Facial Motion Estimation with the TMS320C62xx Family

Francisco Barat Quesada, Patricia M. Burgess*, Rudy Lauwereins
K.U.Leuven, Belgium **Texas Instruments Inc.*
Francisco.Barat@esat.kuleuven.ac.be, p-burgess@ti.com, Rudy.Lauwereins@esat.kuleuven.ac.be

Abstract

Very low bitrate coding is an important part of the MPEG-4 standard. To obtain maximum compression in videoconference sequences a technique called semantic coding can be used. With this technique, the face of the speaker is represented by a model, and its movement between frames is coded with a small set of parameters. The quality of the motion description grows with the number of parameters used. As with all MPEG standards, the encoder algorithm is not described. This leaves the door open to many possible implementations.

In this paper we describe an algorithm used to calculate the facial motion parameters and the implementation of this algorithm using the current members of the TMS320C62xx family, which can be the starting point for a "Simple Facial Animation Visual Profile" encoder. The algorithm is based on the optical flow constraint equation and a set of equations that describe the movement of the face. The estimation is performed in two parts: first the movement of the head as a rigid object is estimated and then, the facial movements are calculated.

The implementation of this algorithm is then discussed for the following C62xx family members: C6201, C6201B, C6202, C6203 and C6211. The implemented algorithms make full use of the VLIW processing power of the processors. The main difference between the implementations is the use of memory and DMA; they all maintain the same processing kernels.

1 Introduction

Many different methods have been used in video coding throughout the years. The goal behind this evolution is data compression. Hybrid coding is today's choice for most applications. Good examples of this fact are the MPEG-1 [1], MPEG-2 [2], and H.263 [3] standards.

Many of today's distributed applications need to transmit video information through greatly limited bandwidth channels. In such applications however, e.g. videophone systems, these techniques don't offer the required quality level.

In these applications, the coding standards mentioned above suffer from artifacts that greatly reduce the picture quality. There is a need for very low bitrates, and high quality video sequences. Usually, the images in this kind of applications depict a known type of object (e.g. in videophony, a talking head). Since the images captured nearly always portrait the known object, there is a lot of useful information in the picture itself that can be used to improve coding efficiency.

Model-based coding (also known as semantic coding) is a technique that uses this information to further compress the sequence. Model-based coding of faces has been recently internationally standardized for the first time in the MPEG-4 standard [4]. With the facial animation tool of MPEG-4, the movement of a face can be coded with as little as 2kbps.

In model-based coding, the real world in front of the camera is represented by a 3D model and the movement of the model is represented by a set of motion parameters. The use of these parameters is the key to higher compression ratios.

In the case of facial coding these parameters can be divided in two groups: global and local motion parameters. The global motion parameters treat the face as a rigid object and represent its movement as a combination of translation and rotation. The local motion parameters represent localized movement of parts of the face, such as opening the mouth, turning the eyes, and raising the eyebrows.

The SNHC (Synthetic and Natural Hybrid Coding) group of MPEG-4 has standardized a tool, called facial animation tool, [5], that is the basis for semantic coding. In this tool, local motion parameters are called Facial Animation Parameters (FAPs). There are 68 standardized FAPs. 66 are low-level FAPs that correspond to simple movements of the face, and 2 are high-level FAPs that correspond to expression and viseme animation. A small sample of them can be seen in Table 1.

Simple FAPs can be divided in two categories: translation based and rotation based. The difference between them is based on the kind of movement they represent. Translation based FAPs specify the translation of part of the face (e.g. raise_l_ear), while rotation based FAPs specify the rotation of part of the face (e.g. yaw_r_eyeball). In our simplified approach, we only work with translation based FAPs. This limits the

available set of FAPs but permits the use of simpler equations.

Table 1 Sample FAPs

FAP name	FAP description
open_jaw	Vertical jaw displacement (does not affect mouth opening)
raise_r_m_eyebrow	Vertical displacement of right middle eyebrow
raise_l_ear	Vertical displacement of left ear
yaw_r_eyeball	Horizontal orientation of right eyeball
stretch_l_cornerlip	Horizontal displacement of left inner lip corner

As with all MPEG standards, the facial animation tool encoder algorithm is not described. This gives way to many possible implementations. In this paper we present an algorithm used to estimate both the global and the local motion parameters. The algorithm is based on [6], [7], [8] and [9].

