PiCoGA are temporary stored in a set of 16 local buffers (1024 32-bit words each). The buffers are implemented as dual port memories addressed by programmable address generators on the PiCoGA side, while the system side they are seen as a single memory entity accessed via standard bus connection, DMAs and/or Network-on-Chip. The usage of such buffers instead of commonly utilized FIFOs adds flexibility to the infrastructure since it allows to significantly increase locality of the computation, as it will appear evident in the test-case here presented.

III. Motion Detection Algorithm

The Motion detection algorithms provides the capability to detect a human, a vehicle or an object in movement with respect to a static background. This could be useful for example to activate an alarm in case of security applications or to start the recording in case of area monitoring system. A typical motion detection algorithm is shown in Figure 2. Most of the processing is performed on the image resulting from the absolute *pixel-to-pixel difference* between the current frame and the background, which can be stored during the setup of the intelligent camera. Even if this differentiation allows to isolate the object under analysis (if any), too many details are present as the complete grayscale. For that, *binarization* is applied to the frame. Given a threshold conventionally fixed to $0.3\times$ the maximum value of the pixels, binarization returns an TopValue if the current pixel is greater than the threshold, and a BottomValue otherwise. The resulting image could still be affected from noise (spurious pixels) or on the contrary could be affected by some hole. This “cleaning” task is acomplished by the *opening* phase, implemented by two operators:

- **erosion**, that working on 3x3 pixel matrices substitues the central pixel with the minimum in the matrix, thus removing random noise.
- **dilatation**, that working on 3x3 pixel matrices substitues the central pixel with the maximum in the matrix, closing eventual small holes and reinforcing details.

The next step is the *edge detection* which allows identification of the boundaries of the human or object that is moving in the monitored area. This operation is implemented by a simple convolution applied to 3x3 pixel matrices using the Sobel algorithm. The Sobel edge detector uses two convolution kernels to detect vertical and horizontal edges defined as:

$$E(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} K(i,j) * p(x-i, y-j)$$

where $E(x,y)$ is the pixel under elaboration, $p(h,k)$ represents the pixel in the 3x3 matrix centered in (x,y), and K is the Sobel matrix, for horizontal and vertical edge detection, defined as:

$$K_h = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}; K_v = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

The gradient is then computed approximating the combination of the two components, as in the following formula:

TABLE I

SOFTWARE-ONLY PERFORMANCE EVALUATION

Kernel	Cycles/pixel	%
Absolute difference	19	2.6%
Max pixel value	9	1.3%
Binarization	11	1.5%
Erosion	137	19.2%
Dilatation	136	19.0%
Inv.Bin. Edge detection	393	55.0%
Merging	10	1.4%
Total	715	100.0%

$$E = \sqrt{E_h^2 + E_v^2} \simeq |E_h| + |E_v|$$

The resulting image is then *binarized*, since the aim of the application is not to detect the magnitude of the gradient but the presence of a gradient. Finally the detected edge is *merged* with the original image. For that goal, inverse binarization is applied: the background is filled by 1s and moving image edges by 0s, thus allowing to implement the merge operation with a multiplication. Table I shows the computational requirement for motion detection on a MIPS-like RISC processor.

IV. MOTION DETECTION ON DREAM

The implementation or more in general the acceleration of the above described application on the reconfigurable platform, is driven by two main factors:

- **instruction/data level parallelism**: each operation (e.g. binarization, edge detection, ...) shows relevant instruction level parallelism. Given a specific image processing kernel, computation associated to each pixel is independent from the elaboration of other pixels, although the reuse of adjacent pixel proves beneficial to minimize memory access.
- **data size**: after binarization, the information content associated to each pixel can be represented by only 1 bit (edge/no edge), thus allowing to store up to 32 pixel in a 32-bit word. This significantly reduces memory utilization without implying additional packing/unpacking overhead, as it would be the case with 32-bit processors, since DREAM may handle shifts by programmable routing.

This last consideration provides additional benifits since:

- the *erosion* phase requires the search of the minimum within the pixels in a 3x3 matrix, but is implemented on DREAM by a single 9-bit input 1-bit output AND;
- the *dilatation* phase requires the search of the maximum within the pixels in a 3x3 matrix, but is implemented on DREAM by a singel 9-bit input 1-bit output OR;
- edge detection can be strongly simplified as explained in the following.

