REAL TIME EXECUTION OF OPTIMAL EDGE DETECTORS ON RISC AND DSP PROCESSORS

Lionel LACASSAGNE[1,2], Frantz LOHIER[1,2], Patrick GARDA[1]

[1]Laboratoire des Instruments et Systèmes
Université Pierre et Marie Curie
4 place Jussieu - B.C. 252
75252 Paris Cedex 05, FRANCE

lionel | lohier | garda@lis.jussieu.fr

[2]Electronique Informatique Applications
Bâtiment 4, Burospace
route de Gisy
91571 Bièvres Cedex, France

eia@wanadoo.fr

ABSTRACT

This paper presents the real time implementations of the Canny-Deriche optimal edge detectors on RISC and DSP processors. For each type of architecture, the most leading optimization techniques are described. A comparison is then made between RISC and DSP processing speeds.

1. INTRODUCTION

Canny-Deriche operators have asserted themselves to the edge detection field which stands as a fundamental component of image processing and computer vision.

The main drawback is the prohibitive computational power they require. It has led people to design dedicated hardware implementations (FPGA, ASIC) to achieve the real time execution of these detectors. On the other hand, the still increasing performance of RISC and DSP calls into question the need for a dedicated architecture.

This paper shows that crafty software implementation of Canny-Deriche edge detectors may achieve real time execution on state of the art RISC and DSP processors.

2. DERICHE'S OPERATORS

2.1 Reminder of Canny-Deriche optimal filters

Canny's approach [1] consists in finding the optimal FIR filter which satisfies the three following constraints for a Heaviside input signal: good detection, good localization, low maximum multiplicity due to the noise.

Deriche [3], using Canny's approach, has looked for an IIR filter which satisfies the same constraints. He got the same differential equation, but while changing the conditions at the limits, he obtained, for the Canny's performance index, an improvement of 25%.

Deriche's operators are used for two important methods of edge detection. The first is based on the gradient maxima, the second, on the laplacian's zero crossings. The state of the art methods combine both of them. In this paper we will focus on the gradient method. The implemented operators are those proposed by Deriche in [3].

2.2 Deriche's gradient

The directional derivative is the result of a smoothing in one direction followed by a derivation in the other.

The smoothing operator equation is:

$$y_1(n) = k\left[x(n) + e^{-a}(a-1)x(n-1)\right] + 2e^{-a}y_1(n-1) - e^{-2a}y_1(n-2)$$

$$y_2(n) = k\left[e^{-a}(a-1)x(n+1) - e^{-a}(a-1)x(n+2)\right] + 2e^{-a}y_2(n+1) - e^{-2a}y_2(n+2)$$

$$y(n) = y_1(n) + y_2(n),\ k = \frac{(1-e^{-a})^2}{1+2ae^{-a}-e^{-2a}}$$

The derivative operator is:

$$y_1(n) = -kx(n-1) + 2e^{-a}y_1(n-1) - e^{-2a}y_1(n-2)$$

$$y_2(n) = kx(n+1) + 2e^{-a}y_2(n+1) - e^{-2a}y_2(n+2)$$

$$y(n) = y_1(n) + y_2(n),\ k = (1-e^{-a})^2$$

3. OPTIMIZATIONS

In this section, we introduce algorithmic and architectural optimization techniques which lead to real-time execution on RISC processors. The algorithm is coded in C and all computations are done in floating point. The keys to speed up the process are as follows: a well-fed pipeline and ALUs, a few memory access, an efficient cache use, and the storage of intermediary results in the registers.

3.1 Algorithmic optimization

The first optimization is to modify the equation of the filter in order to reduce its complexity. The major breakthrough was achieved by Garcia-Lorca in [4]. He introduced a new decomposition of the optimal IIR edge detector with a lower computational burden. FGL provides the same Canny's performance index than Deriche. Deriche's smoother is replaced by two passes of a first order smoother.

Causal smoothing filter : $y(n) = (1-g)x(n) + gy(n-1)$

Anti-causal smoothing filter : $y(n) = (1-g)x(n+1) + gy(n+1)$

Deriche's derivative filter is computed by the convolution of the following 2×2 kernels, followed by the L1 magnitude:

$$\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} \text{ and } \begin{bmatrix} -1 & -1 \\ +1 & +1 \end{bmatrix}$$

We can even get a lower complexity by replacing the two passes of the FGL's smoother by a single pass of a 2nd order filter which is the square of the first order filter. The following table shows that the Garcia-Lorca's operator complexity is half that of Deriche.

