DSP implementation of real time edge detectors

V. GEMIGNANI [(1)], M. DEMI[(1)(2)], M. PATERNI[(1)], M.GIANNONI[(1)], A. BENASSI[(1)]
[(1)] CNR - Institute of Clinical Physiology , v. Moruzzi 1, Pisa
[(2)] Esaote SpA, Florence
ITALY

Abstract: - In image processing, the real time execution of contour tracking algorithms is a challenging operation. In fact, the detection and tracking of an edge through a sequence of images, require the performing of demanding algorithms. This paper presents the real time implementation of the two mathematical operators which we commonly use to detect edges: (i) the "gradient of Gaussian", which is the operator mostly used in literature and (ii) the "**b** operator", a new operator which we have been using for a few years. Their algorithms are implemented on the TMS320C64x, which is a state of the art DSP in image processing applications. Timings are reported of the two algorithm implementations.

Key-Words: edge detector, DSP, real time, contour tracking.

1 Introduction

Image analysis often involves the automatic contour tracking of the object under investigation. Usually, a contour tracking algorithm requires: 1) *filtering* the images by means of a derivative operator to obtain image maps where the discontinuities between the gray levels of different structures are enhanced, and 2) *localizing* the points of the contour one is looking for by examining the previous maps [1,2]. According to this approach, many authors have faced the problem and many contour tracking procedures have been proposed. For example, the magnitude of the gradient of Gaussian (GoG) is one of the edge operators most widely used in the filtering process. As regards the localization procedure, many authors have faced the problem of locating the contour by starting from an approximate contour of the object under investigation. The approximate starting contour on one frame is usually obtained by exploiting the contour data computed on the previous frames. Other authors, however, have exploited the availability of a priori knowledge of the global shape of the objects to develop localization procedure based on a model of the object.

However, no matter which localization procedure is chosen, they frequently include complex regularization algorithms, so that large computational resources are necessary to obtain real time implementations. In a previous paper [3], a system for real time contour tracking was presented. The hardware platform adopted was a standard Personal Computer equipped with a Digital Signal Processor (DSP) board based on the TMS320C80 multi processor. A new mathematical operator, which we will call **"b** operator", was used in the filtering process. A contour which had been determined on a frame of the sequence was used as the approximate starting contour to locate the contour on the subsequent frame and the operator proved to be robust to noise and particularly suitable for a real time implementation. Additional constraints on the movement and on the shape of the contour were introduced to regularize the procedure and consequently, to increase its robustness to noise. The system was tested on sequences of clinical images and it proved to be a versatile tool to evaluate vascular silhouettes in real-time. However, when further developments of the real time procedure were considered, we ran into two problems: the computational resources were insufficient to obtain real time performances when using more demanding regularization algorithms and the work necessary to implement more complex algorithms on the DSP we adopted was quite onerous indeed.

Nowadays, the Texas Instruments TMS320C6000 platform provides some of the fastest DSPs on the market, so that the TMS320C64x was considered to re-implement a new procedure. In this paper, however, only the filtering process is considered and the implementations of the GoG operator and of the **b**

operator are presented. The execution time of the two algorithms is computed and a comparison is reported. Moreover, a comparison with a software implementation on a standard Personal Computer is reported. It is a known fact that the CPUs of the PC are becoming more and more powerful and consequently, it is important to take into account a PC implementation also since it is almost always the simplest and most reasonable solution.

To conclude, finally, a comparison is reported with the implementation of the **b** operator on the TMS320C80 which we illustrated in a previous paper.

2. Edge Detectors

In this section, the edge detection algorithms based on the GoG operator and on the **b** operator are briefly described.

