Implementation on DSP of an active contour algorithm for endocardium tracking in echocardiographic images

Juan F. Zapata and Ramón J. Ruiz

Depto. Electrónica, Tecnología de Computadoras y Proyectos
Universidad Politécnica de Cartagena
Muralla del Mar s/n, 30202, Cartagena, Spain.

ABSTRACT

In first place, in this paper, the basic process of parallel code implementation is discussed for a VLIW architecture. Parallel code modules allow the implementation of a contour active (snake) for segmentation and tracking of endocardium in echocardiographic images. In second place, this work discusses the performance obtained through this design model. In this case, it is necessary to check performances in order to obtain a qualitative and quantitative measurement of our implementation. We have chosen one example which permits to understand the methodology used in order to obtain the maximum performance of hardware features of VLIW processor: the distance between points of active contour, very used in different modules composing the active contour algorithm.

Keywords: VLIW Architecture, DSP Processor, Active Contours

1. INTRODUCTION

DSP is a key enabling technology for many types of electronic products.[1] DSP-intensive tasks are the performance bottleneck in many computer applications today. Computational demands of DSP-intensive tasks are increasing very rapidly. In many embedded DSP applications, general-purpose microprocessors are not competitive with DSP-oriented processors today. Digital signal processor will become increasingly important in an expanding range of electronic products over the next several years, much as microprocessors and microcontrollers have over the past two or three decades

Operations developed for image processing are computationally intensives, mainly due to the tremendous volume of optical information and the complex mathematical operations required. This aspect of computational demand becomes decisive when a real-time result is required. General purpose processors are not appropriate for real-time medical processing imaging applications because of technical tradeoffs in hardware design. General purpose processors are extremely capable when it comes to data manipulation but are inefficients for mathematical calculation. DSP (digital signal processing) processors work in an opposite way since they are efficient for mathematical calculation and less efficients in data manipulation.[2] Digital signal processing is one of the most powerful technologies and has shown enormous growth and had an important technological impact in several areas, e.g. in telecommunications, medical imaging, radar and sonar, high fidelity music reproduction, to name just a few. DSP systems are perfectly capable of solving the problem of endocardium motion tracking in echocardiography images.

A digital signal processor accepts one or more discrete-time inputs, $x_i[n]$, and produces one or more outputs, $y_i[n]$, for $n = \ldots, -1, 0, 1, 2, \ldots$, and, $i = 1, \ldots, N$.[3] The inputs could represent appropriately sampled (analog-to-digital conversion) values of continuous time signals of interest, which are then processed in the discrete time domain, to produce outputs in discrete-time that could then be converted to continuous time, if necessary.

DSP systems are often characterised by the algorithms used. The algorithm specifies the arithmetic operations to be performed but does not specify how that arithmetic is to be implemented. It might be implemented in software on an ordinary microprocessor or programmable signal processor, or it might be implemented in custom

Further author information: (Send correspondence to Juan F. Zapata)
Juan F. Zapata: E-mail: juan.zapata@upct.es, Telephone: 34 968 32 64 58

integrated circuits. The selection of an implementation technology is determined in part by the required speed and arithmetic precision.

Numerous papers have considered the problem of automatic contour tracking.[4–6] One approach is to use active contour models to track the boundaries of anatomic structures in medical images.[7] This technique is a suitable alternative to classic edge-based segmentation techniques, and combines the detection of grey level transitions within the image together with the need to obtain a closed contour. This task involves both hardware and software aspects. In fact, while purely hardware solutions have proved to be inefficient and unsuitable for complex algorithms, a purely software solution based on a PC does not offer a large bandwidth in terms of MOPS and system throughput.[8] Therefore, an integrated hardware/software platform based on a DSP processor located in a host PC was chosen as environment to develop active contour algorithms. Due to its full programmability, high performance and novel architecture, the TMS320C6701 is one of the most suitable devices for researching and testing new digital signal processing algorithms.