Facial motion estimation algorithms are extremely computation-hungry. For this reason, a powerful architecture is needed for their implementation. The C6x family, [10], provides such an architecture. With its double datapath VLIW architecture, the C6x promises enough performance for the implementation of the presented algorithm. We will focus on the currently anounced fixed-point members of the family, namely the C6201, C6201B, C6202, C6203 and C6211. All these members share the same processing core but have a different configuration of memory and peripherals. These differences will lead to implementation differences.

The paper is organized as follows. In section 2 we describe the algorithm, without discussing its implementation on the C6x family. In section 3 we study an implementation of this algorithm in the already mentioned members of the C62xx family. In section 4 we present some preliminary results of the implementation of the algorithm on these processors and, finally, in section 5 we derive some conclusions.

2 Description of the algorithm

In this section we describe the algorithm used to estimate the movement. The algorithm is based on the traditional optical flow equation.

The estimation system needs a 3D mesh of the face and the corresponding texture to create the synthesized image by affine deformation and texture mapping. The mesh is made up of triangles. The model used in our experiments (known as the Candide model) can be seen in Figure 1. The process through which this model is obtained is not discussed here. This step is very important for a successful implementation of this kind of coders. There is a great amount of work being done in this field.

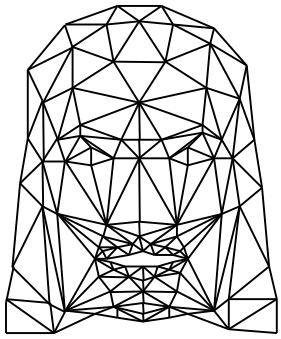

Figure 1 The Candide model

The estimation of the head movement between two consecutive frames is done using a gradient-based approach based on the work of on [6], [7], [8] and [9]. The basic idea is to synthesize an image similar to the one recently captured from the camera and, by analyzing the differences between them, find a new set of parameters. This new set of parameters should move the model to a position as close as possible to the position of the face in the captured image. This technique receives the name of analysis by synthesis.

The movement of a point k of the 3D model can be described by the following equation:

$$\mathbf{p_k}(t+\Delta t) = \mathbf{R}\cdot\left(\vec{\mathbf{p}}_\mathbf{k}(t) + \sum_{n=0}^{N}\Delta a_n \cdot \vec{\mathbf{V}}_{\mathbf{k,n}}\right) + \vec{\mathbf{T}}$$

Where p_k is a point of the model, R is the rotation matrix, T is the translation vector, $V_{k,n}$ are the deformation vectors associated with each of the FAPs we are working with, and a_n are the intensities of each of the FAPs. In this equation we have only represented translation based FAPs. As we mentioned before, the other kind of simple FAPs, rotation based, are left out from this model. The introduction of them in the above equation would complicate the system. Furthermore, these rotation based FAPs (that represent eye movement) are better obtained with other methods. High-level FAPs can be obtained from the set of low-level FAPs. This procedure is not discussed here.

Movement is estimated in two phases. First, the global movement is estimated as if there was no local movement involved. Then, local movement is estimated as if the global movement had been precisely solved, as in [7]. This approach allows us to make certain simplifications on the calculations without introducing a noticeable amount of error. Once the global parameters have been found, the local motion parameters are estimated between the input image and a synthesized image using the recently found global parameters. In this second step, there is no global

movement because it has already been accounted for in the previous step.

We will now focus on the global motion calculation. If we assume that local movement is negligible and that global movement between two frames is very small, we can represent the velocities of each point as:

$$\begin{pmatrix} v_{k,x} \\ v_{k,y} \\ v_{k,z} \end{pmatrix} \cong \begin{pmatrix} 0 & -R_z & R_y \\ R_z & 0 & -R_x \\ -R_y & R_x & 0 \end{pmatrix} \cdot \begin{pmatrix} p_{k,x}(t) \\ p_{k,y}(t) \\ p_{k,z}(t) \end{pmatrix} + \begin{pmatrix} T_x \\ T_y \\ T_z \end{pmatrix}$$