Inverse-binarized edge detection can be represented by:

TABLE II

COMPLEXITY ANALYSIS FOR N COLUMNS, M ROWS IMAGE PROCESSING

Kernel	Complexity
Absolute difference	$\left(\frac{N*M}{3*4}\right) + t_{ov};\ t_{ov} = 89$
Max pixel value	$\left(\frac{N*M}{3*4}\right) + t_{ov};\ t_{ov} = 51$
Binarization	$\frac{M}{3} * \left(\frac{16*N}{32} + t_{ov2}\right) + t_{ov1};\ t_{ov1} = 36;$ $t_{ov2} = \begin{cases} 0 & N\ mod\ 32 = 0 \\ 66 & N\ mod\ 32 \neq 0 \end{cases}$
Erosion	$3 * \left\lceil \frac{N}{32} - 1 \right\rceil + \left\lceil \frac{M}{3} - 1 \right\rceil \left(\frac{N}{32} - 1 + t_{ov2}\right) + t_{ov1}$ $t_{ov1} = 88;\ t_{ov2} = 95$
Dilatation	$3 * \left\lceil \frac{N}{32} - 1 \right\rceil + \left\lceil \frac{M}{3} - 1 \right\rceil \left(\frac{N}{32} - 1 + t_{ov2}\right) + t_{ov1}$ $t_{ov1} = 88;\ t_{ov2} = 95$
Inv.Bin. Edge detection	$3 * \left\lceil \frac{N}{32} - 1 \right\rceil + \left\lceil \frac{M}{3} - 1 \right\rceil \left(\frac{N}{32} - 1 + t_{ov2}\right) + t_{ov1}$ $t_{ov1} = 91;\ t_{ov2} = 98$
Merge	$\frac{M}{3}\left(\frac{N}{4} + t_{ov1}\right) + t_{ov}$ $t_{ov} = 74;\ t_{ov1} = \begin{cases} 14 & N\ mod\ 32 = 0 \\ 18 & N\ mod\ 32 \neq 0 \end{cases}$

$$\begin{aligned} IB(x,y) &= \begin{cases} 0 & if E(x,y) > 0 \\ 1 & if E(x,y) = 0 \end{cases} \\ &= \begin{cases} 0 & if |E_h(x,y)| + |E_v(x,y)| > 0 \\ 1 & if |E_h(x,y)| + |E_v(x,y)| = 0 \end{cases} \end{aligned}$$

Since each pixel can be represented by 1 bit, the result of horizontal and vertical Sobel convolution is in the range $[-4, +4]$ requiring 4 bits. Moreover, it should be noted that given E_h and E_v components, IB(x,y) will be 1 if and only if all the bits of E_h and E_v are $0s$, making possible to implement this computation by an 8-input NOR. As a consequence the final merging phase will be implemented by an 8-bit bitwise AND operation, instead of an 8-bit multiplication.

The processing chain is based on simple operations repeated many times for all the pixels in a frame. In the case of DREAM and PiCoGA, all these operations do not fully exploit parallelism made available by the reconfigurable device. It is thus possible to operate concurrently on more pixels at time unrolling inner loops in the computation flow. As a consequence, PiCoGA needs to receive data with higher bandwidth, and this becomes the first factor limiting the level of unrolling. The optimal level of unrolling is determined considering that $\sim$95% of the computation time is spent on operations working on a 3x3 pixel matrix. In order to effectively sustain high throughput we need to be able to properly pipeline computation on PiCoGA. We concurrently compute on three different rows at a time for all the computations, since most of the operations requires pixels from 3 adjacent rows:

- erosion, dilatation and edge detection read data from 3 adjacent rows and provide the result for the row in the middle. In this case, since each pixel is represented by 1 bit, we elaborate $3 * 32 = 96$ pixels per PiCoGA operation, packing 32 pixels in a single 32-bit memory word stored in the local buffer.

TABLE III
DREAM-BASED IMPLEMENTATION SUMMARY

	80x60 chunk	
Kernel	Cycles/pixel	SpeedUp
Absolute difference	0.11	173×
Max pixel value	0.09	100×
Binarization	0.45	24×
Erosion	0.42	326×
Dilatation	0.42	326×
Inv.Bin. Edge detection	0.43	914×
Merging	0.17	59×
Total	2.09	342×

- the other operation can work on 3 adjacent rows to maintain a certain degree of regularity in the data organization. In this case, 8 bits are used to represent a pixel and we can pack 4 pixel in each memory word, resulting in an elaboration of 12 pixel at time.