2. HARDWARE OVERVIEW: TMS320C60

2.1. TMS320C60 EVM evaluation board

The choice of DSP-based hardware permits the development of a powerful and highly flexible system. Due to its full programmability and high performance, the TMS320C6701 processor mounted on the TMS320C60 EVM is a suitable device for researching and testing DSP algorithms.[9] In the work described in this paper, a personal computer was equipped with the Code Composer Studio development environment which helps to construct and debug embedded real-time DSP applications. It provides tools for configuring, building, debugging, tracing and analysing programs. Texas Instruments DSP's provide on-chip emulation support that enables Code Composer Studio to control program execution and monitor real-time program activity. Communication with this on-chip emulation support occurs via an enhanced JTAG link. This link is a low-intrusion way of connecting into any DSP system. Emulator interface provides the host side of the JTAG connection. The heart of the TMS320C60 EVM evaluation board is the Texas Instruments TMS320C6701 processor. The C6701 is based on a VLIW-like architecture which allows it to execute up to eight RISC-like instructions per clock cycle.[10] It is capable of executing all TMS320C62xx instructions, and has added support for floating-point arithmetic and 64-bit data. It uses an 1.8-volt core supply (with 3.3-volt I/O), and executes up to 334 million MACs per second at 167 MHz.

The two data paths of the C6701 support for 64-bit data and IEEE-754 32-bit single-precision and 64-bit double-precision floating-point arithmetic. Each data path includes a set of four execution units, a general-purpose register file, and paths for moving data between memory and registers. The four execution units in each data path comprise two ALUs, a multiplier and an adder/subtractor which is used for address generation. The ALUs support both integer and floating multiplies. The two register files each contain sixteen 32-bit general-purpose registers. To support 64-bit floating point arithmetic, pairs of adjacent registers can be used to hold 64-bit data. The C6701 offers support for floating-point reciprocal and reciprocal square root estimation, and for converting data between fixed- and floating-point formats.

The on-chip memory system of the TMS320C67xx implements a modified Harvard architecture, providing separate address spaces for program and data memory. Program memory has a 32-bit address bus and a 256-bit data bus. Each of the two data paths is connected to data memory by a 32-bit address bus and two 32-bit data buses. Since there are two 32-bit data buses for each data path, the TMS320C67xx can load two 64-bit words per instruction cycle. The only member of the TMS320C67xx family that is currently available, the TMS320C6701, has 64 Kbytes of 32-bit on-chip program RAM and 64 Kbytes of 16-bit on-chip data RAM. The maximum sustainable on-chip data memory bandwidth for a 167 MHz TMS320C6701 is 334 64-bit Mwords or 668 32-bit Mwords per second.

The TMS320C6701 has one external memory interface, which provides a 23-bit address bus and a 32-bit data bus. These buses are multiplexed between program and data memory accesses. The peak external memory bandwidth for a 167 MHz TMS320C6701 is 167 32-bit Mwords per second. In typical applications, however, the external memory bandwidth is expected to be significantly lower because peak bandwidth requires very fast SRAM, which is not practical in many applications.

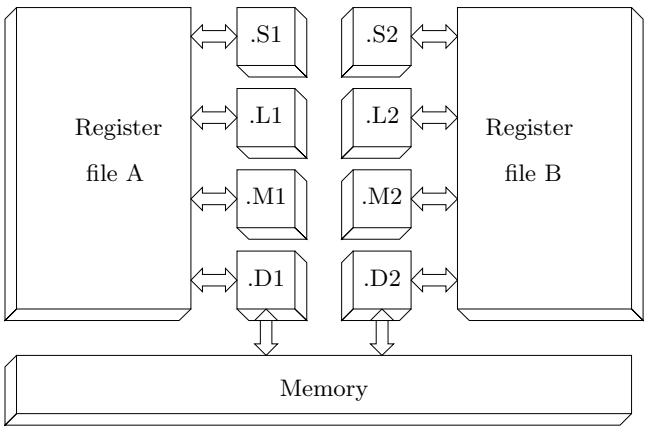

Figure 1. TMS320C67xx's core

Addressing modes supported by the TMS320C67xx include register-direct, register-indirect, indexed register-indirect, and modulo addressing. Immediate data is also supported. The TMS320C67xx does not support modulo addressing for 64-bit data.

The TMS320C67xx does not support hardware looping, and hence all loops must be implemented in software. However, the parallel architecture of the processor allows the implementation of software loops with virtually no overhead.

The peripherals on the TMS320C6701 include a host port, four-channel DMA controller, two TDM-capable buffered serial ports and two 32-bit timers.