The traditional optical flow equation relates changes in image brightness intensity with velocity vectors:

$$I_{k,x} \cdot v'_{k,x} + I_{k,y} \cdot v'_{k,y} + I_{k,t} = 0$$

Assuming orthogonal projection along the z axis, we can arrive to the following equation by combining the previous equations:

$$\begin{pmatrix} -\mathrm{I}_{k,y} \cdot p_{k,z}(t) \\ \mathrm{I}_{k,x} \cdot p_{k,z}(t) \\ -\mathrm{I}_{k,x} \cdot p_{k,y}(t) + \mathrm{I}_{k,y} \cdot p_{k,x}(t) \\ \mathrm{I}_{k,x} \\ \mathrm{I}_{k,y} \end{pmatrix}^T \cdot \begin{pmatrix} R_x \\ R_y \\ R_z \\ T_x \\ T_y \end{pmatrix} = I_k(t) - I_k(t + \Delta t)$$

The unknowns in this equation are R_x, R_y, R_z, T_x and T_y. (T_z is no longer present because a movement along the z axis produces no change in an image formed by orthogonal projection.) The other coefficients can be calculated for every point of the model. I_x, I_y and I_t represent the partial derivatives that can be obtained from the image data. The z value can be interpolated from the z value on the vertices of each triangle. With this, we obtain an over-determined equation system:

$$\mathbf{H} \cdot \mathbf{U} = \mathbf{Y}$$

Where H is a matrix of coefficients size (n,5), U is the unknown matrix, size (5,1) and Y is another matrix of coefficients, size (n,1). (n is the number of pixels processed). We can solve this system by a least mean squares method (LMS):

$$\mathbf{U} = \left(\mathbf{H}^{\mathbf{T}} \cdot \mathbf{H}\right)^{-1} \cdot \mathbf{H}^{\mathbf{T}} \cdot \mathbf{Y}$$

Once the global movement is calculated, a new synthesized image using this recently obtained parameters is created. The differences between this synthesized image and the input image are due only to local movement (if the global motion estimation process worked within a certain margin of error). In this case, the velocities of each point k can be written as:

$$\vec{v}_k = \sum_{n=0}^{N} \Delta a_n \cdot \vec{\mathbf{V}}_{\mathbf{k,n}}$$

Again, by use of the optical flow equation we get the following equation, similar to the global motion estimation one:

$$I_{k,x} \cdot \sum_{n=0}^{N} \Delta a_n \cdot V_{k,n,x} + I_{k,y} \cdot \sum_{n=0}^{N} \Delta a_n \cdot V_{k,n,y} + I_k(t + \Delta t) - I_k(t) = 0$$

The unknowns are the intensities of each FAP, a_i. The values of $V_{k,n}$ of each pixel can be interpolated from the $V_{k,n}$ values at each vertex of the triangles. The rest can be directly computed from the images. Using a similar approach to global motion, we can find the unknowns (A):

$$\mathbf{A} = \left(\mathbf{F}^{\mathbf{T}} \cdot \mathbf{F}\right)^{-1} \cdot \mathbf{F}^{\mathbf{T}} \cdot \mathbf{I}$$

Due to non-linearities in the system, the above procedures do not provide an exact solution. To obtain a better one an iterative process is done. This process is represented in Figure 2.

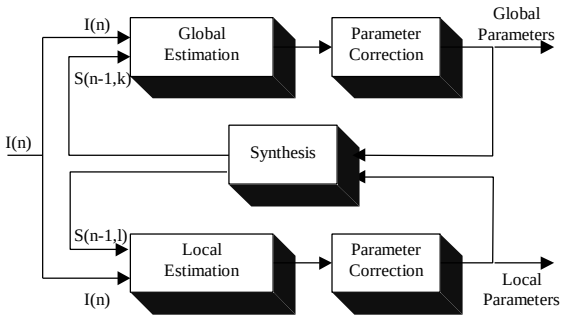

Figure 2 Analysis through synthesis loop

In a first step, global motion is estimated. The control flows around Synthesis, Global Estimation, and Parameter Correction for a number of times before doing the local motion estimation loop. The minimum number of iterations depends on the noise of the images, the number of pixels used in the computations, the movement between the images and many other factors. The number of iterations can be fixed in the program (providing a known execution time) or can depend on the values estimated (with an unknown execution time). In case that the number of iterations is determined at runtime, a useful technique for deciding when to stop iterating is when the movement parameters change less than a certain amount.

After calculating global movement, the local estimation loop is executed. It includes Synthesis, Local Estimation, and Parameter Correction. Again, the number of iterations can be fixed or decided at runtime.