Deriche		FGL 1st order		FGL 2nd order	
MUL	ADD	MUL	ADD	MUL	ADD
26	24	16	14	12	14

Array 1: filter complexity

3.2 Memory access optimization

When the image is vertically swept, two neighboring pixels in the same column are not stored side by side in the memory. As this can cause read cache faults, the vertical smoothing is replaced by a horizontal filter applied to the transposed image. As Deriche and Garcia-Lorca filters are recursive, the storage of the output in a register, as the input of the next iteration, decreases the transfers from the RAM to the cache.

3.3 Processing optimization

Loop-unrolling is a very efficient technique that ensures a better use of the pipeline [5]. Instead of running the loop n times , the loop body is duplicated k time, and the new loop body is executed n/k time. Loop unrolling has to be performed by hand as state of the art optimizers lake to do so automatically for recursive equation.

The following example shows the pseudo-code for Garcia-Lorca's 2nd order filter. The loop is unrolled 3 times with regard to the filter formula. At the beginning of the loop y_k contains the value Y[i-k].

$$y(n) = b_0 x(n) + a_1 y(n-1) + a_2 y(n-2) \qquad (1)$$

$x_0 \leftarrow X[i]$, $x_1 \leftarrow X[i+1]$, $x_2 \leftarrow X[i+2]$

$y_0 \leftarrow b_0 x_0 + a_1 y_1 + a_2 y_2$, $y_2 \leftarrow b_0 x_1 + a_1 y_0 + a_2 y_1$,

$y_1 \leftarrow b_0 x_2 + a_1 y_2 + a_2 y_0$

$Y[i] \leftarrow y_0$, $Y[i+1] \leftarrow y_1$, $Y[i+2] \leftarrow y_2$

4. BENCHMARKING

4.1 Operational scheme

We benchmarked four RISC processor families for image sizes varying from 128 to 640. Each measure was done 10 times, then linear regression was performed to get more reliable results. Two versions of each filter (Deriche and Garcia-Lorca) were implemented, the first without optimization, the second with all the optimizations described in the previous section. The processor characteristics are summarized in the table below[1]:

Processors	Frequency (MHz)	Cache L1 (KB)	Cache L2 / L3 (KB)	RAM (MB)
PA 8000	180	1+1 MB	0 / 0	64
Ultra Sparc 1	143	16+16	512 / 0	64
Pentium MMX	200	16+16	512 / 0	48
Alpha 21164	500	8+8	96 / 8 MB	128

Table 2**: processors characteristics**

4.2 Results

The table gives the time in milliseconds for computing the gradient of image sizes 128, 256 and 512. The last two columns show the maximum image size for which the processing is performed in real time.

	128		256		512		Real Time	
processors	Deriche	FGL	Deriche	FGL	Deriche	FGL	Deriche	FGL
PA 8000	25.9	10.8	114.4	43.8	690.5	301.8	158	246
Ultra Sparc1	20.1	11.0	94.5	47.4	562.1	253.0	172	234
P 200 MMX	29.1	11.9	149.8	63.5	660.6	259.9	144	216
Alpha 500	10.5	4.7	46.2	13.3	262.2	71.7	248	392

Table 3**: RISCs' results**

The graphs below show the running time for Deriche and Garcia-Lorca algorithms (optimized and non optimized) as a function of the image size.

[1] For the Pentium, MMX instructions are not used in this implementation, but we took advantage of the cache size, which is bigger than for MMX Pentium