2.2. Pipeline performance and memory access considerations

The pipeline is most effective when it is kept as full as the algorithms in the program allow it to be. It is useful to consider some situations that can affect pipeline performance. A fetch packet (FP) is a grouping of eight instructions. Each FP can be split into from one to eight execute packets (EPs). Each EP contains instructions that execute in parallel. Each instruction executes in an independent functional unit. In our case six units. The effect on the pipeline of combinations of EPs that include varying numbers of parallel instructions, or just a single instruction that executes serially with other code, was considered in our implementation. In general, the number of execute packets in a single FP defines the flow of instructions through the pipeline. Another defining factor is the instruction types in the EP. Each type of instruction has a fixed number of execute cycles that determines when this instruction's operations are complete.

Finally, the effect of the memory system on the operation of the pipeline was considered too. The TMS320C6701 has a memory configuration typical of a DSP, with program memory in one physical space and data memory in another physical space. Data loads and program fetches have the same operation in the pipeline, they just use different phases to complete their operations. With both data loads and program fetches, memory accesses are broken into multiple phases. This enables the TMS320C6701 to access memory at a high speed.

3. IMAGE ANALYSIS MODULES IMPLEMENTATION

In this section, in first place, we pointed the basic process of parallel code development on a VLSI architecture. Traditional development flows in the DSP industry have involved validating a C model for correctness on a host PC or Unix workstation and then painstakingly porting that C code to hand coded DSP assembly language. This can be both time consuming and error prone. This process tends to encounter difficulties that can arise from maintaining the code over several projects.

New emergent development tools have advantages to allow the compiler to do all the laborious work of instruction selection, parallelising, pipelining, and register allocation. This allows the programmer to focus

on getting the desired result. These tools simplify the maintenance of the code, as everything resides in a C framework that is simple to maintain, support, and upgrade. The recommended code development flow for the TMS360C6xx involves the phases described below.

1. Compile and profile native C code. Validates original C code. Determines which loops are most important in terms of MIPS requirements
2. Add restrict qualifier, loop iteration count, memory bank, and data alignment information. Optimise C code using *intrinsics* and other methods of optimisation (e.g. data size).
3. Write assembly code

In second place, in this section, we compare the performance obtained using this development flow. This allows to obtain a qualitative and quantitative measure of goodness of our implementation. In this sense, it is need to use some example which permits to understand the methodology used in order to obtain the maximum performance of the features hardware of our processor. Although we could have elected the scalar product, we have elected the distance between *snaxels*, a image module very used in different algorithms of active contours. So that, we introduce the active contour algorithm to implement to next briefly.

3.1. Active contours

Geometrically, a snake or deformable active contour is a spline with a parametric representation $\mathbf{v}(s) = (x(s), y(s))$ embedded in the image plane $(x, y \in R)$, where x e y are the coordinate function and $s \in [0, 1]$ is the parametric domain.[11]

Energetically, the active contour is subordinated to two kinds of energy that dictate the behaviour of the deformable model. The equation (1) can be viewed as a representation of the energy of the contour and the final shape corresponds to the minimum of this energy.

$$E^*_{snake} = \int_0^1 E_{int}(\mathbf{v}(s)) + E_{ext}(\mathbf{v}(s)) \tag{1}$$

The first term of the functional, is the internal deformation energy. It characterises the deformation of a stretchy, flexible contour. Two parametric functions, $\omega_1(s)$ and $\omega_2(s)$, dictate the simulated physical characteristics of tension and rigidity of the contour, respectively. In this sense, an important problem in computer vision is to determine weights for the optimisation of a multiple function. There are techniques for the optimisation of both weights.[12] An approach more simple is to use a heuristic value of tuning for each kind of image. An increment of $\omega_1(s)$ tend to eliminate laces and loops. An increment of $\omega_2(s)$ increase the smoothness and reduce the flexibility.

$$\int_0^1 E_{int}(\mathbf{v}(s))\ \mathrm{d}s = \int_0^1 \omega_1(s) \left| \frac{\partial \mathbf{v}(s)}{\partial s} \right| + \omega_2(s) \left| \frac{\partial^2 \mathbf{v}(s)}{\partial s} \right|^2 \ ds \tag{2}$$

The second term, the image energy, couples the snake to the image. traditionally,

$$\int_0^1 E_{ext}(\mathbf{v}(s))\ \mathrm{d}s = \int_0^1 P(\mathbf{v}(s)) + E_{con}(\mathbf{v}(s))\ \mathrm{d}s \tag{3}$$

where $P(\mathbf{v}(s))$ denotes a scalar potential function defined on the image plane.