3 Implementation description

The above algorithm has been implemented for the following members of the C62xx family: C6201, C6201B, C6202, C6203 and C6211.

All members of the C62xx family share the same processing core. The main difference between them is the configuration of the internal memory and the integrated peripherals. This will lead to different algorithms for memory management and external memory accesses. For this reason, we will first discuss the implementation of the computation part of the algorithm, which is common to all

devices. We will then focus on the implementation of the memory management on the different members of the family.

3.1 Processing

The motion estimation process returns a set of parameters that describe the movement that has to be applied to the model in order to synthesize a face as similar to the one in the input image. This process is applied to each input image. The parameters obtained are combined in an additive manner to finally obtain a set of absolute positioning parameters.

The main steps for one iteration of the global motion estimation process presented above, are as follows:

1. Synthesize an image with the calculated parameters of the previous input image.
2. Calculate the H and Y matrices.
3. Solve the over-determined system of equations and obtain U.

Step 1 involves creating a synthesized image in a memory buffer, be it internal or external to the chip. Step 2 accesses this buffer and the captured image buffer. This can be seen in Figure 3. A major improvement in the execution time can be obtained if we note that the synthesized image is only used in the calculation of the H and Y matrices. If we mix steps 1 and 2 into a more complex one, we can reduce the memory requirements and the memory accesses. This will lead to a better performance. Thus, steps 1 and 2 are merged into a step that combines the synthesis and the analysis (this concept is different to the analysis by synthesis mentioned before). This is a kind of loop merging. Every time a pixel is synthesized, its contribution to the H and Y matrices is computed. The pixel value is not used anymore. Memory accesses are transformed into faster register transfers.

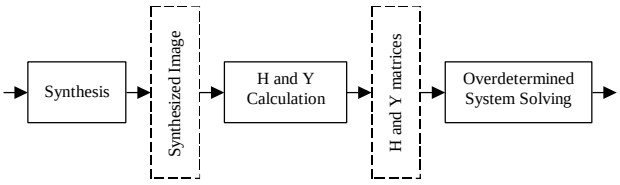

Figure 3 Non optimized estimation iteration

The size of the H and Y matrices is proportional to the number of pixels processed, which can be quite large. These matrices are multiplied by H^T in step 3 in the LMS solving step seen before. It is possible to calculate the contribution of each pixel to H^TH and H^TY during step 2. This reduces the number of memory accesses and the memory requirements, as can be seen in Figure 4. Thus, the main processing algorithm is transformed into the following:

1. Calculate H^TH and H^TY values during synthesis and analysis.
2. Solve the regular equation system.

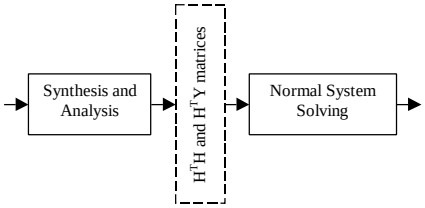

Figure 4 Optimized estimation iteration

Pixel processing is done by grouping pixels into triangles. All the pixels inside a triangle are processed in a sequential manner: from left to right and from top to bottom. This leads to an improvement in performance by reuse of values. The following is the pseudocode for step 1 above:

```
for (each triangle)
{
    Load triangle data.
    for (each line in a triangle)
        for (each pixel in a line)
        {
            Calculate synthesized pixel.
            Calculate partial derivatives.
            Calculate coefficients for H
            Calculate HtH contribution.
       }
    }
}
```

To achieve the highest processing velocity, all the data for one triangle is first stored into internal memory. This is represented by "Load triangle data" in the pseudocode. The ideal situation would be having the texture buffer and the captured image both in internal memory. This is not possible in most of the C62xx devices due to the size of internal memory. The solution to this problem varies from one device to another. This will be discussed in the following subsection.

The local motion estimation process is very similar; the main differences are in the calculations of the coefficients of the equation. In this case, for each triangle we have a set of FAPs that affect it. The ones that do not affect the triangle have $V_{k,n}$ value of 0. Instead of calculating all the coefficients from the local motion estimation equation, we can limit the calculation to those non-zero coefficients. A different loop is used depending on the number of FAPs that affect the triangle. Each loop is an optimized version of the more general one. This approach can be seen in the code below.

```
local(int nFAP)
{
    switch (nFAP)
    {
        case 0: break;
        case 1: local_1();  break;
        case 2: local_2();  break;
        case 3: local_3();  break;
        ...
        default: local_n(nFAP);
    }
}
```

With this technique, an enormous reduction in cycle count is obtained. This reduction will be discussed in a following section. The major drawback of this approach is the increase in code size. If there is not enough memory to store all the needed versions, the most common numbers of FAPs are the ones to be coded in an optimized way.