Let $f(n,m)$ be the gray level map of an image and let $g(n,m)$ be a Gaussian function. The square magnitude of the gradient of Gaussian of the function $f(n,m)$ is defined as:

$$G = \left|\nabla(f * g)\right|^2 \qquad (1)$$

From (1) we obtain equation (2) which is used to perform the filtering process.

$$G(n,m) = \\ = \left(\sum\sum_{(k,l)\in\Theta} f(n-k,m-l)\cdot g_x(k,l)\right)^2 + \\ + \left(\sum\sum_{(k,l)\in\Theta} f(n-k,m-l)\cdot g_y(k,l)\right)^2 \qquad (2)$$

$g_x(k,l)$ and $g_y(k,l)$ are the derivatives of the Gaussian function with respect to x and y and they are defined in a neighborhood Θ of the point (n,m).

To introduce the **b** operator, let us consider two circular domains Θ_1 and Θ_2 of the point (n,m). Let $g_1(k,l)$ and $g_2(k,l)$ be two Gaussian weight functions with a unitary summation on Θ_1 and Θ_2 respectively. Let us compute the mean value of $f(n,m)$ on the circular domain Θ_1 at a point $\mathbf{p}=(n,m)$

$$f_1(\mathbf{p}) = \sum\sum_{(k,l)\in\Theta_1} f(n-k,m-l)g_1(k,l) \qquad (3)$$

Let us now associate every pixel of the circular domain Θ_2 to a mass function $h(\mathbf{p},k,l)$ so that:

$$h(\mathbf{p},k,l) = \left|f_1(\mathbf{p}) - f(n-k,m-l)\right| g_2(k,l) \qquad (4)$$

The function $h(\mathbf{p},k,l)$ represents the spatial distribution of the variability of the gray levels of the domain Θ_2 with respect to the local mean $f_1(\mathbf{p})$ computed at point $\mathbf{p}$. The center of mass of the function $h(\mathbf{p},k,l)$ is computed as:

$$\mathbf{b}(\mathbf{p}) = \begin{cases} \frac{1}{e(n,m)}\sum\sum_{(k,l)\in\Theta_2} h(\mathbf{p},k,l)\mathbf{\Gamma} & \text{if } e(n,m)\neq 0 \\ 0 & \text{if } e(n,m) = 0 \end{cases} \qquad (5)$$

where Γ is a discrete vector with components (-k,-l) and:

$$e(n,m) = \sum\sum_{(k,l)\in\Theta_2} h(\mathbf{p},k,l) \qquad (6)$$

Both the GoG and the **b** operators can be used to localize a point of a discontinuity by starting from a point p of an approximate contour. However, there are significant differences between the two localization procedures. The GoG magnitude provides a function which enhances the points of the gray level discontinuities with local maxima. Therefore, given a starting point p, the nearest contour point can be located by dragging point p to the nearest local maximum of the function. In other words we need to compute the magnitude of the GoG operator along a path which joins point p to the nearest local maximum of the function. The search direction can be computed either by exploiting a priori knowledge of the shape of the object we are looking for or the image data itself. In the first case, the search direction is generally selected perpendicular to the approximate starting contour. In the second case, it can be obtained by computing the gradient of the GoG magnitude at the points of the starting contour.

When computed at a point p close to a discontinuity, the **b** operator provides a vector. This vector locates a new point p' which is nearer to the discontinuity than the starting point p. Therefore, the nearest point of the discontinuity can be located by iteratively computing the mass center of the gray level variability. When any new iteration occurs, the starting point is the mass center which is determined by means of the previous iteration. The procedure converges fast and few iterations are sufficient to reach the discontinuity.

To summarize, when the GoG operator is used, the filtering process is performed at all the points of a search path. On the contrary, the **b**

operator allows us to "jump" to the discontinuity in just a few steps.

However, no matter which operator is used, we can assume that the time necessary to locate a point of a discontinuity is approximately equal to the time necessary to perform the filtering process at a single point *p*, multiplied by the number of steps necessary to reach the discontinuity. In the next section the time necessary to execute the filtering process at a single point is considered.