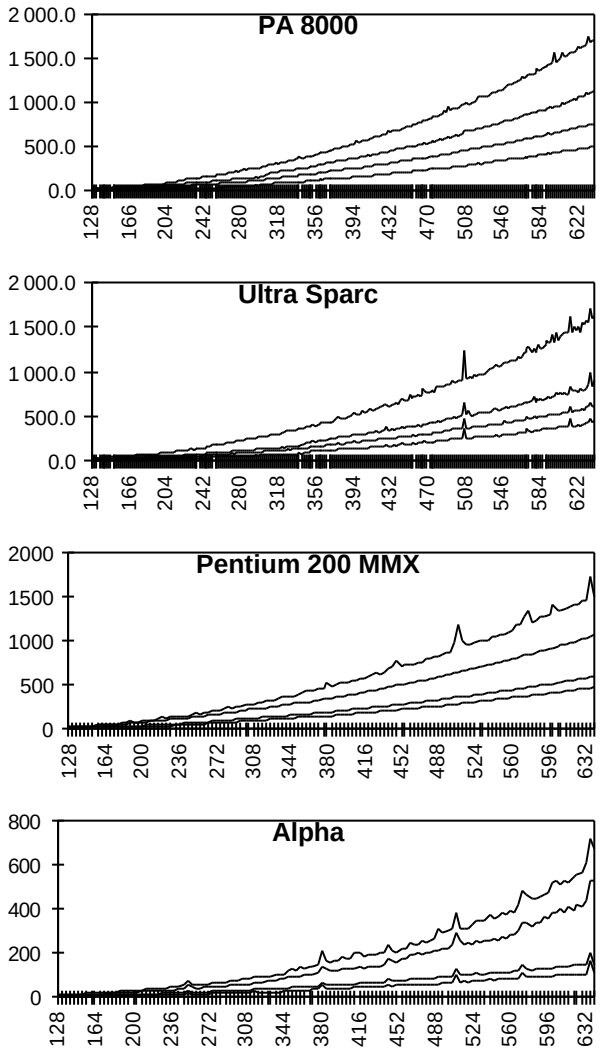

Figure 1**: RISC**

4.3 Results analysis

The graphs look alike. The gain between non and optimized version is greater for Deriche (50%) than for Garcia-Lorca. This may be due to the quality of the compiler: as Garcia-Lorca's filter is very simple, it is more optimized by the compiler. But the running time ratio between Deriche and Garcia-Lorca filters is in the range 2-3.

The PA8000, Ultra1 and Pentium are very close, but the Alpha is by far the fastest: from 2-3 to 2-5 times respectively for Deriche and Garcia-Lorca filters. The cache faults appear on all processors and for the same image sizes. For those critical sizes, the cache fault number is connected to the cache size (especially the L1 cache).

5. DSP IMPLEMENTATION

In the previous section, we have shown that real-time performance is now conceivable for reasonable sizes on RISC processors. We now introduce 2 DSP architectures that lead to even better performance using assembly language.

5.1 SPMD processing on the C80 VLIW advanced DSPs

TI's C80 is capable of **2,5 Gops** at 60 Mhz thanks to a RISC floating point processor (the Master Processor, MP) and to 4 advanced 32 bits fixed point VLIWs (or Parallel Processor, PP), all gathered in a single chip. Processors access internal memory (cache or general purpose static memory) through a crossbar thus inducing minimum contention. An advanced DMA controller performs all processors data transfers between off-ship and on-chip memory where processing takes place.

5.1.1 VLIW instructions

4 sets of operations can be done in parallel in a single cycle of the 3 levels instruction pipeline. PP's 64 bits VLIW instruction can use the multiply hardware, the general ALU hardware and 2 memory load/store address units:

- ***PP's multiply hardware***: among many possible combinations, 2 cycles are necessary for 2 rounded and truncated multiplies between 2 16 bits integers and 2 fixed point constants.
- ***PP's ALU***: it's extremely powerful and explains the C80's overall calculation performance. It can combine algebraic and arithmetic operations in a single cycle. Its general equation is of the form $A \& f(B,C) \pm g(B,C)$ where A, B and C correspond to the raw ALU input ports (& stands for bitwise logical AND). B can result from a register left shift ($B \circ reg \backslash\backslash amount$) and C can be used to generate a mask of a specified number of bits ($C \circ 2^n - 1$). f and g summarize independent boolean functions.
- ***PP's address units***: 2 independent powerful address units can access data without contention in a direct or indexed way. Moreover arithmetic (+/-) is allowed on any pointer register before or after the memory access.

5.1.2 Hardware loop controllers

To avoid handling a loop counter and the associated compare and branch instructions, each PP features a hardware mechanism to cope with up to 3 nested loops. Thus, loop unrolling techniques aren't necessary to gain performance.

5.1.3 Edge detection on the PPs

- Complexity for first order FGL IIR filter:

PP's registers file is "multi-ported". Registers can be used both as operand and destination of operations. Registers are modified at the end of all the parallel operations. The following pseudo code gives the initialization phase and the **2 cycles** kernel loop (';' stands for sequential, '||' for parallel):

Init. : $x \leftarrow X[0] \parallel \tau \leftarrow 0 ; \theta \leftarrow x \cdot b_0 \parallel x \leftarrow X[1]$
1st cycle : $\theta \leftarrow x \cdot b_0 \parallel y \leftarrow \theta + \tau \parallel x \leftarrow X[i \leftarrow i+1]$
2nd cycle: $\tau \leftarrow y \cdot a_1 \parallel Y[j \leftarrow j+1] \leftarrow y$
(cf equation 1, $a_2=0$, τ and θ are intermediate results)

- Gradient magnitude computation:

To achieve **11 cycles for 4 pixels** per PP, we use the ALU SIMD processing ability. The general ALU's equation can operate on a 32 bits register seen as 4 separate bytes. A so called 4 bits multiple status flag (*mf[1…4]*) is maintained for each split operation. This flag can be used to merge 4 masks in a 32 bits register which in turn can be used to perform 4 selections between bytes from two 32 bits registers. This technique allows to calculate 4 absolute values in just 2 cycles.