3.2 Memory management

We will now discuss those aspects dependent on the different devices. The size and number of internal data RAM blocks of the different devices can be seen in Table 2 (each memory block is divided in four memory banks for concurrent access). As we can see, there is a great difference between the different devices. We treat them separately.

Table 2 Memory configurations

	Program Memory Size	Data Memory Size	Data Memory Configuration
C6201	512Kb	512Kb	1 block
C6201B	512Kb	512Kb	2 blocks
C6202	2Mb	1Mb	2 blocks
C6203	3Mb	4Mb	2 blocks
C6211	32Kb + 512Kb shared	32Kb + 512Kb shared	2 level cache or memory blocks

We make the assumption that external memory is able to provide data at a burst rate of one memory access per cycle for DMA transfers.

Data memory is used to store the data structures of the model and the image data for calculations. The 3D model data needed for the calculations should be stored in internal memory to obtain the highest speed. The amount of memory needed depends on the number of vertices, triangles and FAPs of the model. For the model used in our system, the amount of memory needed is less than 10KB.

The C6201, [12], has one internal memory block of 64Kbytes divided in four 16 bits wide banks. The size of the images being processed will determine the memory management algorithm. This is also true for the other devices.

In case that we are working with QCIF images (352x288, 25344 bytes), there is enough memory to store both the texture image and the captured image in the internal memory. Every time that an image is processed, the next captured picture should be stored in internal memory. This is done using a DMA transfer. During this transfer, the CPU is stalled waiting for it to finish. A better approach is loading half of the image and start processing it when it is loaded. While the upper half is being processed, the loading of the lower part can proceed.

The CPU and the DMA controller will collide sometimes while accessing the internal memory, but by setting the DMA´s priority to less than the CPU, it can be guaranteed that no CPU time is lost. This is due to the fact that the processing of the upper half image takes more time than its transfer. This would be done in the first iteration. For the next iterations, the captured image is already loaded.

If the size of the image is bigger (e.g. CIF), this approach cannot be used due to memory size limitations. The solution involves a greater number of memory transfers. First, the part of the image that has to be processed is divided in smaller rectangular parts that cover a set of triangles. To begin processing, the corresponding parts of the texture buffer and the captured image buffer containing a set of triangles have to be transferred to internal memory.

Depending on the size of these rectangles, we can choose among two different update possibilities. The first one is to load the biggest possible blocks (of the texture and the input image) by using all available memory. In this case, when the processing of a block ends, the transfer of the next block can begin. This approach has the disadvantage of leaving the CPU idle during DMA transfers.

If smaller blocks are chosen, the transfer of block n+1 can occur simultaneously with the processing of block n. The sizes of the blocks have to be chosen in such a manner that there is enough memory for two complete sets.

The major disadvantage of this approach comes from the fact that as the size of the blocks decreases, so does the efficiency of the memory transfers. The DMA controller can only transfer rectangular blocks, and the processing is done on triangles. If there is only one triangle per block, the efficiency of the DMA transfers is less than 50%. That is, for 100 pixels transferred, less than 50 are used in the calculations. As we increase the size of the blocks, so does the efficiency. This happens because we are now able to pack the data in a better way inside the blocks. See Figure 5. This is the approach we chosefor our system.

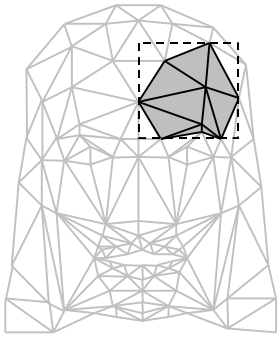

Figure 5 Useful regions of a block

The main two differences between the C6201 and the C6201B, [12], are: higher clock frequencies and a separation of the memory in two different blocks that can be accessed in parallel.

These differences do not offer any advantage apart from permitting a greater number of iterations to be performed in the same time interval (due to a faster clock). A change in the memory transfer algorithm is not needed. The same criteria of the C6201 to select the algorithm can be used.