In the original formulation of Kass et al,[11] the internal energy term E_{int} is defined as the first and second derivative of the contour, giving to the active contour certain characteristics of tension and rigidity, respectively. Discretising the equation (2) for a point of the contour, the term E_{int} can be approximated by finite differences:

$$E_{int}(\mathbf{v}_i) = ||\mathbf{v}_i - \mathbf{v}_{i-1}||^2 + ||\mathbf{v}_{i-1} - 2\mathbf{v}_i + \mathbf{v}_{i+1}||^2 \tag{4}$$

where parameters, ω_1 and ω_2 can be adjust to the unity so that no energy rule over the other.

It is important that all the energy terms that contribute to the total energy can match up and be compared. That is why all the types of energy that role get normalised. The internal energy, in concrete, with the middle distance between *snaxels* $l(\mathbf{v})$:

$$l(\mathbf{v}) = \frac{1}{n}\sum_{i=1}^{n} ||\mathbf{v}_i - \mathbf{v}_{i-1}||^2 \tag{5}$$

which permits the internal energy to be not variable to changes of scale and rotations.

Design of external potentials, external energy term, and its extraction of the image allows to exploit theirs spatial continuity and that the contour interacts with its surround in a dynamic way. This active and autonomous process can be controllated through potentials with ability of imposing constraints. Cohen[13] thought in the inclusion of a force to oblige to the active contour to expand and to contract in absence of other type of forces, and in this manner, to remove solves some of the problems encountered in the original model. Cohen formulates the balloon model in the next form:

$$F = k_1\vec{n}(s) - k\frac{\nabla P}{||\nabla P||} \tag{6}$$

where $\vec{n}(s)$, is a normal unitary vector to the curve at point $\mathbf{v}(s)$, k_1, is the force amplitude and k a scale factor and P, is the image potential.

The change of sign of k_1 or the orientation of the curve have an effect of deflation instead of inflation in the curve. k_1 and k are chosen such than they are of the same order so an edge point can stop the inflation force. The discretised balloon energy term proposed is below.

$$e_{jk}(\mathbf{v}_i) = -[\vec{n_i} \bullet (\vec{\mathbf{v}_i} - \vec{p_{jk}}(\mathbf{v}_i))] \tag{7}$$

where $\vec{n_i}$ is a unitary normal vector to the curve $\mathbf{V}$ at point $\mathbf{v}_i$. $p_{jk}(\mathbf{v}_i)$ is the point in the image which is related spatially to the element $e_{jk}(\mathbf{v}_i)$ in the neighbourhood matrix of energy.

Alternatively to the balloon force, before defined, other methods have been proposed to resolve the problem of initialisation which was resolved by placing the active contour near of the interest features of desired contour. Among this approaches, we can highlight multiresolution methods[6] or utilisation of attraction potentials.[14] Xu and Prince designed a new external force called *gradient vector flow* force or GVF force.[15,16] The GVF force expects to solve the mentioned problems before. The gradient vector flow field is a vectorial field which is derived of the image through the minimisation of a energy functional. This external GVF force is computed as a diffusion of the gradient vectors of a gray-level or binary edge map derived from the image. A snaxel $\mathbf{v}_i$, under influence GVF force $f(\mathbf{v}_i)$ lonely, it is moved in the direction of this GVF force. Each term of energy for such an element, in the neighbourhood matrix of GVF energy can be expressed as a scalar product as follows:

$$e_{jk}(\mathbf{v}_i) = -[\vec{f}(\mathbf{v}_i) \bullet (\vec{\mathbf{v}_i} - \vec{p_{jk}}(\mathbf{v}_i))] \tag{8}$$

where $f(\mathbf{v}_i)$ is the GVF force which is related to the spacial location of $\mathbf{v}_i$ and $p_{jk}(\mathbf{v}_i)$ is the pixel on the image which is related to $e_{jk}(\mathbf{v}_i)$ of neighbourhood matrix of energy.