Filling the VLIW slots as possible and taking advantage of PP's ALU SIMD feature, we obtain the following estimations:

	Per PP		Tot. Cycles
	Cycles/pixel	Num. pass	
EGL 1st order	2	8	4×512^2
Gradient	11/4	1	$11/16 \times 512^2$
Total cycles			1228800
Raw duration at 60 Mhz (without transfer)			**20 ms**

Table 4**: C80 raw estimations for a 512^2 image**

5.1.4 DMA data transfer during calculation

While processing, the DMA brings the completed data back to external memory and downloads new data to be processed. The offset of the next peace of external buffer's data is automatically handled by the DMA. During transfer, no contention occurs with processing as internal memory banks are carefully toggled. On the other hand, DMA requests contention occur as the DMA is a shared resource among processors.

5.2 VLIW processing on the C6x advanced DSP

VLIW processors appear as an alternative to the MIMD parallel processors such as the C80. TI's C62 is a VLIW DSP capable of 1.6 GIPS at 200 Mhz. This performance is results of :

- a 12 levels VLIW instruction pipeline with 5ns cycle
- 256 bits VLIW instruction coding 8 operations which can partially be executed sequentially or concurrently[1]
- 2 16x16 multiplies, 1 shift, 2 adds/subtracts, 1 branch, 2 loads/stores (with pointer modification) can be done in parallel thanks to 8 independent functional units
- Several DMAs leading to about 250 Mbytes/s peak.

5.2.1 Optimization techniques

Software pipelining is the main technique used to obtain performance from the C6x. Due to the different latencies of operations in the execution level of the pipeline (2 cycles for multiply, 5 for a load operation, 6 for a branch and 1 cycle for any other operation), we maximize the use of the different units by executing the largest number of parallel operations for the loop body keeping in mind that they do not concern the same processing iteration. We then provide a prolog and epilog piece of code to guarantee that operations get properly time stamped in accordance with the body. In addition, the C6x features 2 split 16 bits add/subtracts which individually allow 2 parallel additions of 2 16 bits numbers, all in a single cycle (using 2 functional units only). Applying those techniques yields to the following estimations:

	Cycles/pixel	Num. pass	Tot. cycles
EGL 1st order	1/1	8	8×512^2
EGL 2nd order	2/1	4	8×512^2
Gradient	4/2	1	2×512^2
Total cycles for EGL 1 or EGL 2 and Gradient			2621440
Raw duration at 200 Mhz (without transfer)			**13 ms**

Table 5**: C62 raw estimation for a 512^2 image**

5.2.2 DMA transfer during calculation

As with the C80, parallel DMA transfer is possible without contention, but C6x's DMA offers less features and requires more processor intervention when downloading 2D array's of data in several blocks. Its throughput is also lower than the 480 Mbytes/s C80's DMA peak performance.

5.3 Comparing both DSP architectures

As far as the raw estimations are concerned, both DSP run almost as fast for the gradient calculation, but the C6x is faster on the IIR filter thanks to its high operation frequency. Including the cost of memory transfers would determine which of those 2 architectures is best suited for low level image processing.

6. CONCLUSION

In the section 4, we have shown through benchmarks that C software implementation on floating point RISC processors achieve real time execution of optimal filters for image sizes up to 384^2 on DEC Alpha processors and up to about 256^2 on others. In the section 5, we have shown through estimations, that software implementation in assembly language on advanced fixed point DSPs achieve real time execution for image size larger than 512^2.

[1] Texas calls this the VelociTI™ technology.

More specifically, 512^2 images may be processed at 50 frames/s by the C80 at 60 Mhz, and 75 by the C62 at 200 Mhz. However that does not consider the impact of memory transfer. In the worst case, this may double the processing time, leading to a 25 frames/s processing of 512^2 Images.

Those results outperform state of the art implementation of optimal edge detectors on DSPs and FPGAs [5].

7. REFERENCES

[1] J.F. Canny. *A computational Approach to Edge Detection*, IEEE Trans on PAMI, vol 8,6 p79-698 (1986).

[2] R. Deriche. Fast Algorithms for low level-vision, IEEE Trans. on PAMI, vol 12,1 (1990).

[3] F. Garcia-Lorca. *Filtres recursifs temps reel pour la detection de contours : optimisations algorithmiques et architecturales*. These Universite d'Orsay (1996).

[4] J.L. Hennessy, A. Patterson. *Computer Architecture. A quantitative approach*. 2nd edition Morgan Kaufmann Publishers (1996).

[5] Y. Kim, kim@ee.washingtom.edu. Whasington University http://icsl.ee.washington.edu/projects/iclib.