ESIEE Engineering

IF4-ARCH - PROJECT

Matěj Kubička
Divij Babbar

8.1.2011

Contents

List of Tables

List of Figures

List of listings

List of terms and abbreviations

BIOS	Basic Input-Output System
B&W	Black and White
CSL	Chip-Support Library
DSP	Digital Signal Processor
DDR2	Dual Data Rate 2
EDMA	Enhanced Direct Memory Access
EMIF	External Memory Interface
FP	Floating Point
FPS	Frames per second
GPP	General Purpose Processor
GPR	General Purpose Registers
HAL	Hardware Access Layer
JTAG	Joint Test Action Group (IEEE 1149.1)
LUT	Lookup table
MIPS	Milion instructions Per Second
MSB	Most significant bit
PP	Parallel Processing
TI	Texas Instruments

1 Introduction

Our goal is to implement and optimise a series of image processing algorithms on a DSP development kit TMS320DM6437. The result should produce real-time line detection mechanism on the image stream.

Firstly, we capture an image from the camera via the DSP and then we make it treatable by converting it into a black and white image. Second, we use on the black and white image, the Deriche Filter (optimised by Garcia-Lorca). The Deriche derivator uses the Robertson operator to get the gradients of the edges[1]. Then we binarize the image by tresholding and apply a Hough transform. The Hough transform will give a result on the hough plane, where the points with the maximum intensities will represent the lines that are occurinng the most. For detection of the lines we use thresholding which is an effective technique put in place to select the lines with highest votes. Once, we have extracted the relevant peaks on the hough plane then we can map them back onto the image and get the lines.

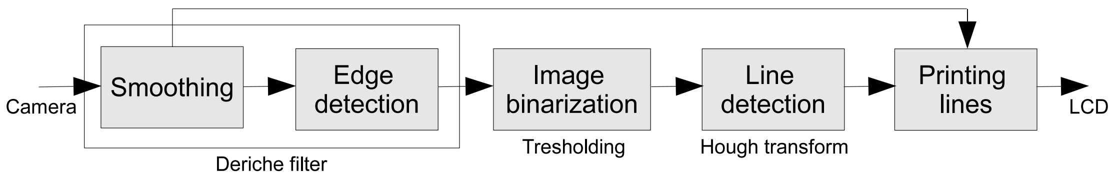

Figure 1: Image processing chain

Deriche operators have emerged for edge detection in many application areas in image processing and in computer vision. Their main drawback is their slow performance due to the large volume of calculations they require. However, their good results led to study different solutions to accelerate their performance. One of the optimised version is the Garcia Lorca approximative Deriche algorithm. The Garcia Lorca filter reduces the complex equations of the Deriche filter into two cascaded filters in causal and anti-causal senses. The causal and anticausal equations are represented and explained in the proceeding chapters. As a by-product of this implementation we get the gradients of the image in question. These gradients are kept for later use.

Binarisation is simply an application of thresholding. We select an intensity value for the pixels. Whatever is above this particular value is given the highest intensity, and whatever is below is given the lowest intensity. This renders the image comparable to a zero or one format since there are only two pixel values in question now, and hence the term binarisation. The binarisation is a necessary step because the Hough transform acts on the binary images.

Hough's transform is a method to represent all possible points on lines, as points. This is an application of the fact that a line can be uniquely identified by two parameters, which make the x and y axes of the hough space (called accumulator). If we map all the possible points (which implies all lines by extension) into the Hough space then we would obtain an ordered space where each point would basically represent a line from the original space. Once, we have the Hough accumulator we can extract and print[2] the lines.

[1] These gradients will be used again while using the optimised version of the Hough transform

[2] How to print a line is explained in proceeding chapters.

1.1 Document organisation

Next chapter covers used hardware equipment, its parameters, performances and software tools used for programmation, compilation, and profiling measurements. Chapter 3 contain information and analysis of used algorithms. Chapters 4 and 5 cover a basic version of our processing chain - we deal with very basic implementation of given image processing chain on a PC and on a DSP. In chapter 5 we try to apply algorithm-architecture matching techniques to optimize performance of the basic implementation on a DSP.

Finally, Chapter 6 aims to optimise used algorithms while keeping algorithm-architecture techniques. Two versions are measured, they both share the same code, but one is without compiler optimizations and one is with -O3[3] compiler optimizations.

In the last chapter we compare all versions, make conclusions and make some important remarks.

1.2 Measurement methodology

We have measured execution time of every used operator - Deriche smoother, Deriche derivator, Hough transform and printing of the result. To get statistically relevant results, 10 measurements ($x_0 ... x_9$) were done for each operator and average values were used for profiling (for calculation see $\bar{x}$ in (1)). To transform results from instruction cycles to seconds, we've noted CPU clock frequency as $f_{CPU} = 594 \cdot 10^6 Hz$. From average processing time in instruction cycles we can calculate processing time in miliseconds (equation (2)) and in percent (equation (3)).

$$\bar{x} = \frac{1}{10} \sum_{i=0}^{9} x_i \tag{1}$$

$$T[ms] = \frac{1}{1000} \frac{\bar{x}}{f_{CPU}} \tag{2}$$

$$T[\%] = \frac{\bar{x}}{f_{CPU}} \cdot 100 \tag{3}$$

Frames per second (FPS) indicates processing chain performance, how many images are processed per second. We have calculated fps of whole system and fps of every and each operator. It is a purely theoretical value which indicates algorithm performance. Operations per pixel (OPPX) indicates how many clock cycles were necessary to compute value of a single pixel at the output. We have calculated it again for whole system and for every operator separately. Gain was calculated as a ratio between operator performance in previous and actual version. As in previous cases, we have measured gain between operators separately and between two versions of the whole processing chain.

$$FPS = \frac{f_{CPU}}{\bar{x}} \tag{4}$$

$$OPPX = \frac{\bar{x}}{300 \cdot 200} \tag{5}$$

[3] Optimization aiming processing speed and allowing increase of the program code.

$$Gain = \frac{\bar{x}_{REF}}{\bar{x}} \tag{6}$$

To calculate performances of whole system (FPS, OPPX, CPU load), we had to measure processing time outside our processing chain. We have measured total time between two frames (denoted as T_P) and time required to process implemented processing chain (T_{CH}). With T_{CH} we can calculate system FPS and OPPX. CPU load is calculated as a ratio between T_{CH} and T_P, see equation (7). This indicator shows how much CPU time in percent is used by the image processing chain itself.

$$CPUload = \frac{T_{CH}}{T_P} \cdot 100 \tag{7}$$

2 Hardware & software resources

To fulfill the task, we have created a program running on x86 GPP (PC) which allowed us to quickly design first version of the processing chain. We wrote a PC version and then merged our results together to create a version running on a DSP.

Given DSP came with BIOS, a HAL called CSL (Chip-support library) and with demo-application which sampled data from camera and sent them directly to the LCD screen. This allowed us to overstep all the allocation, loading, storing and data transferring problems. Please note that the CSL uses dedicated DMA channel to transfer between camera/LCD and external memory through one of the two EMIF's[4] [3].

2.1 Development kit & environment

The application was tested on a development board TMX320DM6437 from TI. The manufacturer emphasizes following features [8]:

- 600-MHz C64x+ DaVinci CPU (4800 MIPS)
- One Video input via NTSC/PAL or RAW data
- One Video output via NTSC/PAL and YpbPr/RGB
- Audio I/O: S/PDIF Interface, analog, and optical
- PCI, 10/100 Ethernet MAC
- UART, CAN I/O, and VLYNQ
- 16 Mbytes of non-volatile Flash memory,
- 64 Mbytes NAND Flash, 2 Mbytes SRAM
- 128 Mbytes of DDR2 DRAM

As a development environment we have been provided with TI's Code Composer Studio (in short CCS or CCStudio) which is IDE with editor, compiler, programmer and debugger. The debugging was done via USB, although there is on-board JTAG interface, but we haven't used it.

2.2 TMS320C64x+ processor architecture

TMS320C64x+ is a 600Mhz (4800 MIPS peak) fixed-point DSP with VLIW [5] architecture and 256 bits long instruction word. Single instruction word contain 8 fields for primitive instructions and therefore capability to execute up to 8 instructions in PP. CPU consists from 32 fixed point GPR's in two datapaths (64 GPR's in total) and eight functional units, each equipped with 6 ALU's and two multipliers. Each of the 8 functional units (.M1, .L1, .D1, .S1, .M2, .L2, .D2, and .S2) is capable of executing one instruction

[4] It uses external memory interface A (EMIFA, on-chip DSP peripheral).

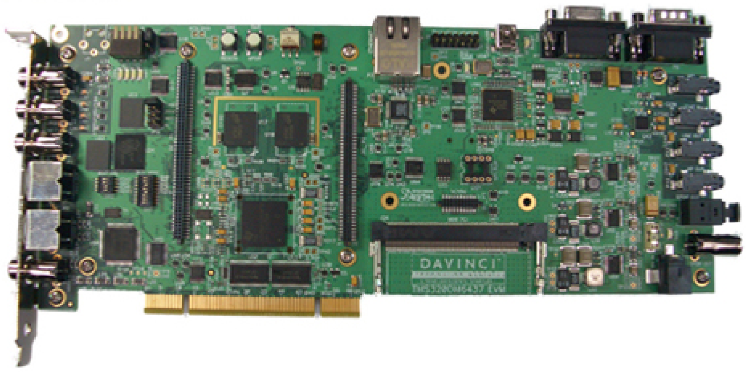

Figure 2: TMX320DM6437 EVM DaVinci board

every clock cycle. The .M functional units perform all multiply operations. The .S and .L units perform a set of arithmetic, logical, and branch functions. The .D units are designed to load /store data from/to memory [3]. Every .M unit can perform 1 multiplication of two 32 bit words, or 2 multiplications of 16 bit shorts, or 4 multiplications of two octets per clock cycle.

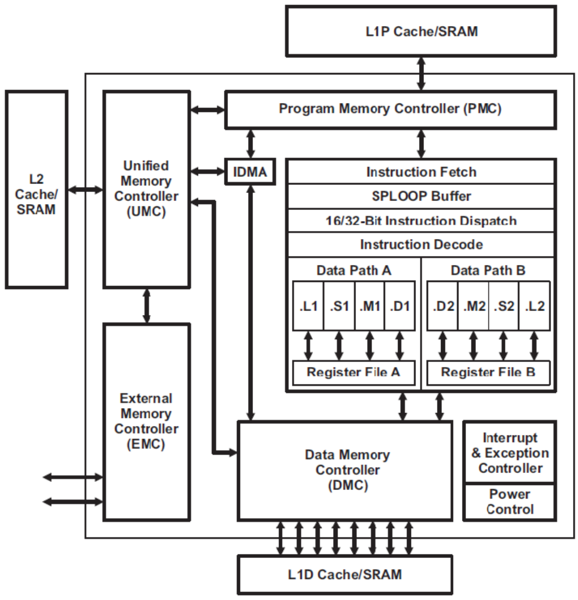

Figure 3: TMS320C64x+ DSP Block Diagram (taken from [4])

2.3 Memories description

The DSP datasheet says on the topic of memories: The C64x+ core uses a two-level cache-based architecture. The Level 1 Program memory/cache (L1P) consists of 32 KB memory space that can be configured as mapped memory or direct mapped cache. The Level 1 Data memory/cache (L1D) consists of 80 KB - 48 KB of which is mapped memory and 32 KB of which can be configured as mapped memory or 2-

way set associated cache. The Level 2 memory/cache (L2) consists of a 128 KB memory space that is shared between program and data space. L2 memory can be configured as mapped memory, cache, or a combination of both. (taken from [3]).