The C6211, [13] and [14], is the low cost member of the C62xx family. The main difference with the C6201 is a slower clock and a different memory organization. The slower clock rate will directly affect the performance of the system. Its internal memory organization is the source for the change to the memory management algorithm.

The C6211 has 64KB of memory that can be configured as data memory or level 2 cache, [15]. For our system, this internal memory is configured as 32KB, 2 way associative L2 cache and two 16KB data memory blocks. The two memory blocks can be used in the same manner that the data memory of the C6201/C6201B was used.

There is no need to store the 3D model data in internal memory. This data can be accessed through the L2 cache, providing a high hit rate. This provides a source of simplification in the algorithm, and permits the use of more complex data structures. The simplification would only affect the outer loops of the algorithm. The main kernel would remain unmodified.

Software pre-fetching could be used to reduce the amount of data transferred, but the C62x hardware does not support it.

With the C6202, [16] and [17], the selection of algorithms is easier. Due to its increased memory, it is possible to store all the data needed if we work with QCIF images. If the image being used is CIF, the rectangular blocks used can be bigger. The same technique used for the C6201 can be used, but in this case, the memory transfers are more efficient. We can now have two big complete sets of data to be processed.

The C6203, [18] and [19], has 512KB of internal data RAM. This is enough memory to store several copies of the images in it, even if the image size is CIF. Additionally, the high clock frequency of this device, makes it specially suited for the implementation of this algorithm.

Table 3 indicates the memory algorithm choice based on image size and C62xx device. The members with larger internal memories use the algorithms with less external memory bandwidth, as expected. Store all means that all the image buffers needed are stored in internal memory; small blocks means that two small block buffers are used to store data; and big blocks means that bigger blocks can be used in the previous approach.

Table 3 Algorithm selection

	QCIF	CIF
C6201/C6201B	Store all	Small blocks
C6211	Small blocks	Small blocks
C6202	Store all	Big blocks
C6203	Store all	Store all

4 Results

At this moment, just the main motion estimation kernels have been coded in assembly language. The rest of the program is written in C.

The global motion estimation kernel has been written in partitioned serial assembly, [20]. After being optimized by the assembly optimizer [10], we obtain a software-pipelined loop of 26 cycles with two iterations in parallel. With 110 instructions in the loop, this gives over 4 instructions per cycle. This is far less than the ideal limit of 8.

The number of cycles for this algorithm is limited by the number of general purpose registers. Ideally, all the coefficients of the H^TH and H^TY matrices should be stored in internal registers. With this, the loop could be executed in around 16 cycles. This would lead to more than 20 live registers, close to the 32 register limit of the C62xx core. If we add other live registers, such as pointers to needed structures, we soon reach the register limit. To solve this, some coefficients of the matrices are stored in memory. This changes the lower bound of the algorithm to 24, due to memory accesses.

Two ways to improve performance of this algorithm by making modifications to the architecture would be increasing the number of general purpose registers, and including a load double word instruction similar to the C67 lddw instruction, [10]. The increase in available registers would improve software pipelining by allowing more stages to proceed in parallel. The lddw would produce a reduction in the usage of the D units.

If these two modifications were done, the main limit to the processing speed, would be the number of multipliers available. The algorithm used is mainly composed of

MAC operations that would greatly benefit from more multipliers.

In a similar manner, the local motion estimation has been written using partitioned serial assembly. As mentioned in the section before, a general version was coded and then specialized versions with a fixed number of FAPs were written.

The general version has one parameter that specifies the number of FAPs to be used. This version is composed of several nested loops, one outer loop that processes all the pixels in one line, and three inner loops that calculate the variable number of coefficients of a pixel. The main problems present here are the nesting of the loops and the unknown number of FAPs at compile time. This leads to quite inefficient code, as can be seen in Table 4. The number of cycles has been obtained by optimized serial assembly. By hand coding, a number of cycles four times lower is expected.

The optimized versions with fixed number of parameters have smaller number of cycles. This is possible because the two nested loops are converted into only one loop by completely unrolling the inner one. Software pipelining further improves performance. The cycle count for the different versions can be seen on Table 4. This cycle count has been obtained from optimized partitioned serial assembly.