To allow the concurrent access to 3 adjacent rows, we use a simple 3-way interleaving scheme in which each row is associated to a specific buffer by the rule: $buffer\_index = \#row\ mod\ 3$. Rows are stored contiguously in each buffer, and each PiCoGA operation read a row chunk per cycle. Referring to 1, address generators are programmed in order to scan the buffers where rows are stored according to the above described access pattern, while programmable interconnect is used to dynamically switch between referenced rows. Boundary effects due to the chunking are handled internally to PiCoGA that can hold the pixels required for the different column elaborations in internal registers, thus avoiding data reread. Depending on the size of the frame, the available level of pipelining increases, augmenting the number of pixels in a row. As a consequence, the amount of memory required to store the frame under elaboration also increases. In particular, the DREAM buffers may store up to 640x480 binarized frames, allowing to localize most of the memory access with an immediate gain in energy consumption. Bigger frames can be elaborated in chunks, performing the computation on subframes.

V. RESULTS

Table II reports the computational complexity of motion detection on DREAM for the different required kernels, parametrized in terms of the frame size (NxM). As an example, Table III shows the cycle-count in the case of 80x60 chunks. This is an interesting sub-case since all necessary data for computation can be stored locally to DREAM, including both the original frame and the background which are the most demanding contributions in term of memory requirements. Cycle-counts are reported normalized with respect to the number of pixels in the image, to give a direct comparison with respect to the pure software implementation in Table I. In addition, Fig. 3 shows the potential performance gain for bigger frames in terms of cycles/pixel reduction. Speedups range from 342× to ∼1200×. It should be noted that larger images cannot be stored entirely on the DREAM memory sub-system, although the packetization performed after binarization allows to hold internally the most critical part of the computation up to the frame size 640x480. Fig. 4 shows the dramatic reduction in memory access provided by the DREAM solution. For the software solution, memory transfers are considered for pixel access only, without including accesses due to temporary results, stack management and so on. Considering the overall motion detection application, the DREAM solution needs roughly 2 memory accesses per pixel, wheras a software implementation requires ∼39 memory accesses per pixel. We also considered a software-optimized SIMD-like access in which the processor could be able to handle 4-pixel per cycle (also during comparison). Also in this case our reconfigurable solution achieves ∼80% memory access reduction, with a consequent benefit in terms of energy consumption.

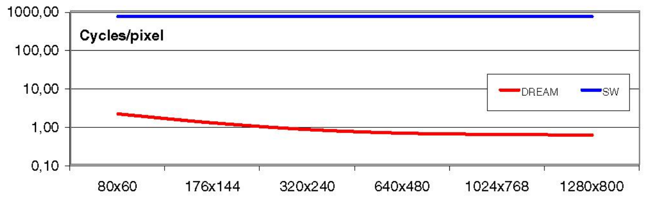

Fig. 3. Performance gain vs. frame size

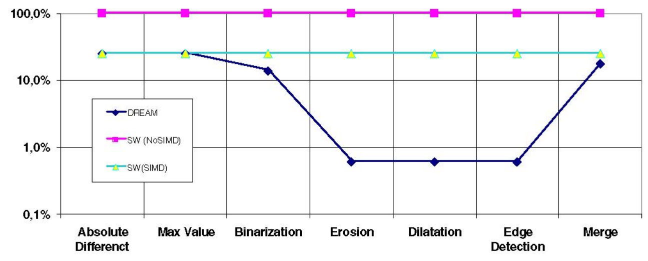

Fig. 4. Normalized memory access per pixel

VI. CONCLUSION

In this paper we presented embedded reconfigurable computing as a way to accelerate image processing in application like intelligent camera where hardwired solutions often show lack of flexibility. As a test-case, we considered a motion detection engine. Performance evaluation for a basic 80x60 frame showed a 342× speed-up with respect to a processor based solution, and ∼90% reduction in memory access. Bigger frames can achieve better performance figure (up to 1200×), with the drawback of an additional memory demand. In the middle, near-optimal solutions can mantain a good level of performance with the capability to hold intermediate results on the DREAM memory sub-system with a direct gain in energy consumption and communication congestions.

REFERENCES

[1] F. Campi et al. *A dynamically adaptive DSP for heterogeneous reconfigurable platforms*, Proceedings on IEEE/ACM DATE 2007.
[2] Morpheus project web-site, *www.morpheus-ist.org*
[3] C. Mucci et al. *A C-based Algorithm Development Flow for a Reconfigurable Processor Architecture*, IEEE Int'l Symposium on System on Chip, Nov 2003.