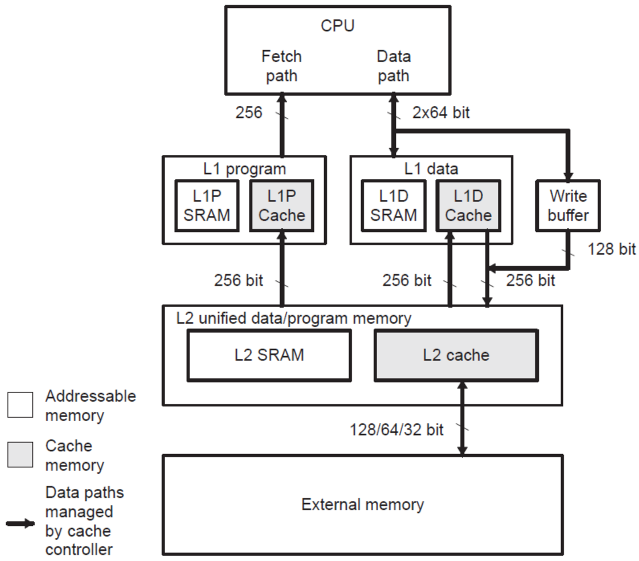

Figure 4: C64x+ Cache memory architecture

Access time to L1 cache and as well to L2 cache and RAM is 1 clock cycle. Both memories are SRAM running on the same frequency as the CPU. For further information see [6]. External memory is DDR2[5] connected through EMIFA peripheral of the DSP. Data are transferred with EDMA peripheral support. Used DDR2 memory uses 166Mhz memory clock, so access times are at approximately 4x times higher than in case of on-chip cache memories.

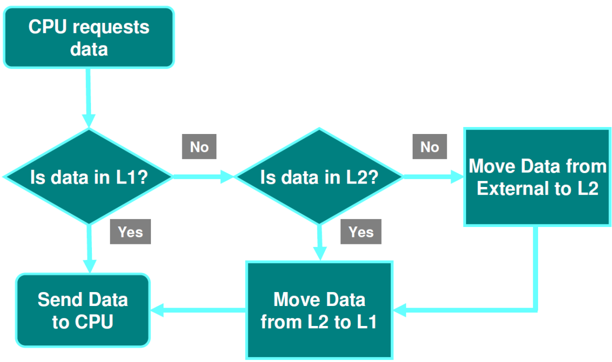

Figure 5: Cache data flow (see [7])

[5]Type DDR2-667 by JEDEC

2.4 Profiling techniques

For results analysis we have measured absolute CPU clock time between every operator. We always sample 10 consecutive frames and calculate average values per operator for analysis purposes. CSL library function C64P_getltime() returns actual CPU clock time as 32 bit integer.

Listing 1: Time measurement approach

```
unsigned int profiling[50];
for(i=0;i<NO_ITERATIONS;i++) {
  /* Load the image from camera */
  ...

  /* Apply processing chain */
  v=i%5;
  profiling[5*v] = C64P_getltime();
  deriche_gl(0.2);
  profiling[5*v+1] = C64P_getltime();
  roberts(30);
  profiling[5*v+2] = C64P_getltime();
  hough_transform();
  profiling[5*v+3] = C64P_getltime();
  print_lines(150);
  profiling[5*v+4] = C64P_getltime();

  /* Show the result on LCD screen */
  ...
}
```

3 Algorithms description & analysis

In this chapter we analyze performance of used algorithms. All used operators are evaluated: deriche smoother, deriche derivator, tresholding for binarisation and a hough transform. The analysis is done for calculative operations like additions and multiplications and therefore it is not an absolute measure of algorithm running time. First of all, used architecture uses parallel processing, so number of instructions does not equal to number of cycles. Additionaly, and more importanly, calculations do not involve program flow control (loops, function calls, etc.) and memory accesses, which take most of the processing time. In practice, running time of used algorithms is by several orders higher than calculated.

3.1 Deriche smoother

Frederico Garcia Lorca version of Deriche smoother is using 2^{nd} order filter to smooth the image. We smooth horizontally and vertically in causal and anticausal way[6]. Complete algorithm analysis can be found at [9].

Listing 2: Deriche smoother

```
/* Horizontal smoother */
for(int i=0;i<height;i++){
  /* Causal way */
  for(int k=2;k<width;k++){
      La[i][k] = g1*Image[i][k] + g2*La[i][k-1] - gg*La[i][k-2];
  }
  /* Anti-causal way */
  for(int k=width-3;k>=0;k--){
      Lb[i][k] = g1*La[i][k] + g2*Lb[i][k+1] - gg*Lb[i][k+2];
  }
}

/* Transpose the image */
transpose(Lb, width, height);

/* Vertical smoother */ /* Note: it is horizontal smoother on transposed image */
for(int i=0;i<width;i++){
  /* Causal way */
  for(int k=2;k<height;k++){
      Lc[i][k] = g1*Lb[i][k] + g2*Lc[i][k-1] - gg*Lc[i][k-2];
  }
  /* Anti-causal way */
  for(int k=height-3;k>=0;k--){
      Ld[i][k] = g1*Lc[i][k] + g2*Ld[i][k+1] - gg*Ld[i][k+2];
  }
}
/* Transpose the image back */
transpose(Ld, width, height);
```

[6] In simple terms it means from left to right, from right to left, up to down and down to up.

For simplified version of horizontal smoothing implementation see listing above. Horizontal smoothing is done in a way that we go through all the image lines and smooth them in causal way (by going from left to right) and then in an anticausal way (by going from right to left). Vertical smoothing is done in the same way, only the image matrix has to be transposed. There are $320 \times 200 \times 4 = 240,000$ loop iterations in which we do 6 multiplications and 4 additions per pixel.

The horizontal smoothing accesses memory row-by-row, which provides better locality for cache memory performance than column-by-column approach manifested in vertical smoothing without transposition. The problem is the image transposition, we have to do it manually and therefore the problem with locality is not solved.

3.2 Deriche derivator

For edge magnitude and gradient detection we use Roberts operator made of convolution of two core matrices R_H and R_V (8) with the image. As a result we get two convoluted images N_H and N_V (9). Core matrices are moved by 90 degrees between each other, R_H core matrix detects edges in horizontal direction and the R_V in vertical direction. As a result, convoluted images N_H and N_V are moved by 90° to each other too. Therefore the image gradient is a by-product of the edge detection mechanism - horizontal matrix R_H provides X coordinate and vertical matrix R_V provides Y coordinate of the gradient. If we calculate magnitude (10) we get resulting intensity of the edges. If we count argument (11) we get the direction of the gradient.

$$R_H = \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} R_V = \begin{bmatrix} -1 & -1 \\ 1 & 1 \end{bmatrix} \tag{8}$$

$$N(x, y) = \sum_{i=0}^{1} \sum_{k=0}^{1} R_X(i, j) I_{SRC}(x + i, y + j) \tag{9}$$

$$|E(x, y)| = \sqrt{N_H(x, y)^2 + N_V(x, y)^2} \tag{10}$$

$$\angle E(x, y) = \arctan\left(\frac{N_V(x, y)}{N_H(x, y)}\right) \tag{11}$$

ANSI-C implementation is listed below. We have 6 additions per pixel and $(height - 2)\dot{(}width - 2)$ loop iterations which makes about 350,000 operations for image of size 300×200 pixels. Please note that our implementation of this algorithm is not creating new image matrices for both convolutions, but directly counts magnitude (euclidian distance in 2D space) and the argument (an angle).

Listing 3: Deriche derivator - Roberts

```
#define gpx(x,y) m[x+(y)*width]
float grady[width*height], gradx[width*height]

void roberts()
{
  int j, k;
  for(k=1; k<height-1;k++) /* Lines */
    for(j=1; j<width-1;j++) { /* Columns */
      /* ROBERTS */
      gradx[j+k*width] = -gpx(j,k)-gpx(j+1,k)+gpx(j,k+1)+gpx(j+1,k+1);
      grady[j+k*width] = -gpx(j,k)+gpx(j+1,k)-gpx(j,k+1)+gpx(j+1,k+1);
    }
}
```

3.3 Thresholding & binarisation

This algorithm basically thresholds the image at some intensity level and says that every pixel below threshold have lowest intensity (0) and every pixel above threshold have highest intensity (255). C language implementation is listed below. We suppose that input image is resulting edge magnitude. This algorithm contain no additions or multiplications, just comparation at each pixel on the image, 1 read access and 1 write access per pixel.

Listing 4: Image binarisation

```
void binarize(float * abs, float treshold)
{
  int j, k;
  for(k=0; k<height;k++) /* Lines */
    for(j=0; j<width;j++)  /* Columns */
      abs[j+k*width] = (abs[j+k*width]>treshold)?255.0:0;

}
```

3.4 Hough transform

This algorithm is suposedly most exhaustive amongst all the ones in use here, because it has algorithmic complexity of $O(n^3)$. The other algorithms are quadratic ($O(n^2)$). Hough trasnform takes every pixel of binarised image and looks whether it is an active pixel (white pixel). If the pixel is white the algorithm adds a vote to every possible line which goes through this pixel.

Votes are stored in the Hough accumulator space with two dimensions: ρ and ϕ. The ρ is a distance and ϕ is angle. In other words, Hough accumulator space is a space with polar coordinates. The transform between polar and cartesian coordinates is given by:

$$\left.\begin{aligned} x &= \rho cos(\phi) \\ y &= \rho sin(\phi) \end{aligned}\right\} \tag{12}$$

$$\left.\begin{aligned} \rho &= \sqrt{x^2 + y^2} \\ \phi &= atan(\tfrac{y}{x}) \end{aligned}\right\} \tag{13}$$

Therefore, Hough transform takes every white pixel and adds vote to every line with distance given by $\rho = x \cdot \cos(\phi) + y \cdot \sin(\phi)$ and with every possible angle (from 0 to 180). The algorithm in C language is listed below. Please note that $a \cdot \pi/180$ in the Listing 5 is a conversion of angle a in degrees to radians. This algorithm does two additions and two multiplications for 180 angles of each white pixel on the image. Therefore, in worst case of completely white image we get $300 \times 200 \times 180 \times 4$ operations (43,200,000 operations for 300×200 pixels image).

Listing 5: Standard Hough transform algorithm

```c
void hough() {
  /* run-in-vars */ int k,j,a; float rho;
  /* maximum distance */ float maxrho = sqrt(width*width+height*height);
  /* cleaning */ for(k=0; k<360*180;k++) h[k] = 0;
  /* calculating accumulator space */
  for(k=1; k<height;k++) /* Lines */
    for(j=1; j<width;j++) /* Columns */
      if(Image[j + k*width])
        for(a=0;a<180;a++) { // Angles
          rho = (j*cos(a*PI/180.0) + k*sin(a*PI/180.0));
          h[a+180*rho]++;
        }
}
```

3.5 Printing detected lines

We use linear algebra to find the line. First of all we convert the line described as pair of $\{\rho, \phi\}$ into following form:

$$\begin{aligned} A_x + d_x k &= X \\ A_y + d_y k &= Y \end{aligned} \tag{14}$$

where $[A_x, A_y]$ is a known point on a line, (d_x, d_y) is a direction vector and k is multiplication constant. The known point can be retrieved simply from the hough accumulator space by transforming polar coordinates in accumulator to cartesian coordinates on the picture.

$$\begin{aligned} A_x &= \rho cos(\phi) \\ A_y &= \rho sin(\phi) \end{aligned} \tag{15}$$

To find a direction vector (d_x, d_y) we can take advantage of $[A_x, A_y]$. If we consider $[A_x, A_y]$ coordinates as a vector, we get a direction (A_x, A_y), which is perpendicular to the line direction. To get the direction vector, we just rotate the vector by 90° as it is done at (16). Resulting vector will have correct direction, but unknown size. In order to be able to draw lines pixel by pixel with k simply incremented by 1, we have to normalize the direction vector to have a size 1 (17).

$$(d_x, d_y) = (-A_y, A_x) \tag{16}$$

$$(d_x, d_y)' = \frac{(d_x, d_y)}{||d||} \tag{17}$$

With representation of a line in (14), we can look for an intersections with rectangle given by image boundaries . When looking for intersection with a line we are trying to solve following system of linear equations (18) to find k_1 and k_2.

$$\begin{aligned} A_1x + d_{1x}k_1 &= A_2x + d_{2x}k_2 \\ A_1y + d_{1y}k_1 &= A_2y + d_{2y}k_2 \end{aligned} \tag{18}$$

To get intersections with a rectangle, we have to test intersection with all 4 lines on all sides. We can obtain following results: (note: we have decided to count number of intersections and print only lines with 2 intersections).