Table 4 Local estimation cycles

FAPs	Generic Code	Optimized Code
1	94	4
2	122	10
3	154	20

Simulations are to be done to obtain performance figures with different memory algorithms and C6X devices.

Even though we do not have exact figures for a complete system, we can make some calculations on an upper bound on the expected performance. Assuming that 25% of the pixels in the image are to be processed, and that each pixel being processed takes 26(global movement)+50(local movement) cycles, this would give, for a QCIF image:

176x144x(26+50)x0.25=481,536 cycles

For a 200MHz device, this would mean that we could iterate more than 400 times per second. If our target frame rate was 10 frames per second, we would be able to iterate 40 times per frame, more than enough to correctly calculate the movement. This figure would be a peak rate. A more reasonable figure, to account for other processing, would be dividing it by 4, which still is enough for real time.

5 Conclusions

In this paper we have described a facial motion estimation algorithm. This algorithm is able to calculate global and local motion of the face. The implementation on the C62xx family has been discussed. Initial results on the assembly coding of the main kernels have been presented. Then, different memory management algorithms for the problem at hand have also been proposed. A decision on the best algorithm for each device has been proposed. Though there are no results to contrast this intuitive assignment, it is believed that they will perform as expected. Simple calculations have shown that real time performance could be achieved with a 200MHz device. Among all the C62xx devices, the greatest performance is expected from the C6203, which, due to its increased clock frequency and internal memory size, is not a surprise. This implementation could be the starting point for a complete MPEG-4 facial animation encoder for the C62x platform.

Acknowledgements

The authors wish to thank M. Ho, J.A. Muñoz and S. Alexandres. Without them all, this document would not have been possible. Part of this project has been carried out under the TI Elite University Program. R. Lauwereins is a senior research associate of the Flemish FWO.

References

[1] ISO/IEC 11172, Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s.

[2] ISO/IEC 13818, Generic Coding of Moving Pictures and Associated Audio.

[3] ITU-T SG 15 WP 15/1, "Draft Recommendation H.263 (Video coding for low bitrate communication)," Doc. LBC-95-251, October 1995.

[4] MPEG Video and SNHC, "Text of ISO/IEC FDIS 14496-2: Visual," Doc. ISO/MPEG N2502, Atlantic City MPEG Meeting, Oct. 1998.

[5] ISO/IEC 14496-2 Coding of audio-visual objects Part 2: Visual

[6] Koch, R. "Dynamic 3-D Scene Analysis through Synthesis Feedback Control", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 15, No. 6, pp. 556-568, June 1993.

[7] C. S. Choi, K. Aizawa, H. Harashima, and T. Takabe, "Analysis and Synthesis of Facial Image Sequences in Model-Based Image Coding", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 4, No. 3, pp. 257-275, June 1994.

[8] H.Li, and R.Forchheimer, "Two-View Facial Movement Estimation", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 4, No. 3, pp. 276-287, June 1994.

[9] H. Li, and P. Rovainen, "3-D Motion Estimation in Model-Based Facial Image Coding", IEEE Transactions on Pattern

Analysis and Machine Intelligence, Vol. 15, No. 6, pp. 545-555, June 1993.
[10] "TMS320C6000 CPU and Instruction Set Reference Guide", Texas Instruments SPRU189D, March 1999.
[11] "TMS320C6000 Optimizing C Compiler User's Guide", Texas Instruments SPRU187E, February 1999.
[12] "TMS320C6201/TMS320C6201B Datasheet", Texas Instruments Datasheet SPRS051E., May 1999
[13] "TMS320C6211 Datasheet", Texas Instruments Datasheet SPRS073A, March 1999.
[14] "How to Begin Development Today with the TMS320C6211 DSP", Texas Instruments application report SPRA474, September 1998.
[15] "TMS320C6211 Cache Analysis", Texas Instruments application report SPRA472, September 1998.
[16] "TMS320C6202 Datasheet", Texas Instruments Datasheet SPRS072A, January 1999.
[17] "How to Begin Development Today with the TMS320C6202 DSP", Texas Instruments application report SPRA473, September 1998.
[18] "TMS320C6203 Datasheet", Texas Instruments Datasheet SPRS086A, May 1999.
[19] "How to Begin Development Today with the TMS320C6203 DSP", Texas Instruments application report SPRA570, May 1999.
[20] "TMS320C62x/C67x Programmer's Guide", Texas Instruments SPRU198C, May 1999.