It is worth noting that a few papers, which deal with the problem of obtaining real time performances, adopt a one dimensional (1D) edge detection operator [4,5]. This approach drastically reduces the computational requirements of the edge operator. However, the performance and the robustness to the noise also decrease so for this reason, in this paper we adopted 2D edge detection operators.

3. Implementation

The algorithms were implemented on the new TMS320C64x DSP (Figure 1). The device belongs to the second generation of fixed point DSPs of the high performance Texas Instruments DSP platform. The core of these new devices is based on an advanced Very Long Instruction World (VLIW) architecture, which has special features to speed-up image processing algorithms [6]. In particular, the new core can perform multiple operations with 8-bit packed data. This feature was widely exploited in our software implementation because our data are 8-bit gray scale values. Furthermore, the clock frequency of the new DSPs has increased significantly: a 600 MHz version of the devices is available today and a 1.1GHz version is planned for the near future.

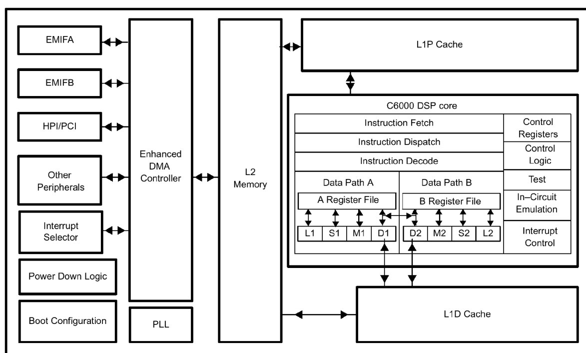
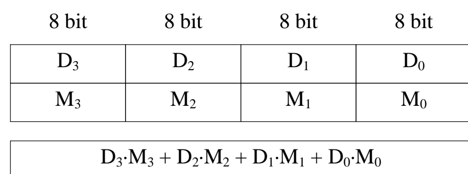


Figure 1: The C64x has a cache-based architecture, with a separate level-one program and data caches. The core has eight functional units (L1 S1 M1 D1 L2 S2 M2 D2) and can execute up to eight 32-bit instructions per cycle.

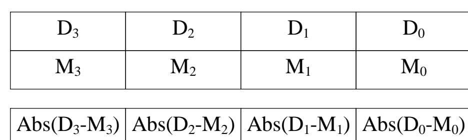
Both algorithms were implemented in a fully optimized hand–coded assembly. This solution was chosen to efficiently exploit the resources of the DSP and to achieve the best performances. The procedures were made C-callable so that they can be easily used in a program coded in C. The GoG algorithm requires two convolutions of the image data with the masks g_x and g_y . These convolutions are computed locally at the point *n,m* of the image. In other words, the convolution algorithm performs a point-by-point multiplication of the mask with the image data on the neighborhood Θ, and then sums the results. This operation is efficiently performed by the DOTPU4 instruction, which returns the dot-product between four sets of packed 8-bit values (figure 1-a). Two DOTPU4 instructions can be performed in parallel so that eight sets of packed 8-bit values can be multiplied and accumulated in a single cycle. Therefore, the convolution algorithm processes 8 pixels of the neighborhood Θ per cycle or, in other words, it requires 1/8=0.125 cycles per pixel (cpp). Consequently, the execution time of both the two convolutions of the GoG algorithm is about 0.25cpp which, in turn, gives rise to a total execution time equal to 0.25 cycles multiplied by the number of pixels of Θ. This result takes into account only the main loops of the code we implemented. A few further cycles are necessary to set up the loops and to compute the final square values and the sum. More exact results are shown in the next paragraph.



8 bit	8 bit	8 bit	8 bit
D_3	D_2	D_1	D_0
M_3	M_2	M_1	M_0

$D_3 \cdot M_3 + D_2 \cdot M_2 + D_1 \cdot M_1 + D_0 \cdot M_0$

(a)



D_3	D_2	D_1	D_0
M_3	M_2	M_1	M_0
$Abs(D_3-M_3)$	$Abs(D_2-M_2)$	$Abs(D_1-M_1)$	$Abs(D_0-M_0)$

(b)

Figure 2: (a) The DOTPU4 instruction returns the dot-product between four sets of values. The source operands D_i and M_i are 8-bit packed unsigned data. The destination operand is a 32-bit unsigned value. *(b)* The SUBABS4 instruction calculates the absolute value of the differences between 8-bit data. Both the source operands D_i and M_i and the destination operand are 8-bit packed unsigned data.