- 1 intersection when line is going through a corner
- 2 intersections between any two sides
- ∞ intersections for line overlapping with rectangle side

4 Basic implementation (version I)

This was a very first version we implemented, with no optimisation. The image processing chain was designed and tested in synthetic environment on a PC and then ported to a DSP.

4.1 Features

- All operations in floating point (IEEE 754)
- Classic version of Hough transform for lines detection
- Used trigonometric functions from C standard library
- Frederico Garcia Lorca version of Deriche filter
- Roberts operator used for Deriche derivator

4.2 Basic implementation on PC

In order to speed-up development process we decided to setup a PC environment in which we can develop our application. We have used a program for smoothing images given as a part of practicals related to a cache memory and changed it in a way in which it loads an image, applies processing chain and stores the result to a file. The program was written in C language, compiled with GCC. Profiling results are shown in Table 1. Appendix A contain source codes of this program[7].

To be able to run tests quickly, we wrote a script which compile the source code, run the program and show the results. The script takes care about correctness of given arguments and also checks whether compilation was successful. It allowed us to make development noticeably quicker.

We have also created a script for profiling. It uses gprof profiling tool to generate performance statistics. Both scripts and profiling result are enclosed in Appendix A.

Function name	Time [%]	Total time [s]	Time/call [ms]	Calls [-]
deriche_gl	0	0.00	0.00	7
roberts	0	0	0	7
hough_lines_slow	93.75	0.15	21.43	7
print_lines	0	0	0	7
main	6.25	0.01	0.01	7

Table 1: PC version profiling results

The software was tested on a machine with Intel Core 2 Duo P8440 (2.26Ghz/3M L2 Cache) CPU under OS Debian 6.0.2. Compiler optimizations were disabled. Results clearly shows that hough trasform consumes most of the computing time, compared to other operations in the image processing chain. Function main() takes about 6% of time, but that is not essential for the result.

[7] As well as profiling results and used shell scripts.

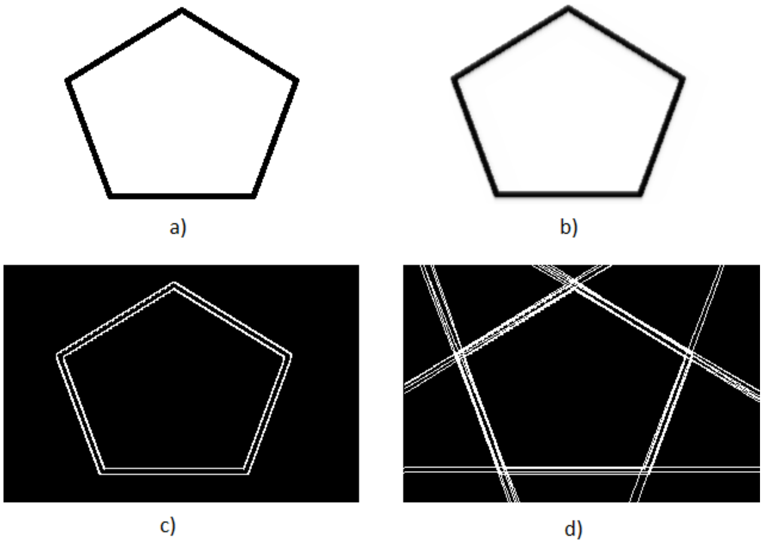

Figure 6: Example: (a) original image, (b) smoothed image, (c) binarized edges, (d) result

For the testing purposes our constants were set to 0.8 for the smoothing filter, edges were tresholded at level 160 for binarisation and all lines in hough accumulator space with more than a 125 votes were shown in the resulting image. For the result see Figure 6.

4.3 Basic implementation on DSP

The PC version was designed in a way that porting to a DSP is simple as possible, hence porting of our source code to a C64 series DSP was quite straight-forward. Program for DSP was changed for a fixed-size image (300px to 200px) and routines for loading/storing the image matrix were added. Image processing chain was placed between these two routines.

4.4 Implementation notes

We are working with image of size 300×200 in grayscale (light intensities) and using a floating point matrices (according to IEEE 754). All calculations are done in floating point, although our processor doesn't have hardware support for decimal numbers.

4.4.1 Basic version of Deriche smoother

Right from the beginning, we have run into troubles with implementing smoother of Garcia-Lorca version of a Deriche filter. Our result kept being unstable. There were several reasons:

- We misused constants gamma and alpha. Gamma is equal to minus exponent of alpha, but we took gamma directly as alpha
- There was a mistake in counting pixel values we put to the part of equation wrong operator (there should have been minus, but we put a plus)

- There were issues with storing data to correct matrices

Those mistakes accumulated overtime unknown algorithm proven to be problematic to identify. The second issue also created a problem with float overflowing to $+\infty$.

4.4.2 Roberts operator

Implementing a Roberts operator for edge detection and gradient value was quite straight-forward and unproblematic. Only drawback was that we had to be careful to do the calculations correctly. Originally we have interchanged N_X with N_Y and although that haven't changed result magnitude, the gradient direction was different. Directly after edge extraction, we did a thresholding to binarize the image.

4.4.3 Hough transform

We have prepared accumulator space made of unsigned short datatype, because longest possible line on our picture has 360 pixels, which is the maximum number of points which can vote for one line.

4.5 Performance analysis

Performance was evaluated using approach described in chapters 2.4 Profiling techniques and 1.2 Measurement methodology. We are making 10 tests and use average values to calculate processing times, frames per second (fps), operations per pixel (oppx). Memory usage and memory calls are evaluated from the source code.

Function name	time [%]	time [ms]	fps	oppx[8]	mem. usage[9]	mem. accesses[10]
Deriche smoother	6.75	164.99	6.06	1633.47	1.44MB	12rd + 4wr
Deriche derivator	3.30	80.68	12.39	798.81	480kB	8rd + 1wr
Hough transform	89.29	2181.09	0.46	21592.79	369.6kB	1rd + 180wr
Printing result	0.13	3.19	312.63	31.66	369.6kB	-

Table 2: Basic version profiling results

- Resulting values are average calculated out of 10 measurements
- Performance result for the alogrithm which prints the detected lines is highly dependent on actual number of detected lines.
- Memory accesses of printing line algorithm cannot be evaluated, because it depends on actual length of the line.
- Gains cannot be calculated, since this is a first version of our solution.

[8] Operations per pixel

[9] Memory usage refers to total amount of memory which is managed by the algorithm

[10] Theoretical value per pixel

From the result analysis we can see that majority of processing time is taken by the Hough transform, which slows down the system to 0.4 fps. Majority of memory is used by the Deriche filter, because it uses 4 additional image matrixes of floats for smoothing calculations. CPU is busy with the image processing chain in absolute majority of time (99.47%).

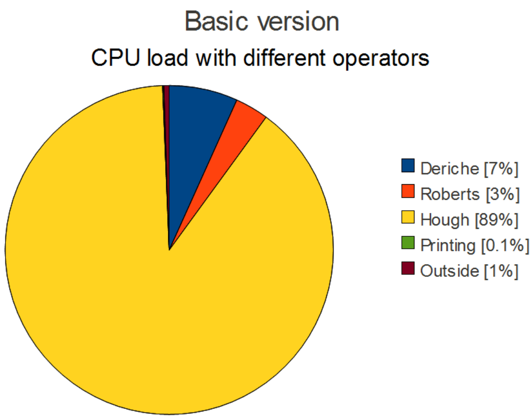

Figure 7: Basic version - CPU load by different operators

Summary of basic version shows most important indicators of this version's performance. It is resulting fps, total CPU load created by implemented image processing chain, amount of operations per pixel (oppx) and total memory usage.

Indicator	**Value**
FPS	0.412
OPPX	24056
CPU load	99.47%
Mem. usage	1569.6kB
Gain	-

Table 3: Basic version summary

4.6 Conclusions

Good thing is that this version works. On other hand, there are many problems. Used hough algorithm implementation takes just too much time to be used in real-life applications, big part of processing time is taken by floating point calculations and the Deriche smoother uses ridiculous amount of memory for smoothing.

There are many ways to optimize this version. Key to good performance is probably optimizing Hough algorithm itself, then move the calculations from floating point to fixed point and also we should take advantage of VLIW architecture and help compiler to be more effective.

5 Algorithm-architecture matched version (version II)

5.1 Features

- Algorithm-architecture matching techniques.
- Optimization with loop unrolling, software pipelining, and register rotations applied to the algorithms from basic version.

5.2 Hardware notions

From profiling results of the basic version we've seen that most of the procesing time is taken by the Hough transform itself. This version doesn't change any algorithm, but it optimizes previously used algorithms to better utilize its hardware resources. The hardware solution uses following parameters [4]:

- Several multiplications can be done in one cycle (with hardware multiplier)
- Several additions can be done in one cycle
- Bit shift operation of arbitrary number of bits in one cycle (with barrel shifter)
- Floating point arithmetic is not supported (fixed-point CPU)
- Access to external memory is 6x slower than to internal SRAM with size of 128kB
- Thanks to parallel processing a simple operations (like incrementing a register) are executed often with no cost, together with other instructions.
- The processor has hardware support to process up to 3 nested for loops with no overhead (with SPLOOP)

DSP architecture is VLIW, which allows paralel processing of up to 8 instruction as was stated in the introduction part. Problem is that effective usage of this feature is completely dependent on the C compiler used. We can help the compiler a bit to find an optimal solution, but in the end a great deal of the resulting performance depends on the compiler, not on the programmer.

5.3 Implementation notes

Note: because in the project description was requested to move from floating point calculations to fixed point at the third version, this kind of optimisation is not used in here, although it belongs to the set of algorithm-architecture matching techniques

5.3.1 Lowering memory usage of the Deriche smoother

Previous version required 4 additional float matrices in order to compute the smoothed image. Some type of intermediate buffer is needed indeed, but 4 additional matrices makes the system slower just because of the DMA transfers between L2 cache and the DDR2 external memory.

As was described before, we have to smooth each line of the image in two directions: causal (left to right) and anticausal (right to left). When we analyze what the algorithm actually does, we find out that the intermediate buffer is required only for storing a line smoothed in one direction. So finally, we can take line of the original image, smooth it in causal way, store the result to the line buffer and then we smooth the line in anticausal way and store it back to the image.

In this way we managed to lower memory requirements from 960kB (4 float matrices 300×200 px) to 1200B (one line of floats).

5.3.2 Smart transpositions

The algorithm first does the horizontal smoothing, then transposes image matrix and finally does the horizontal smoothing again (which technically becomes vertical smoothing). When we have the result, we have to transpose image matrix for second time in order to get back original picture representation.

This optimization lies in reordering the smoothing algorithm so we need only one additional transposition. If we suppose that we get the source image matrix already transposed, we have to do the vertical smoothing first, then transpose the matrix and then we can do horizontal smoothing. The transposition is done only once. The optimization gain comes from the way in which the image matrix is loaded from the camera driver: we can load it one way or another, but if we load it directly transposed, we save time in the deriche smoother.

5.3.3 Optimizing memory accesses of the Deriche smoother

Deriche Garcia-Lorca smoother requires 12 read acesses to a memory and 4 write access in order to smooth one pixel. With the register rotation we optimized the algorithm that it need only 4 read accesses and 4 write accesses.