3.2. Implementation

Although all the code running on the DSP processor can be written in C because the C compiler generates a quasi-efficient code, the performance of the application can be maximised by using compiler options, intrinsic instructions, and overcoat assembly code transformation. The computational cost of the active contour algorithm constitutes most the computational cost of whole procedure. Therefore, the algorithm is implemented in assembly language on the four execution units in each data path and executed in parallel in order to utilise all the hardware resources of DSP and exploit all the capabilities of the architecture VLIW and software pipelining.

Because most of the millions of instructions per second (MIPS) in DSP applications occur in tight loops, it is important for the application to make maximal use of all the hardware resources in important loops. Fortunely, loops inherently have more parallelism than non-looping code because there are multiple iterations of the same code executing with limited dependencies between each iteration.

To maximise the efficiency of the code, the application schedules as many instruction as possible in parallel. In order to schedule instructions in parallel, the relationships, or dependencies, between instruction was determined, as it is showed in Figure 2 . Dependency means that one instruction must occur before another, e.g. a variable must be loaded from memory before it can be used. Only independent instructions can execute in parallel, dependencies inhibit parallelism.

Listing 1 shows a part of assembly code for snaxel distance and allocated resources. The following rules affect the assignment of functional units: load (LDH, LDW and STW) instructions must use a .D unit, multiply (MPY and MPYSP) instructions must use a .M unit. add (ADD and ADDSP) instructions use a .L unit, subtract (SUB) instructions use a .S unit. and branch (B) instructions must use a .S unit. The ADD and SUB can be on the .S, .L, or .D units.

The symbol || define which instruction are executing in parallel. The first parallel instruction have not this symbol. The symbol @ define which iteration of the loop the instruction is executing each cycle. For example, the rightmost column shows that on any given cycle inside the loop, the ADDSP instruction are adding data for iteration n, the MPYSP instruction are multiplying data for iteration n+2 (@@), The SHL instruction are moving bits for iteration n+3 (@@@), and so on.

No value can be live in a register for more than the number of cycles in the loop. Otherwise, iteration n+1 write into register before iteration n has read that register. Therefore, a value is written to a register at the end of cycle, then all children of that value must be read the register before the end of the new cycle. The live-too-long problem means that no loop variable can live longer than the iteration interval, because a child would then read the parent value for the next iteration. To solve this problem, the parent can be moved to a later cycle but this is not always a valid solution. Another solution is to break up the lifetime of a variable by inserting move (MV) instructions. The MV instruction breaks up the variables path in smaller pieces and can be live for minimum iteration interval.

If-then-else statements in C cause certain instructions to execute when the if condition is true and other instructions to execute when it is false. One way to accomplish this in assembly code is with conditional instructions on one of five general-purpose register. Conditional instructions can handle both the true and false cases of if-then-else C statement. Branching is one way to execute the if-then-else statement: branch to the ADD instruction when the if statement is true and branch to the SUB when the if statement is false. However, because each branch has five delay slots, this method requires additional cycles. Furthermore, branching within the loop makes software pipelining almost impossible. Using conditional instructions, on the other hand, eliminates the need to branch to the appropriate piece of code after checking whether the condition is true or false. For this, instructions can be used conditionally on the zero and nonzero values of a condition register. This method also allows to software pipeline the loop and achieve much better performance than with branching.

Software pipelining is a technique used to schedule instruction from a loop so that multiple iterations execute in parallel. The parallel resources on the 'C6x make it possible to initiate a new loop iteration before previous iteration finish. The goal of software pipelining is start a new loop iteration as soon as possible. The modulo iteration interval scheduling table is used as an aid to creating software pipeline loops. Table 1 provides a method to keep track the available resources on a cycle-by-cycle basis to ensure that no resource is used twice on any given cycle and shows how a software-pipelined loop executes. In this case, different instructions use the same functional unit. For example, the LDHs and MVs use the .D units. The MPY and MPH are scheduled on the .M unit. Asterisks define which iteration of the loop the instruction is executing each cycle. Delay slots show in Figure 2 are shown in Table 1.

4. RESULTS AND CONCLUSIONS

The distance between snaxels can be implemented using the sequential assembly code in Listing 2 (assembly code that has not been register-allocated and is unscheduled). A parallel assembly code is proposed in Listing 3, and in Figure 2 is shown its dependency graph. Table 1 shows the module iteration interval scheduling with software pipeline. This software pipeline allows to obtain a better assembly code because the pipeline and its resources are better used. This better assembly code is shown in Listing 1.