The computation of the **b** operator requires a rather more complex algorithm. First of all, a convolution algorithm is necessary to compute the value $f_l(n,m)$. Once this value is calculated, the absolute values of the differences between the data and f_l must be computed on Θ_2. This task was efficiently executed by the SUBABS4 instruction. This new instruction performs, in a single cycle, the absolute values of the differences of four sets of packed 8-bit data (figure 2-b). While these absolute values are computed at every point of Θ_2, the obtained values are also multiplied by the coefficients of the mask $g_2(k,l)$ to compute $e(n,m)$ and then by the coefficients -k and -l to compute the two Cartesian components of the vector $\boldsymbol{b}(n,m)$. To summarize, the **b** algorithm is split into two parts: (i) a convolution performed on Θ_1 and (ii) a summation performed on Θ_2. As we mentioned above, the convolution requires about 0.125cpp. As regards the summation, a software implementation was obtained in which the inner loop of the code is three instructions (i.e. three cycles) long. At each iteration of the loop, four pixels of Θ_2 are processed, so that 3/4=0.75 cycles per pixel are necessary to perform the summation on the neighborhood Θ_2. In conclusion, when $\Theta_1 = \Theta_2$, the **b** algorithm requires about:

0.125 cpp + 0.75 cpp = 0.875 cpp.

4. Results

In Table1, the timing of the two algorithms for sizes of the neighborhoods $\Theta=\Theta_1=\Theta_2$ varying from 13^2 to 49^2 is reported. As regards the c64x, the timing was performed using a simulator of the device (when this paper was written, silicon was not yet available). When evaluating the performances of an algorithm implementation, the results also depend on where the data are stored. In this paper, we assumed that all the data were stored in the level one (L1) cache memory, which is the memory accessed directly by the CPU. This condition provides the best results. On the contrary, if a value is not in the L1 cache (cache miss condition), then the code execution is suspended until the value is retrieved.

A comparison with an implementation of the two algorithms on a standard Personal Computer is also reported. The PC processor was an AMD Athlon 850MHz and the algorithms were coded in C. As far as the data caching is concerned, the function runs repetitively with the same data so that only the internal cache memory is used. Having carried out the testing in this way, the results obtained can be compared correctly to those obtained with the c64x.

	TMS320C64x 600MHz		**Athlon 850MHz**	
Size of Θ	GoG	**b**	GoG	**b**
13^2	0.19	0.31	7.3	15
25^2	0.38	0.88	26	52
37^2	0.69	1.8	57	113
49^2	1.1	3.1	98	198

Table 1: Timing of the two operators. All times are in µs.

Size of Θ	**TMS320C64x 600MHz**	**TMS320C80 60MHz**
13^2	0.31	8.1
25^2	0.88	16
37^2	1.8	29
49^2	3.1	46

Table 2: a comparison between the performances of the two DSP is reported. The timing refers to the execution of the **b** operator. All times are in µs.

As previously mentioned, the results of table 1 are obtained when all the data are in the L1 cache memory. However, if the code runs with new data, then some cache misses can occur and the execution time increases. To be able to account for this additional overhead is not easy without performing a complete simulation of the system, although an estimate can be reported. Let the image data be in the L2 internal SRAM and let the L1 cache memory be empty. This situation gives rise to a “cache miss” in all the first accesses to the data and can be considered the worst case. However, no matter which operator is used, the first access to the data occurs in a convolution algorithm. When using the GoG the first access to the data occurs in the first of the two convolutions with the masks g_x and g_y while when using the **b** operator the first access to the data occurs in the convolution necessary to compute $f_l(n,m)$. In this sort of algorithm, which performs multiple memory accesses, the cache misses are pipelined and the code execution is suspended for two cycles per memory access [7]. Therefore, the time necessary to execute the first convolution triples and, consequently, it will require 0.375 cpp. This loss of performances involves an overhead of up to 100% in the GoG based algorithm, and an

overhead of up to 29% in the **b** operator based algorithm.