In order to count pixel value we need to read last three consecutive pixels. The idea behind register rotation is to store actual pixel and use it next loop iteration as previous pixel and loop after that as pixel before previous pixel. For example implementation see listing below:

Listing 6: Deriche smoother register rotation

```
register int p1, p2, p3;
p3=ig1*(mint2[ii++]); p2=(ig1*mint2[ii++]+ig2*l[0]);
for(k=2;k<width;k++){
  p1 = p2; p2 = p3;
  p3 = (g1*(mint2[ii++]) + g2*p2   gg*p1);
  l[k] = p3;
}
```

5.3.4 Optimizing convolutions of the Deriche smoother

Robertson uses core matrices R_H and R_V and convolutes them with the source image. Result is used to get a gradient vector and intensity of detected edge. Original computation is shown in Listing 7 (for more information see chapter 3.2 Deriche derivator).

Listing 7: Deriche derivator basic version

```
for(k=0; k<height-1;k++) // Lines
  for(j=0; j<width-1;j++) { // Columns
    gradx = -in[j+k*width]-in[j+1+k*width]+in[j+(k+1)*width]+in[j+1+(k+1)*width];
    grady = -in[j+k*width]+in[j+1+k*width]-in[j+(k+1)*width]+in[j+1+(k+1)*width];
  }
```

There are 8 read memory accesses per each convolution. But pixels on address $(j+1, k)$ and $(j+1, k+1)$ will be next loop iteration on address (j, k), and $(j, k+1)$ respectively. The idea is to use actual pixel next loop iteration as a previous pixel without reading the memory. Altered algorithm is listed in Listing 8. It requires only 2 read memory accesses:

Listing 8: Deriche derivator optimised version

```
for(k=0; k<height-1;k++) // Lines
  for(j=0; j<width-1;j++) { // Columns
    p2 = mint[j+1+k*width]; p3 = mint[j+1+(k+1)*width];
    convy = -p1+p4-p2+p3;
    convx = -p1+p2-p4+p3;
    p1 = p2; p4 = p3;
  }
```

5.3.5 Loop unrolling and software pipelining

Loop unrolling is a technique which lowers overhead generated by loop control. Then, it copies the content of a loop several times into a single iteration. Software pipelining reduces overhead when next operation depends on finishing of current operation. Instead of waiting to finish, software pipeline prepares solution of the next iteration. This techniques are especially useful with VLIW architecture, which allows us to do up to 8 operations together during the same clock cycle.

Compiler optimizes loops automatically, it calculates optimal way of loop unrolling and applies software pipelining where possible. We can only help by describing minimum and maximum number of iterations of the loop, for this we have pragma MUST_ITERATE(min, max, step) [10]. For example see Listing 9 below.

Listing 9: Using of MUST_ITERATE pragma

```
#pragma MUST_ITERATE(width, width);
for(j=0; j<width-1;j++) { // Columns
  p2 = mint[j+1+k*width]; p3 = mint[j+1+(k+1)*width];
  convy = -p1+p4-p2+p3; convx = -p1+p2-p4+p3;
  p1 = p2; p4 = p3;
}
```

We told the compiler additional information about loops where it was possible. Doing software pipelining to take advantage of PP in a C code is not possible, because C language was not designed to support this feature, the compiler takes care of that. In order to apply both optimisation methods, compiler optimisation -O3 neds to be enabled. This optimisation (-O3) allows optimisation methods which make the program code larger in order to make the algorithm running faster.

5.4 Performance analysis

Performance was evaluated using approach described in chapter 2.4 Profiling techniques and 1.2 Measurement methodology. We made 10 tests and used average values to calculate processign times, frames per second (fps), operations per pixel (oppx). Memory usage and memory calls are evaluated from the source code.

Function name	**time[%]**	**time[ms]**	**fps**	**oppx**[11]	**mem. usage**[12]	**mem. access**[13]	**gain**[14]
Deriche smoother	7.92	163.27	6.12	1616.40	481.2kB	4rd + 4wr	1.01
Deriche derivator	0.76	15.84	63.10	156.87	480.0kB	2rd + 1wr	5.09
Hough transform	88.58	1825.94	0.54	18076.82	369.6kB	1rd + 180wr	1.19
Printing result	0.18	3.72	268.47	36.87	369.6kB	-	0.86

Table 4: Profiling results of algorithm-architecture matched version

We can see that Hough transform keeps on being problematic, although performance was raised by 20%, the algorithm is still incredibly slow, resulting in slowing down the system to almost the same perfomance as in previous version.

Deriche smoother memory usage was lowered from 1.44MB to 1.2kB, which is improvement of more than 1000 times. Unfortunately speed performance of this algorithm didn't change much, only by 1%. We account this result to floating point operations, which make every loop iteration heavy and hence loop unrolling and software pipelining are not really effective.

Deriche derivator is 5.3 times faster than in previous version thanks to rotating registers. We lowered memory accesses from 8 to 2 reads per pixel and also managed to keep other 6 pixels in the GPR register bank, which made them quickly accessible.

Line printing mechanism seems to be bit slower than in the previous case, but it doesn't matter. The processing time varies greatly with the number of detected lines and therefore performance result depends upon image processing chain settings and upon image picture properties.

[11] Operations per pixel
[12] Memory usage refers to total amount of memory which is managed by the algorithm
[13] Theoretical value per pixel
[14] Computed with respect to previous version

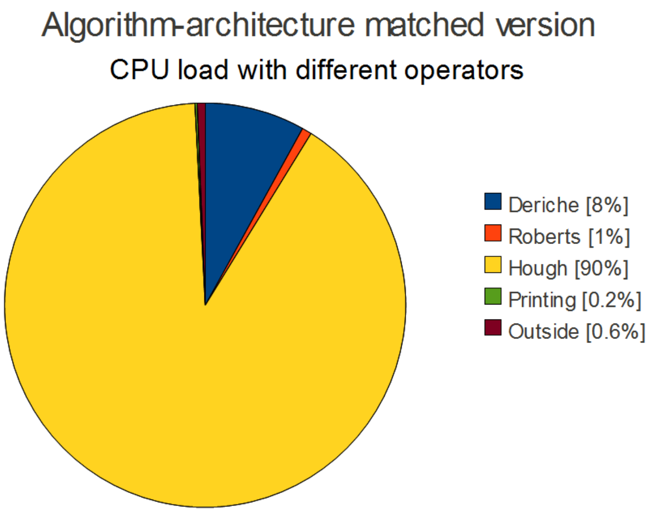

Figure 8: Algorithm-architecture matched version - CPU load by different operators

Summary of basic version shows most important indicators of this version's performance. It is resulting fps, total CPU load created by implemented image processing chain, amount of operations per pixel (oppx) and total memory usage.

Indicator	**Value**
FPS	0.49
OPPX	19886
CPU load	97.45%
Mem. usage	610.8kB
Speed gain	1.21

Table 5: Algorithm-architecture matched version summary

5.5 Conclusions

This version isn't much quicker than it's predecessor. We have managed to take advantage of DSP architecture and raised performance by 21%. ALthough it is a good result indeed, yet the problem is still the Hough transform - the algorithm is too exhaustive.

We should note that the algorithm-architecture matching was not complete, because we haven't moved computations from floating point to fixed point. This thing is asked to be done for the next version. We have big expectations, because it should allow the Deriche smoother optimizations (loop unrolling, software pipelining) to be more effective.

6 Algorithm-optimized version (version III)

In previous chapters we have seen that Hough transform algorithm is very exhaustive. We used floating point calculations on a fixed point CPU. We calculated values of $sin(\phi)$ and $cos(\phi)$ functions manually every call. This version optimizes all of that, changes algorithms and provides a real-time performance.

- Replace all floating point operations by fixed point
- Implement O'Gorman and Clowes version of Hough transform [2]
- Use LUT tables for trigonometric functions
- Use DMA for sending the images to the LCD screen
- Measure execution times of all used operators, profile this version and compare it to previous versions

6.1 Implementation notes

6.1.1 Image matrix represented in fixed point

When changing implementation to run completely in fixed point, first we had to change the datatype of image matrix from float (32 bit FP number) to integer format.

Our camera provides image in YUV color space [11]. Y component is color intensity called luminance, UV components are chrominances responsible for color information. Our camera encodes pixel to 16 bit number, where first 8 bit is Y component and next 2x4 bits are encoded as UV chrominance components. Since for B&W image we need only Y component, our image matrix can be encoded as a matrix of 8 bit unsigned integers.

By this conversion we have moved image matrix from floating point to fixed point, lowered memory occupation 4 times to 60kB, which allowed to move image matrix from DDR2 memory to internal SRAM (accessible part of L2 cache). Access to internal SRAM is approximately 4 times lower than external memory, SRAM is running on 594Mhz (CPU clock), while used DDR2 works on 166 Mhz.

6.1.2 Deriche filter towards fixed point

The deriche filter uses decimal constants for smoothing calculations. To make the filter working with unsigned char datatype, we had to manually convert result of floating point calculations to unsigned char. Converting resulting floats to integers worked properly with performance around 5 fps.

Thanks to this change, the Deriche filter was able to cooperate with unsigned char matrix, but inside it was still doing 12 multiplicatins and 8 summations per pixel in floating point calculated manually by processor. The problem were used floating point constants which are decimals.

The key is to multiply the constants by large number (1000 for example) and make them integer. With this approach we can store decimal number to up to three decimal places. For example number 1.2789 is FP, $1.2789 \cdot 10^3 = 1278.9$ is FP too, but after converstion to integer it becomes 1278. We can see that first three decimals are contained in the integer number. It is not a problem to do calculations all multiplied by 10^3, but drawback is that at some point we have to divide the result back. Division operation takes large amount of time and it is desirable to find a way around it.

It was said in chapter 2 that our CPU has a barrel shifter, so it can do bit shifts of arbitrary size in one clock cycle. It is also well known that bit shift to left by n places is the same as multiplication of the integer number by 2^n. Bit shift to right by n places divides the number by 2^n [15]. In other words, we can multiply/divide in one clock cycle by 1, 2, 8, 16, 32, 64, 128, 256, 512, 1024 and so on..

In our solution we have precalculated floating point constants for smoothing and then multiplied them by 1024 (equals to left bit shift by 10 places, 2^{10} equals 1024). With the new constants we can do all the calculations in fixed point. In the end, the results have to be bit-shifted by 10 places to right in order to get proper results.

Listing 10: Converting floating point to a fixed point

```
float g = exp(-alpha);
int ig1 = ((1-g)*(1-g))*1024;
int ig2 = (2*g)*1024;
int igg = (g*g)*1024;
...
// New fixed point calculations
p3 = (ig1*(matrix[ii++]) + ig2*p2 - igg*p1)>>10;
```

With this new approach the Deriche smoother manifests a performance of 234 fps, which is 40 times faster in comparison to it using the floating point calculations.

6.1.3 O'Gorman and Clowes version of Hough transform

The original version of Hough transform have $O(n^3)$ complexity. O'Gorman and Clowes proposed optimisation technique which reduces complexity to $O(n^2)$ [2]. The idea lies in approximating line direction from gradient information given by Deriche derivator. Instead of adding votes to all lines coming through a white point, we add a vote only to a line which is coming through that point with direction tangential to a gradient direction given by the Deriche derivator.

New algorithm fuses together Deriche derivator, binarisation and hough transform. For each point on the image we apply Roberts operator (posing as Deriche derivator), calculate magnitude of the edge and if it is above treshold, we add a vote to line in accumulator which is going through that point with direction tagential to the gradient.