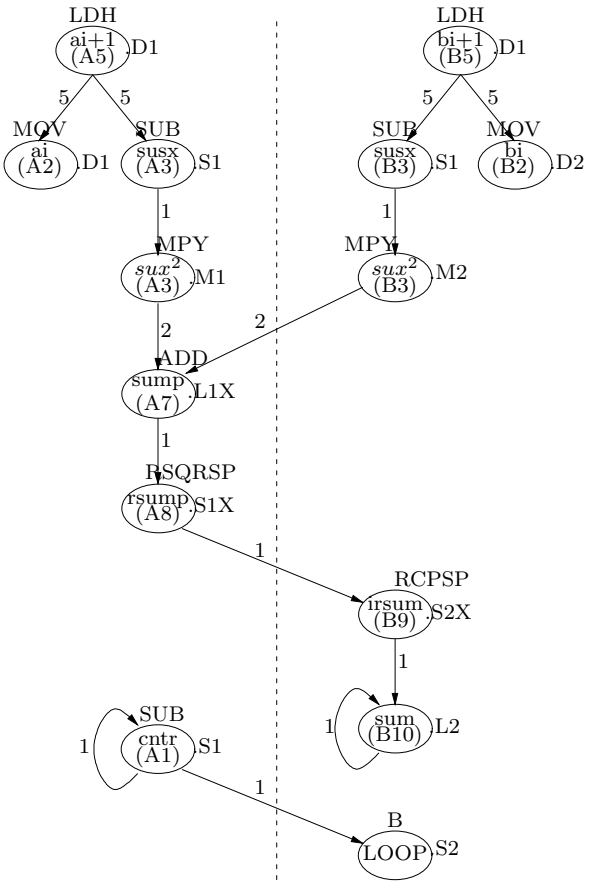

Figure 2. Dependency graph of distance between snaxels.

U./c	0,13...	1,14...	2,15...	3,16...	4,17...	5,18...	6,19...	7,20...	8,21...	9,22...	10,23...	11,24...	12,25...
.D1	LDH	*LDH	**LDH	***LDH					MV	*MV	**MV	***MV	
.D2	LDH	*LDH	**LDH	***LDH					MV	*MV	**MV	***MV	
.M1							MPY	*MPY	**MPY	***MPY			
.M2							MPY	*MPY	**MPY	***MPY			
.L1						SUB	*SUB	**SUB	***SUB	ADD	*ADD	**ADD	***ADD
.L2						SUB	*SUB	**SUB	***SUB				ADD
.S1		SUB	*SUB	**SUB	***SUB						RSQ	*RSQ	**RSQ
.S2								B	*B	**B	***B	RCP	*RCP

Table 1. Module Iteration interval scheduling for distance between snaxels

```
        LDH     .D1     *A4++,A2        ; preload ai
||      LDH     .D2     *B4++,B2        ; preload bi
||      MVK     .S1     cont,A1         ; setup counter
||      ZERO    .L2     B10             ; zero acumulator

  [A1]  SUB     .S1     A1,1,A1         ; decrement counter
||      LDW     .D1     *A4++,A2        ;* load ai & ai+1  from memory
||      LDW     .D2     *B4++,B2        ;* load bi & bi+1 from memory

  [A1]  SUB     .S1     A1,1,A1         ;* decrement counter
||      LDW     .D1     *A4++,A2        ;** load ai & ai+1 from memory
||      LDW     .D2     *B4++,B2        ;** load bi & bi+1 from memory

  [A1]  SUB     .S1     A1,1,A1         ;** decrement counter
||      LDW     .D1     *A4++,A2        ;*** load ai & ai+1 from memory
||      LDW     .D2     *B4++,B2        ;*** load bi & bi+1 from memory

  [A1]  SUB     .S1     A1,1,A1         ;*** decrement counter

        SUB     .L1     A5,A2,A3        ;ai + 1 - ai (x)
||      SUB     .L2     B5,B2,B3        ;bi + 1 - bi (y)

        SUB     .L1     A5,A2,A3        ;*ai + 1 - ai (x)
||      SUB     .L2     B5,B2,B3        ;*bi + 1 - bi (y)
||      MPY     .M1     A3,A3,A6        ;(ai + 1 - ai)*(ai + 1 - ai) (x)
||      MPY     .M2     B3,B3,B6        ;(bi + 1 - bi)*(bi + 1 - bi) (y)

        SUB     .L1     A5,A2,A3        ;**ai + 1 - ai (x)
||      SUB     .L2     B5,B2,B3        ;**bi + 1 - bi (y)
||      MPY     .M1     A3,A3,A6        ;*(ai + 1 - ai)*(ai + 1 - ai) (x)
||      MPY     .M2     B3,B3,B6        ;*(bi + 1 - bi)*(bi + 1 - bi) (y)
||[A1!] B       .S2     LOOP

        SUB     .L1     A5,A2,A3        ;***ai + 1 - ai (x)
||      SUB     .L2     B5,B2,B3        ;***bi + 1 - bi (y)
||      MPY     .M1     A3,A3,A6        ;**(ai + 1 - ai)*(ai + 1 - ai) (x)
||      MPY     .M2     B3,B3,B6        ;**(bi + 1 - bi)*(bi + 1 - bi) (y)
||      MV      .D1     A5,A2           ;preload a1
||      MV      .D2     B5,B2           ;preload b1
||[A1!] B       .S2     LOOP            ;*
```