In the final, table 2 reports a comparison between the performances of the TMS320C64x and of the TMS320C80, which we used in a previous paper. The timing of the c80 does not include the time necessary to move the data to the inner high-speed memory. This additional overhead was about 100% in the real-time applications we implemented.

5. Conclusions

In this paper, we presented the software implementation of two edge detection operators (the GoG and the **b** operator) on a state of the art DSP device. The former is one of the operators most widely used to locate a discontinuity in contour tracking applications. The latter is a new operator which we are using in our real time contour tracking system.

The timing of the two algorithms shows that the computation is about two-three times more demanding when the **b** operator is computed.

However, in order to make the right choice, when focusing on a specific application we must consider the problem of implementing the entire algorithm. When using the GoG operator, as well as when using the **b** operator, the localization procedure is iterative. That is, the *maximum* number of steps necessary to reach the discontinuity is another important parameter of the procedure. This parameter depends on the mathematical operator chosen and can vary a lot. For example, tests on synthetic and clinical images revealed that three steps of the **b** operator are enough to locate the discontinuity even when the starting contour is far from the final contour. On the contrary, when using the GoG the maximum number of steps strictly depends on the length of the search path which can be rather long. Therefore, when focusing on a specific application this parameter must be carefully considered.

As far as the architecture of the C64x is concerned, special features, which improve the implementation of the algorithms, were highlighted. The comparison with the C80 revealed the remarkably better performances of the new device. This improvement was due to both the new architecture and the higher clock frequency. Moreover, the implementation work was much simpler on the c64x.

Ultimately, the results we obtained show that the use of the new DSP is a good answer to the problem of real time contour tracking. In fact, the procedures are so onerous that the computational resources of a standard PC are insufficient to obtain real time performances and special hardware devices are required. On the other hand, these algorithms are quite complex and they cannot be easily implemented on programmable logic (FPGA, CPLD) or on custom devices (ASIC). Therefore, a software solution, such as the use of a DSP device, is a good compromise as regards flexibility and computational power. Moreover, the flexibility of a software solution is important for another reason. In fact, the contour tracking procedures depend on the sort of object under examination. In particular, information on the shape and on the movement of the object can be added to the contour estimation algorithms to regularize and, consequently, to increase the robustness of the procedure. Therefore, if we want to use the same system to deal with several contour tracking problems, then we need to implement more than one procedure. A software-based system remarkably simplifies this work.

References:

[1] J. Canny, A computational approach to edge detection, *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-8, 1986, pp. 679-698.

[2] V. Torre and T. Poggio, On edge detection, *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-8, 1986, pp. 147-163.

[3] V. Gemignani, M. Demi, M. Paterni, A. Benassi, *Real-Time Implementation of a New Contour Tracking Procedure in a Multi-Processor DSP System.* Proc. of CSCC2000 pp. 3521-3526.

[4] Z.Li, H.Wang, Real time 3D Motion Tracking with Known Geometric Models, *Real-Time Imaging*, 1999, **5**, pp. 167-187.

[5] M.Vincze, M. Ayromlou, W. Kubinger, Improving the Robustness of Image-based Tracking to Control 3D Robot Motions, *Proc. ICIAP99, Venice, Italy,* 1999, pp.274-279.

[6] Texas Instruments, *TMS320C6000 CPU and Instruction Set Reference Guide*, SPRU189F

[7] Texas Instruments, *TMS320C6000 Peripherals Reference Guide*, SPRU190D