[15] We consider MSB first

Listing 11: Optimized Hough transform implementation

```c
void hough_lines(int treshold) {
  /* Run-in-vars */ int k,j; float rho; int res, a;
  /* gradients */ int convy, convx; float conv ;
  register int p1,p2,p3,p4;

  #pragma MUST_ITERATE(360*180, 360*180, 1);
  /* clean the accumulator */ for(k=0; k<360*180;k++) h[k] = 0;
  #pragma MUST_ITERATE(height, height);
  for(k=0; k<height-1;k++) { // Lines
    /* pre-load data to registers */ p1 = matrix[k*width]; p4 = matrix[(k+1)*width];
    #pragma MUST_ITERATE(width, width);
    for(j=0; j<width-1;j++) { // Columns
     /* load next column */ p2 = mint[j+1+k*width]; p3 = mint[j+1+(k+1)*width];
     /* calculate gradients */ convy = -p1+p4-p2+p3; convx = -p1+p2-p4+p3;
     /* rotate registers */ p1 = p2; p4 = p3;
     /* binarisation */
     if((abs(convx) + abs(convy)) >= treshold) {
       /* calculate grad angle */ a = (int)atanf(convy/(float)convx);
       /* calculate point dist */ rho = j*cos(a) + k*sin(a);
       /* store to acc */ h[a+90+180*(((int)rho+360)>>1)]++;
    }
  }
}
```

Please note that we have used accumulator with ϕ from $-90°$ to $90°$ with coordinates encoded as $\phi = array\_index - 90$. For example angle $0°$ degrees equal to column 90 in the accumulator space. This is a new feature which was used because of the arctan() function. Arcus tangens is defined on interval $(-\infty, +\infty)$ and output range is $(\frac{-\pi}{2}; \frac{+\pi}{2})$ which is equivalent to $(-90°, +90°)$

6.1.4 Lookup tables instead of trigonometric functions

Trigonometric function sin(), cos() and arctan() from C standard library are calculating functions using a Taylor series. Taylor series is generally complicated equation which requires time to be processed. With calculative approach to trigonometric functions we can calculate result very precisely, disadvantage is time required for the calculation. Aim of this optimisation technique is to pre-calculate resulting values for appropriate argument interval and store the result to a lookup table (LUT).

Lookup table is an array which acts as a mapping of $\mathbb{N}^+ \mapsto \mathbb{R}$, where $\mathbb{N}^+$ is array index and $\mathbb{R}$ is the trigonometric function result. To get the $\mathbb{N}^+$ array index we need to do mapping of function input to array index. For that we have to choose how many function values we want to have on how long input interval.

We are using Hough accumulator space with angle precision of $1°$ (in other words $\phi \in \mathbb{Z}$ at interval $[-90°, 90°]$). This allows detected line angle precision of 1.7% at image size of 300×200 pixels. For full calculations see explanation below - diagonal (longest) line have offset 6 pixels between it's ends with maximum error of 1°. That makes 1.7% of the line length.

$$\begin{aligned}\sqrt{300^2 + 200^2}\sin(1) = 360\sin(1) = 6px \\ \frac{6px}{360px} = 1.7\%\end{aligned} \tag{19}$$

This precision is considered as good enough for both Hough transform and LUT tables. Used lookup table have to contain angles at interval $[-90°, 90°]$ for sin()/cos() and $(-\infty, +\infty)$ for arctan(). At this point we can use advantage of sin() properties: it is an odd function. It means that $\sin(-\phi) = -\sin(\phi)$. Similiarly cos() is an even function and therefore $\cos(-\phi) = \cos(\phi)$. With this knowledge we can redefine argument interval for both sin() and cos() as $[0°, 90°]$ and only setup rules to set sign of the output accordingly to the input. In case of sin() function the output has a same sign as an input. In case of cos() function the output doesn't change no matter positive or negative input. For used LUT implementation example of sin() and cos() see Listing 12.

Listing 12: Used implementation of LUT tables in floating point

```
float cosT[91]; float sinT[91];
float test; int a;

/* Initialize LUTs first */
for(int i=0;i<91;i++) {
  /* i - number of degrees */
  sinT[i] = sin(i*PI/180.0);
  cosT[i] = cos(i*PI/180.0);
}
/* Now test it */
a = 60; /* angle */
test = ((a<0)?-1:1)*sinT[60]; /* test = sin(60 ) = 0.866 */
test = cosT[abs(a)]; /* test = cos(60 ) = 0.5 */

a = -60;
test = ((a<0)?-1:1)*sinT[60]; /* test = sin(-60 ) = -0.866 */
test = cosT[abs(a)]; /* test = cos(60 ) = 0.5 */
```

Creating a LUT table for arctan() function is more complicated, because function is defined on whole $\mathbb{R}$ plane $(-\infty, +\infty)$. Additionally, the function changes dramatically functional values around zero. This let to implementation which uses two LUT tables, one with small step (0.1 between two records in a table) and high precision on interval from $[-10, 10]$ and second with long step (10 between two records in a table). In this way we can save memory while keeping high precision around zero input.

6.1.5 Using LUT tables in fixed point

At this point we have tried to solve already well known problem: FP operations. Functional values of all used trigonometric functions are decimal values, but the processor itself do not have support for floating point and hence it takes time to process them.

To solve this problem we used the same principle as with Deriche smoother. We wanted to preserve at least 4 decimal places to keep our calculations precise, so we decided to use bit shifting of 14 places which multiplies functional values by more than 10^4. As a consequence, the results of all calculations have to be divided by 16384 (2^{14}).

Listing 13: Used implementation of LUT tables in fixed point

```
int cosT[91]; int sinT[91];

/* initialize LUTs */
for(int i=0;i<91;i++) {
  /* i - number of degrees */
  sinT[i] = sin(i*PI/180.0)<<14;
  cosT[i] = cos(i*PI/180.0)<<14;
}

a = 60; /* angle */
test = (100*((a<0)?-1:1)*sinT[60])>>14;  /* sinT[60] = 0.866*16384 = 14188 */
                                         /* test = (14188*100)>>14 = 86 */
```

6.1.6 Utilizing DMA

We have decided not to utilize DMA channel. Our image matrix fits into DSP's internal SRAM memory and it's encoding is different from representation in camera/LCD driver. Conversion needs to be done anyway and therefore DMA utilisation is useless in our case.

6.2 Performance analysis

Performance was evaluated using approach described in chapter 2.4 Profiling techniques and 1.2 Measurement methodology. We are making 10 tests and use average values to calculate processign times, frames per second (fps), operations per pixel (oppx) and optimization gain. Memory usage and memory calls are evaluated from the source code.

Function name	**time [%]**	**time [ms]**	**fps**	**oppx**[16]	**mem. usage**[17]	**mem. acc.**[18]	**gain**[19]
Deriche smoother	23.72	18.97	52.70	187.85	241.2kB	4rd + 4wr	8.6
Deriche derivator Hough Transform	18.01	14.40	69.40	142.64	609.6kB	2rd + 1wr	127.83
Printing result	7.52	6.02	166.01	59.63	369.6kB	-	0.62

Table 6: Profiling results of algorithm optimized version (without compiler help)

[16] Operations per pixel
[17] Memory usage refers to total amount of memory which is managed by the algorithm
[18] Theoretical value per pixel
[19] Computed with respect to previous version

New version of Hough transform in combination with LUT tables and fixed point operations really gave it a boost. Specially new algorithm for Hough transform fused with Deriche derivator is over hundred times faster than it's predecessors. It is a great result and the key to get real-time performance out of the device. Algorithm manifests only 2 read accesses to count the gradient and 1 write access to give a vote in the Hough accumulator.

Deriche smoother shows improvement by almost 9 times. Algorithm is the same, only thing which changed is using of fixed point calculations and avoiding to division operations. Printing lines shows negative improvement, which is caused by image properties and by processing chain settings, as is explained in previous chapters. Significant part (50.74%) of processing time is outside image processing chain - it includes BIOS functions, transferring camera/screen[20] data and so on..

Function name	**time [%]**	**time [ms]**	**fps**	**oppx**[16]	**mem. usage**[17]	**mem. acc.**[18]	**gain**[19]
Deriche smoother	5.34	4.27	233.93	42.32	241.2kB	4rd + 4wr	4.44
Deriche derivator Hough Transform	13.51	10.81	92.52	106.99	609.6kB	2rd + 1wr	1.38
Printing result	7.01	5.60	178.35	55.50	369.6kB	-	1.07

Table 7: Profiling results of algorithm optimized version (with compiler help)

From Table 7 we can see that compiler managed to increase performance significantly. Version without compiler optimizations runs in real-time already, but this version reaches much further, almost to 50 fps. We don't know what compiler did to increase the performace, but now a majority of the processing time is outside our processing chain, which suggest that we are close to performance limit.

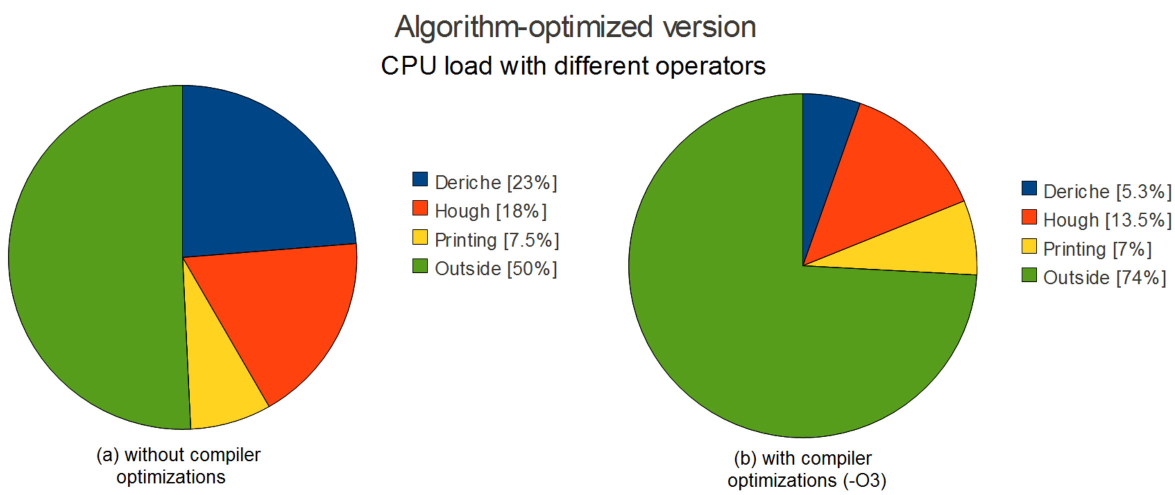

Figure 9: Algorithm-optimized version - CPU load by different operators

[20]Transfering data between peripherals and external memory is done by EDMA without CPU intervention.

Following summary shows most important performance indicators of this version. It is resulting fps, total CPU load created by image processing chain, amount of operations per pixel (oppx), memory usage and optimization gain between two consecutive versions.

Indicator	**Value**	**Optimized**
FPS	25.37	48.33
OPPX	390	204
CPU load	49.25%	25.86%
Mem. usage	610.8kB	610.8kB
Speed gain	50.98	1.9

Table 8: Summary of optimized version

6.3 Conclusions

Because of better algorithms, lookup tables instead of trigonometric calculations and fixed point operations we get more than real-time performance. Optimized version manifests almost 50 fps, which is twice the real-time requirements. This results exceeds our expectations and satisfies our requirements.

7 Conclusion

In a nutshell, we have created 3 versions where we applied algorithm and architecture related optimizations. At the end we managed to get result running over 100 times faster than the basic version. There is tremendous difference between first and last version performances. Hough transform got over 200 times faster, Deriche smoother manifests almosts 40 times faster behaviour without changing the algorithm. Deriche derivator got 4 times faster thanks to register rotation. See Table 9 for complete list of optimization gains between first and last version. Please note that the term "gain" refers to speed performance. To complete description how values in Table 9 were calculated, see chapter 1.2 Measurement methodology.