Listing 1. Distance between snaxels parallel assembly (software pipeline)

Both parallel assembly code shown a better performance as is shown in Table 2. It is important to highlight that the functional units share different instructions, the LDHs and MVs use the .D units and The MPY and MPH are scheduled on the .M unit. This implies a well-planed pipeline.

It is an important fact to highlight that the parallel assembly code uses a few cycles NOP in comparison with nonparallel assembly code. This is due to a better utilisation of resources of the pipeline, which is better that for a sequential assembly code.

Into the loop kernel for parallel assembly code with software pipeline, we can observe the EP (execution packet). Each EP has at least two parallel instructions although the majority has a high number of parallel instructions (there is a EP with seven instructions in parallel). This high parallelisation permits a high performance of pipeline.

Table 2 shows the performance of nonparallel, parallel and parallel with software pipeline code. It is evident that the code can have a better performance in a 86%. This confirms that a development in parallel assembly code on a digital signal processor allows a better performance of an algorithm because the resources of system are better used. In this sense, module iteration interval scheduling table allows a better management of this resources in time and physically in each functional unit.

Summing up, this paper describes a method that help us development more efficient assembly code, understand the code produced and perform manual optimisation once we have developed and optimised our C code using a compiler, extract inefficient areas from our code and rewrite them in linear assembly. A active contour method was implemented using this method and, under this circumstances, the implementation can be considered like very satisfactory and a good result for the DSP processor adopted.

```
; ai into A2

        LDH     .D1     A4++,A5         ;load ai + 1 from memory (x)
        LDH     .D2     B4++,B5         ;load bi + 1 from memory (y)
        NOP     4
        SUB     .S1     A5,A2,A3        ;ai + 1 - ai (x)
        SUB     .S2     B5,B2,B3        ;bi + 1 - bi (y)
        MPY     .M1     A3,A3,A6        ;(ai + 1 - ai)*(ai + 1 - ai) (x)
        MPY     .M2     B3,B3,B6        ;(bi + 1 - bi)*(bi + 1 - bi) (y)
        NOP
        ADD     .S1X    A6,B6,A7        ;(ai + 1 - ai)^2 + (bi + 1 - bi)^2
        RSQRSP  .S1     A7,A8           ;root inverse
        RCPSP   .S1     A8,A9           ;inverse
        ADD     .L1     A9,A10,A10      ;sum+=sum
        SUB     .S1     A1,1,A1         ;decrement counter
        MV      .D1     A5,A2           ;store ai + 1 into A2
        MV      .D2     B5,B2           ;store bi + 1 into B2

  [A1!] B       .S2     LOOP
        NOP     5
```