Operator	**Frames per second**		**Gain**
	Basic version	Optimized version	
Deriche smoother	6.06	233.93	38.6
Deriche derivator	12.39	92.53	209.27
Hough Transform	0.46		
Printing result	312.64	178.36	0.57
System	0.41	48.33	117.45

Table 9: Optimization gain

The very first version was running on a PC and we haven't really thought about optimization. Main goal was to write a starting point: working solution which can be easily ported to a DSP. When we did our first version on a DSP, the performance was so poor that the result was unusable in practice.

So, we decided to use the same algorithms and try to get advantage of the DSP capabilities. The results was algorithm-architecture matched version, which was faster by over 20% while keeping original algorithms and principles. Relatively speaking, it is a good result, but the performance was still quite poor. Partially, also because we haven't moved from floating point to fixed point with our calculations. This was requested for the very last version.

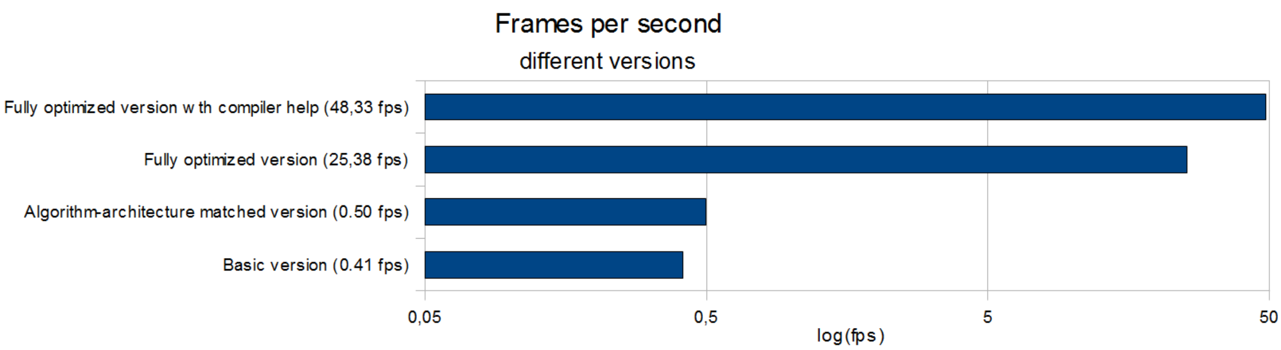

Figure 10: Compared FPS for different versions (on logarithmic scale)

The last version combines all optimization techniques known to us. Firstly, we have used $O(n^2)$ algorithm for Hough transform. Secondly, all calculations were moved from floating point to a fixed point and integer LUT tables replaced floating point trigonometric calculations. All together it created version over 50 times faster than previous. Additionally, compiler optimizations were able to double speed performance to almost 50 fps.

Personally, this project was a challenge for us. First of all used hardware equipment is very advanced. This was a first time we have got around parallel processing on instruction level. Then this was a first time we've been working on an embedded system related to image processing, which is for sure interesting topic for both of us. We have learned many things from this project, it is not only new programming and optimizing experience, we have also got around unexpected results - for example how big difference can relatively small amount of floating point operations do on performance, or how effective register rotations can be.

References

[1] Wikipedia (2012). *Canny edge detector* [online]. [Accessed 15 Jan 2012]. Available from: <http://en.wikipedia.org/wiki/Canny_edge_detector>.

[2] F. O'Gorman and M. B, Clowes. *Finding picture edges through colinearity of feature points* (1973) Available from: <http://www.ijcai.org/Past%20Proceedings/IJCAI-73/PDF/058.pdf>.

[3] Texas Instruments (2008). *TMS320DM6437 Digital Media Processor datasheet* [online]. [Accessed 15 Jan 2012]. Available from: <http://www.ti.com/lit/ds/symlink/tms320dm6437.pdf>.

[4] Texas Instruments (2008). *TMS320C64x/C64x+ DSP CPU & Instruction set. Reference Guide.* [online]. [Accessed 15 Jan 2012]. Available from:<http://www.ti.com/lit/ug/spru732j/spru732j.pdf >.

[5] Philips Semiconductors (2012). *An Introduction To Very-Long Instruction Word (VLIW) Computer Architecture* [online]. [Accessed 15 Jan 2012]. Available from: <http://embedded.cse.iitd.ernet.in/homepage/prjs/vliw_simu/tmvliw.pdf >.

[6] Texas Instruments (2008). *TMS320C64x/C64x+ DSP Cache. User's guide.* [online]. [Accessed 15 Jan 2012]. Available from: <http://www.ti.com/lit/ug/spru862b/spru862b.pdf>.

[7] Grandpierre, Thierry. *Architectures base de DSP pour le traitement des images* [online]. [Accessed 15 Jan 2012]. Available from: <https://extra.esiee.fr/ grandpit/IF4-ARCH-1-IntroDSP.pdf>.

[8] Texas Instruments (2008). *Real-Time DSP in Academia* [online]. [Accessed 15 Jan 2012]. Available from: <http://www.ti.com/europe/docs/univ/download/DSK.pdftextgreater.

[9] Dokladalova, Eva *Program Optimization Methodology* [online]. [Accessed 15 Jan 2012]. Available from: <https://extra.esiee.fr/ dokladae/cours_2011_2012_en.pdf>.

[10] Texas Instruments (2008). *TMS320C6000 Optimizing Compiler v 7.3* [online]. [Accessed 15 Jan 2012]. Available from: <http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=spru187&fileType=pdf>.

[11] Wikipedia (2012). *YUV* [online]. [Accessed 15 Jan 2012]. Available from: <http://en.wikipedia.org/wiki/YUV>.

8 Appendix A - Basic implementation on a PC

Listing 14: Used implementation of LUT tables in fixed point

```
/**************************************************************************/
/* Simple test program to exercise/demonstrate some functionality of IMPACT.
 *
 */
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <stdint.h>
#include <float.h>
#include "mcimage.h"
#include "mccodimage.h"

#define FOR_EACH_PIXEL(inc, data) for(inc=0; inc<(image->row_size*image->col_size);inc++) {
    data }

// it is a prototype, not a function
void processing(float *image,   // (IN) Matrix with source image
    float *result,  // (OUT) Matrix with resulting image
    unsigned int width,   // (IN) Image width
    unsigned int height,  // (IN) Image height
    int argc, char *argv[]);

// DO NOT MESS WITH THIS !!!
void store_me_fimage(char *filename, float *m, int width, int height)
{
  struct xvimage *result; int i, max = 0;
  result = allocimage(filename,height,width,1,1);

  for(i=0; i<(height*width); i++)
    if(max<*(m+i)) max=*(m+i);

  for(i=0; i<(height*width); i++)
    result->imagedata[i]=(unsigned char)(((*(m+i))/max)*255);

  writeimage(result, filename);
  freeimage (result);
}

// DO NOT MESS WITH THIS !!!
int main (int argc, char *argv[])
{
  // define image structure - PINK library
  struct xvimage * image; struct xvimage * result;
  float *fimage;float *fresult; int i; float max = 0;

  // read image PGM - ASCII
  image = readimage(argv[1]);  // lecture d'une image au format PGM !

  if (image == NULL)
    {
      fprintf(stderr, "addconst: readimage failed\n");
      exit(0);
    }

  // result image allocation
  result = allocimage("result",image->row_size,image->col_size,1,1);

  // allocate float matrixes
  fimage = malloc(sizeof(float)*image->row_size*image->col_size);
  fresult = malloc(sizeof(float)*image->row_size*image->col_size);

  // do the image processing with float matrices and progressive image colordepth
  FOR_EACH_PIXEL(i, { *(fimage+i)=(float)image->imagedata[i]; })
  processing (fimage, fresult, image->row_size, image->col_size, argc, argv);
  FOR_EACH_PIXEL(i, { if(max<*(fresult+i)) max=*(fresult+i); })
  FOR_EACH_PIXEL(i, { result->imagedata[i]=(unsigned char)(((*(fresult+i))/max)*255); })

  // write result image
```

```
  writeimage(result, argv[2]);
  freeimage (image);
  return (0);

}
```

Listing 15: project.c - image processing chain implementation

```
#include <math.h>
#include <stdlib.h>


#define gpx(x,y) m[x+(y)*width]
#define ADDR(in, x,y) in[(x) + (y)*m]
#define make_me_fimage(p, width, height) p = malloc(sizeof(float)*width*height)

#ifndef PI
#define PI 3.14159265
#endif

extern void store_me_fimage(char *filename, float *m, int width, int  height);


void deriche_gl(float *Id, float *Ic, float a, int m, int n) {

  float ga = exp(-a);
  float gb = 1-ga;
  float g1 = (1-a)*(1-a);
  float g2 = 2*a;
  float gg = g*a;

  int i,j;
  float Ir[m*n];

  // Horizontal filtering
  for(i = 0; i < n; i++) {
    // Filter causal
    ADDR(Ic,0,i) = ADDR(Id,0,i);
    ADDR(Ic,1,i) = ADDR(Id,1,i);

    for(j = 2; j < m; j++) {
      ADDR(Ic,j,i) = (g1*ADDR(Id,j,i)+g2*ADDR(Ic,j-1,i)+gg*ADDR(Ic,j-2,i))/2;
    }

    // Filter anticausal
    ADDR(Ir,m-1,i) = ADDR(Id,m-1,i);
    ADDR(Ir,m-2,i) = ADDR(Id,m-2,i);

    for(j = m-3; j >= 0 ; j--) {
      ADDR(Ir,j,i) = (g1*ADDR(Ic,j,i)+(g2*ADDR(Ir,j+1,i))+(gg*ADDR(Ir,j+2,i)))/2;
    }
  }


  // Vertical filtering
  for(i = 0; i < m; i++) {
    // Filter causal
    ADDR(Ic,i,0) = ADDR(Ir,i,0);
    ADDR(Ic,i,1) = ADDR(Ir,i,1);

    for(j = 2; j < n; j++) {
      ADDR(Ic,i,j) = (g1*ADDR(Ir,i,j)+(g2*ADDR(Ic,i,j-1))+(gg*ADDR(Ic,i,j-2)))/2;
    }

    // Filter anticausal
    ADDR(Ir,i,n-1) = ADDR(Ic,i,n-1);
    ADDR(Ir,i,n-2) = ADDR(Ic,i,n-2);

    for(j = n-3; j >= 0 ; j--) {
      ADDR(Ir,i,j) = (g1*ADDR(Ic,i,j)+(g2*ADDR(Ir,i,j+1))+(gg*ADDR(Ir,i,j+2)))/2;
    }
  }
```

```
}

void roberts(float *m, float *mout, float treshold, unsigned int width, unsigned int height)
{
  float convy, convx; int j, k;
  for(k=0; k<height-1;k++) // Lines
    for(j=0; j<width-1;j++) { // Columns
      //ROBERTSON
      convx = -gpx(j,k)-gpx(j+1,k)+gpx(j,k+1)+gpx(j+1,k+1);
      convy = -gpx(j,k)+gpx(j+1,k)-gpx(j,k+1)+gpx(j+1,k+1);
      //SOBEL
      //convy = -gpx(j-1,k-1)-2*gpx(j,k-1)-gpx(j+1,k-1)+gpx(j-1,k+1)+2*gpx(j,k+1)+gpx(j+1,k+1);
      //convx = -gpx(j-1,k-1)-2*gpx(j-1,k)-gpx(j-1,k+1)+gpx(j+1,k-1)+2*gpx(j+1,k)+gpx(j+1,k+1);
      if (convy<0) convy = -convy; if (convx<0) convx = -convx; // abs. val. done manually
      mout[j+k*width] = (convy+convx >= treshold) ? 255 : 0;
    }
}

float * hough_lines_slow(float *m, unsigned int width, unsigned int height) {
  /* Run-in-vars */ int k,j, a; float rho; int mrho = (int)sqrt(width*width + height*height);
  /* Make the image */ float *h = malloc(sizeof(float)*mrho*180);
  for(k=1; k<height-1;k++) // Lines
    for(j=1; j<width-1;j++) // Columns
      if(m[j + k*width]) for(a=0;a<180;a++) { // Angles
    rho = (j*cos(a*PI/180.0) + k*sin(a*PI/180.0));
    if(rho) h[a+180*(((int)rho+mrho)/2)]++; }
  return h;
}

void print_line(float *m, float intensity, int x1, int y1, int x2, int y2, unsigned int width, unsigned int height)
{
  int i; int d = (int)sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)); /* line size */
  float ax=(x2-x1)/(float)d, ay=(y2-y1)/(float)d; /* direction vectors */
  for(i=0; i<d; i++) /* print pixel */  m[(x1+(int)(ax*i))+(y1+(int)(ay*i))*width] = intensity;
}

void print_lines(float*m, float *h, unsigned int treshold, unsigned int width, unsigned int height)
{
  int j,k, x[2], y[2], hct; int mrho = (int)sqrt(width*width + height*height);
  int sx=0, sy=0, ex=sx+width, ey=sy+height, is1_x, is2_y, is3_x, is4_y;
  float x1, y1, ax, ay, rho;

  for(k=0; k<mrho;k++) { rho = k*2-mrho;
    for(j=0; j<180;j++) if(h[j+180*k] > treshold) {  hct = 0;
  x1 = rho*cos(j*PI/180.0); y1 = rho*sin(j*PI/180.0); /* get a point on line */
  ax = y1/(float)j; ay = -x1/(float)j; /* get direction vector */
        is1_x = x1-sx+(sy-y1)*ax/ay; is2_y = y1-sy+(ex-x1)*ay/ax; /* calculate intersections */
  is3_x = x1-sx+(ey-y1)*ax/ay; is4_y = y1-sy+(sx-x1)*ay/ax;
  if((is1_x>=sx)&&(is1_x<ex)) { x[hct]=is1_x; y[hct++]=sy; } /* find hits */
  if((is3_x>=sx)&&(is3_x<ex)) { x[hct]=is3_x; y[hct++]=ey; }
  if((is2_y>=sy)&&(is2_y<ey)) { x[hct]=ex; y[hct++]=is2_y; }
  if((is4_y>=sy)&&(is4_y<ey)) { x[hct]=sx; y[hct++]=is4_y; }
  if(hct==2) print_line(m, 255, x[0]-sx, y[0]-sy, x[1]-sx, y[1]-sy, width, height); /* print */
      }
  }
}

void processing(float *m,   // (IN) Matrix with source image
    float *mout,  // (OUT) Matrix with resulting image
    unsigned int width,  // (IN) Image width
    unsigned int height,  // (IN) Image height
    int argc, char *argv[])
{

  float *h; float dercf, edgecf, trshcf; int j;
```

```

  sscanf(argv[3], "%f", &dercf);
  sscanf(argv[4], "%f", &edgecf);
  sscanf(argv[5], "%f", &trshcf);

  deriche_gl(m, mout, dercf , width, height);
  store_me_fimage("deriche.pgm", mout, height, width);
  roberts(mout, mout, edgecf, width, height);
  store_me_fimage("robertson.pgm", mout, height, width)
  h = hough_lines_slow(mout, mout, edgecf, width, height);;
  store_me_fimage("hough.pgm", h, sqrt(width*width+height*height), 180);
  print_lines(mout, h, trshcf, width, height);

}
```

Listing 16: testme.sh - script used to speedup development process

```
if [ $# = 5 ]; then
  rm hough.pgm deriche.pgm robertson.pgm
  gcc -g -lm main.c mcimage.c project.c -o r.out
  if [ $? = 0 ]; then
    ./r.out $1 $2 $3 $4 $5
    display $1 deriche.pgm robertson.pgm hough.pgm $2
  fi
else
  echo "usage ./testme imgin.pgm imgout.pgm deriche_coef edge_binarisation_coef
      tresholding_coef"
fi
```

Listing 17: profile-prj.sh - script used to profile program performance

```
rm a.out
rm gmon.out

gcc -pg -lm main.c mcimage.c project.c -o a.out

./a.out test.pgm test2.pgm 0.8 160 120


rm gmon.sum
mv gmon.out gmon.sum

gprof -s a.out gmon.out gmon.sum
./a.out test.pgm test2.pgm 0.8 160 120
gprof -s a.out gmon.out gmon.sum
./a.out test.pgm test2.pgm 0.8 160 120
gprof -s a.out gmon.out gmon.sum
./a.out test.pgm test2.pgm 0.8 160 120
gprof -s a.out gmon.out gmon.sum
./a.out test.pgm test2.pgm 0.8 160 120
gprof -s a.out gmon.out gmon.sum
./a.out test.pgm test2.pgm 0.8 160 120
gprof -s a.out gmon.out gmon.sum
./a.out test.pgm test2.pgm 0.8 160 120
gprof -s a.out gmon.out gmon.sum

gprof a.out gmon.sum > $1
```

Listing 18: result-prof.txt - profiling results

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 93.75      0.15     0.15        7    21.43    21.43  hough_lines_slow
  6.25      0.16     0.01                             main
  0.00      0.16     0.00      154     0.00     0.00  print_line
  0.00      0.16     0.00       35     0.00     0.00  allocimage
```

```
  0.00      0.16     0.00       28     0.00     0.00  freeimage
  0.00      0.16     0.00       28     0.00     0.00  writeimage
  0.00      0.16     0.00       28     0.00     0.00  writerawimage
  0.00      0.16     0.00       21     0.00     0.00  store_me_fimage
  0.00      0.16     0.00        7     0.00     0.00  deriche_gl
  0.00      0.16     0.00        7     0.00     0.00  print_lines
  0.00      0.16     0.00        7     0.00    21.43  processing
  0.00      0.16     0.00        7     0.00     0.00  readimage
  0.00      0.16     0.00        7     0.00     0.00  roberts

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.


         Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 6.25% of 0.16 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    0.15                 main [1]
                0.00    0.15       7/7           processing [3]
                0.00    0.00       7/28          freeimage [6]
                0.00    0.00       7/28          writeimage [7]
                0.00    0.00       7/35          allocimage [5]
                0.00    0.00       7/7           readimage [12]
-----------------------------------------------
                0.15    0.00       7/7           processing [3]
[2]     93.8    0.15    0.00       7         hough_lines_slow [2]
-----------------------------------------------
                0.00    0.15       7/7           main [1]
[3]     93.8    0.00    0.15       7         processing [3]
                0.15    0.00       7/7           hough_lines_slow [2]
                0.00    0.00      21/21          store_me_fimage [9]
                0.00    0.00       7/7           print_lines [11]
                0.00    0.00       7/7           roberts [13]
                0.00    0.00       7/7           deriche_gl [10]
-----------------------------------------------
                0.00    0.00     154/154         print_lines [11]
[4]      0.0    0.00    0.00     154         print_line [4]
-----------------------------------------------
                0.00    0.00       7/35          main [1]
                0.00    0.00       7/35          readimage [12]
                0.00    0.00      21/35          store_me_fimage [9]
[5]      0.0    0.00    0.00      35         allocimage [5]
-----------------------------------------------
                0.00    0.00       7/28          main [1]
                0.00    0.00      21/28          store_me_fimage [9]
```

```
[6]       0.0    0.00    0.00      28         freeimage [6]
-----------------------------------------------
                 0.00    0.00       7/28          main [1]
                 0.00    0.00      21/28          store_me_fimage [9]
[7]       0.0    0.00    0.00      28         writeimage [7]
                 0.00    0.00      28/28          writerawimage [8]
-----------------------------------------------
                 0.00    0.00      28/28          writeimage [7]
[8]       0.0    0.00    0.00      28         writerawimage [8]
-----------------------------------------------
                 0.00    0.00      21/21          processing [3]
[9]       0.0    0.00    0.00      21         store_me_fimage [9]
                 0.00    0.00      21/28          freeimage [6]
                 0.00    0.00      21/28          writeimage [7]
                 0.00    0.00      21/35          allocimage [5]
-----------------------------------------------
                 0.00    0.00       7/7           processing [3]
[10]      0.0    0.00    0.00       7         deriche_gl [10]
-----------------------------------------------
                 0.00    0.00       7/7           processing [3]
[11]      0.0    0.00    0.00       7         print_lines [11]
                 0.00    0.00     154/154         print_line [4]
-----------------------------------------------
                 0.00    0.00       7/7           main [1]
[12]      0.0    0.00    0.00       7         readimage [12]
                 0.00    0.00       7/35          allocimage [5]
-----------------------------------------------
                 0.00    0.00       7/7           processing [3]
[13]      0.0    0.00    0.00       7         roberts [13]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each entry in this table consists of several lines.  The line with the
 index number at the left hand margin lists the current function.
 The lines above it list the functions that called this function,
 and the lines below it list the functions this one called.
 This line lists:
     index  A unique number given to each element of the table.
    Index numbers are sorted numerically.
    The index number is printed next to every function name so
    it is easier to look up where the function in the table.

     % time This is the percentage of the `total' time that was spent
    in this function and its children.  Note that due to
    different viewpoints, functions excluded by options, etc,
    these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children This is the total amount of time propagated into this
    function by its children.

     called This is the number of times the function was called.
    If the function called itself recursively, the number
    only includes non-recursive calls, and is followed by
    a `+' and the number of recursive calls.

     name The name of the current function.  The index number is
    printed after it.  If the function is a member of a
    cycle, the cycle number is printed between the
    function's name and the index number.


 For the function's parents, the fields have the following meanings:

     self This is the amount of time that was propagated directly
    from the function into this parent.

     children This is the amount of time that was propagated from
    the function's children into this parent.

```

```
     called This is the number of times this parent called the
    function '/' the total number of times the function
    was called.  Recursive calls to the function are not
    included in the number after the '/'.

     name This is the name of the parent.  The parent's index
    number is printed after it.  If the parent is a
    member of a cycle, the cycle number is printed between
    the name and the index number.

If the parents of the function cannot be determined, the word
'<spontaneous>' is printed in the 'name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

     self This is the amount of time that was propagated directly
    from the child into the function.

     children This is the amount of time that was propagated from the
    child's children to the function.

     called This is the number of times the function called
    this child '/' the total number of times the child
    was called.  Recursive calls by the child are not
    listed in the number after the '/'.

     name This is the name of the child.  The child's index
    number is printed after it.  If the child is a
    member of a cycle, the cycle number is printed
    between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole.  This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The '+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.


Index by function name

   [5] allocimage              [4] print_line              [9] store_me_fimage
  [10] deriche_gl             [11] print_lines             [7] writeimage
   [6] freeimage               [3] processing              [8] writerawimage
   [2] hough_lines_slow       [12] readimage
   [1] main                   [13] roberts
```

9 Appendix B - Basic implementation on a DSP

Listing 19: Basic implementation on a DSP

```
/**
 Completely naive version of implementing given image processing chain.