Listing 2. Distance between snaxels nonparallel assembly

ACKNOWLEDGMENTS

This work has been supported by Fundación Séneca of Región de Murcia and Ministerio de Ciencia y Tecnología of Spain, under grants PB/63/FS/02 and TIC2003-09400-C04-02, respectively.

```
;load ai into A2

        MVK     .S1     100,A1          ;setup counter
||      ZERO    .L2     B10             ;zero
||      LDH     .D1     A4++,A2         ;preload ai from memory (x)
||      LDH     .D2     B4++,B2         ;preload bi from memory (y)
LOOP:
        LDH     .D1     A4++,A5         ;load ai + 1 from memory (x)
||      LDH     .D2     B4++,B5         ;load bi + 1 from memory (y)
        SUB     .S1     A1,1,A1         ;decrement counter
        NOP     3
        SUB     .L1     A5,A2,A3        ;ai + 1 - ai (x)
||      SUB     .L2     B5,B2,B3        ;bi + 1 - bi (y)
        MPY     .M1     A3,A3,A6        ;(ai + 1 - ai)*(ai + 1 - ai) (x)
||      MPY     .M2     B3,B3,B6        ;(bi + 1 - bi)*(bi + 1 - bi) (y)
  [A1!] B       .S2     LOOP
        MV      .D1     A5,A2           ;store ai + 1 into A2
||      MV      .D2     B5,B2           ;store bi + 1 into B2
        ADD     .L1X    A6,B6,A7        ;(ai + 1 - ai)^2 + (bi + 1 - bi)^2
        RSQRSP  .S1     A7,A8           ;Root inverse
        RCPSP   .S2X    A8,B9           ;inverse
        ADD     .L2     B9,B10,B10      ;sum+=sum
; jump starts here
```

Listing 3. Distance between snaxels parallel assembly

Code	100 iterations	cycles
Distance between *snaxels* nonparallel assembly	$2 + 100 \times 24$	2402
Distance between *snaxels* parallel assembly	$1 + (100 \times 13)$	1301
Distance between *snaxels* parallel assembly (*software pipeline*)	$1300/4 + 1$	326

Table 2. Comparison of assembly code for distance between snaxels

REFERENCES

1. J. Eyre and J. Bier, "The Evolution of DSP Processor," *IEEE Signal Processing Magazine* , pp. 43–51, Mar. 2000.
2. P. Lapsley, J. Bier, A. Shoham, and E. Lee, *DSP Processor Fundamentals: Architectures and Features*, IEEE Press, 1996.
3. V. Madisetti, *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*, IEEE Press, 1995.
4. C. Davatzikos and J. L. Prince, "An Active Contour Model for Mapping the Cortex," *IEEE Trans. on Medical Imaging* **14**, pp. 65–80, Mar. 1995.
5. C. Davatzikos and J. L. Prince, "Convexity Analysis of Active Contour Models," in *Proc. Conf. on Info. Sci. and Sys.*, pp. 581–587, 1994.
6. B. Leroy, I. Herlin, and L. D. Cohen, "Multi-resolution Algortihms for Active Contour Model," in *12th International Conference on Analysis and Optimization of System*, pp. 58–65, 1996.
7. J. Zapata and R. Ruiz, "Tracking endocardial boundaries in echocardiographic sequences by an active contour," in *First Online Symposium for Electronics Engineers*, 2000.
8. V. Geminami, S. Provvedi, M. Demi, M. Paterni, and A. Benassi, "A DSP-Based Real Time Contour Tracking System," *IEEE* , 1999.
9. "TMS320C62x/C67x Programmer Guide," Texas Intruments, May 1999. SPRU198C.
10. "TMS320C6701 Digital Signal Processor Data Sheet," Texas Intruments, May 1999. SPRS067.
11. M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active Contour Models," *International Journal of Computer Vision* **1**, pp. 321–331, Jan. 1988.
12. M. Gennert and A. Yuille, "Determining the Optical Weights in Multiple Objetive Function Optimization," in *Proceeding ICCV*, pp. 87–89, Tampa, Fl, 1988.
13. L. Cohen, "On Active Contour Models and Balloons," *Computer Vision, Graphics and Image Proccessing* **53**, pp. 211–218, Mar. 1991.
14. L. Cohen and I. Cohen, "Finite-Element Methods for Active Contour Models and Balloons for 2-D and 3-D Images," *IEEE Trans. Pattern Analysis and Machine Intelligence* **15**, pp. 1131–1147, Nov. 1993.
15. C. Xu and J. Prince, "Gradient Vector Flow: A New External Force for Snakes," in *IEEE Proc. Conf. on Comp. Vis. Patt. Recog. (CVPR)*, pp. 66–71, 1997.
16. C. Xu and J. Prince, "Snakes, Shapes, and Gradient Vector Flow," *IEEE Transactions on Image Processing* , pp. 359–369, March 1998.