Authors: Divij Babbar, Kubicka Matej (I4-IMC)
Date: 1/6/2012
Version: Naive

Properties:
-----------
sin() LUT           NOT USED
cos() LUT           NOT USED
atan() LUT          NOT USED
hough version       STANDARD
matrix datatype     FLOAT
loop unrolling      NOT USED

*/

#include <std.h>
#include <gio.h>
#include <log.h>
#include <math.h>

#include "psp_vpfe.h"
#include "psp_vpbe.h"
#include "fvid.h"

#include "psp_tvp5146_extVidDecoder.h"

#include <soc.h>
#include <cslr_ccdc.h>

#include <soc.h>
#include <cslr_sysctl.h>


//pour logger ce qui se passe avec log.h (voir DSPBIOS)
extern LOG_Obj trace;  // BIOS LOG object


/* extrait de l'exemple EDMA3 :
// 48K L1 SRAM [0x10f04000, 0x10f10000), 0xc000 length
// 32K L1 Dcache [0x10f10000, 0x10f18000), 0x8000 length
// 128K L2 SRAM [0x10800000, 0x10820000), 0x20000 length
// 128M DDR2 [0x80000000, 0x88000000), 0x8000000 length are cacheable
*/

#define width 300
#define height 200
#define ADDR(in, x,y) in[(x) + (y)*width]
#define gpx(x,y) m[x+(y)*width]

#ifndef PI
#define PI 3.14159265
#endif

#define NO_OF_BUFFERS       (2u)
#define width 300
#define height 200

// Global Variable Defined
static PSP_VPSSSurfaceParams *ccdcAllocFB[NO_OF_BUFFERS]={NULL};
static PSP_VPSSSurfaceParams *vidAllocFB[NO_OF_BUFFERS] ={NULL};

static FVID_Handle   ccdcHandle;
static FVID_Handle   vid0Handle;
static FVID_Handle   vencHandle;

```

```
static PSP_VPFE_TVP5146_ConfigParams tvp5146Params = {
  TRUE, // enable656Sync
  PSP_VPFE_TVP5146_FORMAT_COMPOSITE, // format
  PSP_VPFE_TVP5146_MODE_AUTO         // mode
};

static PSP_VPFECcdcConfigParams ccdcParams = {
  PSP_VPFE_CCDC_YCBCR_8,  // dataFlow
  PSP_VPSS_FRAME_MODE,    // ffMode
  480,                    // height
  720,                    // width
  (720 *2),               // pitch
  0,                      // horzStartPix
  0,                      // vertStartPix
  NULL,                   // appCallback
  {
    PSP_VPFE_TVP5146_Open, // extVD Fxn
    PSP_VPFE_TVP5146_Close,
    PSP_VPFE_TVP5146_Control,
  },
  0                       //segId
};

static PSP_VPBEOsdConfigParams  vid0Params = {
  PSP_VPSS_FRAME_MODE,                  // ffmode
  PSP_VPSS_BITS16,                      // bitsPerPixel
  //PSP_VPBE_RGB_888,  //ajout TG
 PSP_VPBE_YCbCr422,                    // colorFormat
  (720 *  (16/8u)),                     // pitch
  0,                                    // leftMargin
  0,                                    // topMargin
  720,                                  // width
  480,                                  // height
  0,                                    // segId
  PSP_VPBE_ZOOM_IDENTITY,               // hScaling
  PSP_VPBE_ZOOM_IDENTITY,               // vScaling
  PSP_VPBE_EXP_IDENTITY,                // hExpansion
  PSP_VPBE_EXP_IDENTITY,                // vExpansion
  NULL                                  // appCallback
};

static PSP_VPBEVencConfigParams vencParams = {
  PSP_VPBE_DISPLAY_NTSC_INTERLACED_COMPOSITE // Display Standard
};

#pragma DATA_SECTION(m, ".ExtBuffer")
float m[width*height];

#pragma DATA_SECTION(m2, ".ExtBuffer")
float m2[width*height];

#pragma DATA_SECTION(h, ".ExtBuffer")
short h[360*180];

unsigned int profiling[50];



void robertson(float treshold)
{
  float convy, convx; int j, k;
  for(k=0; k<height-1;k++) // Lines
    for(j=0; j<width-1;j++) { // Columns
      //ROBERTSON
     convx = -gpx(j,k)-gpx(j+1,k)+gpx(j,k+1)+gpx(j+1,k+1);
     convy = -gpx(j,k)+gpx(j+1,k)-gpx(j,k+1)+gpx(j+1,k+1);
     if (convy<0) convy = -convy; if (convx<0) convx = -convx; // abs. val. done manually
     m2[j+k*width] = (convy+convx >= treshold) ? 255 : 0;
    }
}


void transpose(float *in, float *out, int w, int h)
```

```c
{
  int n,m,s=0;
  for(n=0; n<h;n++)
    for(m=0; m<w;m++)
          out[m*h+n] = in[s++];
}

float l[width];
float la[width*height];
float lb[width*height];
float lc[width*height];
float ld[width*height];

void deriche_gl(float g)
{
        float g1 = (1-g)*(1-g);
        float g2 = 2*g;
        float gg = g*g;

        int i,ii,k;

        for(i = 0 ; i < width ; i++){ // lines
                ii=i*height;

                la[ii+0]=g1*m2[ii];
                la[ii+1]=(g1*m2[1]+g2*la[ii+0]);
                for(k=2;k<height;k++){
                    la[ii+k] = (g1*m2[ii+k] + g2*la[ii+k-1] - gg*la[ii+k-2]);
                }

                lb[ii+height-1]=la[ii+height-1];
                lb[ii+height-2]=la[ii+height-2];
                for(k=height-3;k>=0;k--){
                    lb[ii+k] = (g1*la[ii+k] + g2*lb[ii+k+1] - gg*lb[ii+k+2]);
                }
        }

        transpose(lb, lc, height, width);

        for(i = 0 ; i < height ; i++){ // lines
                ii=i*width;
                ld[ii+0]=g1*lc[ii+0];
                ld[ii+1]=(g1*lc[ii+1]+g2*ld[ii+0]);
                for(k=2;k<width;k++){
                    ld[ii+k] = (g1*lc[ii+k] + g2*ld[ii+k-1] - gg*ld[ii+k-2]);
                }

                m[ii+width-1]=ld[ii+width-1];
                m[ii+width-2]=ld[ii+width-2];
                for(k=width-3;k>=0;k--){
                    m[ii+k] = (g1*ld[ii+k] + g2*m[ii+k+1] - gg*m[ii+k+2]);
                }
        }

}

void print_line(float intensity, int x1, int y1, int x2, int y2)
{
  int i; int d = (int)sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)); /* line size */
  float ax=(x2-x1)/(float)d, ay=(y2-y1)/(float)d; /* direction vectors */
  for(i=0; i<d; i++) /* print pixel */  m[(x1+(int)(ax*i))+(y1+(int)(ay*i))*width] =
      intensity;
}

void print_lines_slow(unsigned int treshold)
{
  int j,k, x[2], y[2], hct;
  int sx=0, sy=0, ex=sx+width, ey=sy+height, is1_x, is2_y, is3_x, is4_y;
  float x1, y1, ax, ay, rho;

  for(k=0; k<360;k++) { rho = k*2-360;
    for(j=0; j<180;j++)
      if(h[j+180*k] > treshold) {
```

```
        hct = 0;
        x1 = rho*cos(j*PI/180.0); y1 = rho*sin(j*PI/180.0); /* get a point on line */
      ax = y1/(float)j; ay = -x1/(float)j; /* get direction vector */
          is1_x = x1-sx+(sy-y1)*ax/ay; is2_y = y1-sy+(ex-x1)*ay/ax; /* calculate intersections */
      is3_x = x1-sx+(ey-y1)*ax/ay; is4_y = y1-sy+(sx-x1)*ay/ax;
      if((is1_x>=sx)&&(is1_x<ex
      )) { x[hct]=is1_x; y[hct++]=sy; } /* find hits */
      if((is3_x>=sx)&&(is3_x<ex)) { x[hct]=is3_x; y[hct++]=ey; }
      if((is2_y>=sy)&&(is2_y<ey)) { x[hct]=ex; y[hct++]=is2_y; }
      if((is4_y>=sy)&&(is4_y<ey)) { x[hct]=sx; y[hct++]=is4_y; }
      if(hct==2) print_line(255, x[0]-sx, y[0]-sy, x[1]-sx, y[1]-sy); /* print */
      }
  }
}

void hough_lines_slow() {
  /* Run-in-vars */ int k,j, a; float rho;

  for(k=0; k<360*180;k++)
  h[k] = 0;

  for(k=1; k<height;k++) // Lines
    for(j=1; j<width;j++) // Columns
      if(m2[j + k*width]) for(a=0;a<180;a++) { // Angles
    rho = (j*cos(a*PI/180.0) + k*sin(a*PI/180.0));
    //rho = (j*cosT[a] + k*sinT[a]);
    if(rho) h[a+180*(((int)rho+360)/2)]++; }
}

void start_boucle() {
  PSP_VPBEChannelParams beinitParams;
  PSP_VPFEChannelParams feinitParams;
  GIO_Attrs gioAttrs = GIO_ATTRS;
  PSP_VPSSSurfaceParams *FBAddr = NULL;
  PSP_VPSSSurfaceParams *FBAddrOut = NULL;
  int i,v = 0;
  Uint32 j = 0;
  Uint32 k = 0;

  //Init CSL du DMA
  edma3init();

  // Create ccdc channel
  feinitParams.id = PSP_VPFE_CCDC;
  feinitParams.params = (PSP_VPFECcdcConfigParams*)&ccdcParams;
  ccdcHandle = FVID_create( "/VPFE0", IOM_INOUT, NULL, &feinitParams,
                            &gioAttrs);
  if ( NULL == ccdcHandle) {
    return;
  }

  // Configure the TVP5146 video decoder
  if( FVID_control( ccdcHandle,
                    VPFE_ExtVD_BASE + PSP_VPSS_EXT_VIDEO_DECODER_CONFIG,
                    &tvp5146Params) != IOM_COMPLETED ) {
        return;
  } else {
    for ( i=0; i < NO_OF_BUFFERS; i++ ) {
      if ( IOM_COMPLETED == FVID_alloc( ccdcHandle, &ccdcAllocFB[i] ) ) {
        if ( IOM_COMPLETED != FVID_queue(ccdcHandle, ccdcAllocFB[i] ) ) {
          return;
        }
      }
    }
  }

  // Create video channel
  beinitParams.id = PSP_VPBE_VIDEO_0;
  beinitParams.params = (PSP_VPBEOsdConfigParams*)&vid0Params;
  vid0Handle = FVID_create( "/VPBE0", IOM_INOUT,NULL, &beinitParams,
                            &gioAttrs );
  if ( NULL == vid0Handle ) {
```

```c
    return;
  } else {
    for ( i=0; i<NO_OF_BUFFERS; i++ )  {
      if ( IOM_COMPLETED == FVID_alloc( vid0Handle, &vidAllocFB[i] ) ) {
        if ( IOM_COMPLETED != FVID_queue( vid0Handle, vidAllocFB[i]) ) {
          return;
        }
      }
    }
  }

  // Create venc channel
  beinitParams.id = PSP_VPBE_VENC;
  beinitParams.params = (PSP_VPBEVencConfigParams *)&vencParams;
  vencHandle = FVID_create( "/VPBE0", IOM_INOUT, NULL, &beinitParams,
                            &gioAttrs);
  if ( NULL == vencHandle ) {
    return;
  }

  //Allocation memoire et la structure qui contiendra l'image
  FVID_alloc( ccdcHandle, &FBAddr );
  FVID_alloc( ccdcHandle, &FBAddrOut );


  //================BOUCLE ACQUISITION & COPIE & AFFICHAGE DESIMAGES====================
  // 1)Acquisition
  for( i = 0; i < 10000; i++ ) {

        // Load image
    if ( IOM_COMPLETED != FVID_exchange( ccdcHandle, &FBAddr ) ) {
         return;
    }


  v = i%10;

  // Make the Y matrix transposed
  for(k=0; k<height;k++) // Lines
    for(j=0; j<width;j++)   // Columns
      m2[k+j*height] = (float)(*((unsigned char *)FBAddr->frameBufferPtr + (j*2 + k*2*720)*2
           + 1));


  // integer
  profiling[5*v] = C64P_getltime();
  deriche_gl(0.2);
  profiling[5*v+1] = C64P_getltime();
  robertson(30);
  profiling[5*v+2] = C64P_getltime();
  hough_lines_slow();
  profiling[5*v+3] = C64P_getltime();
  print_lines_slow(150);
  profiling[5*v+4] = C64P_getltime();

  // Print the Y matrix
  for(k=0; k<height;k++) // Lines
    for(j=0; j<width;j++) { // Columns
      *((unsigned char *)FBAddrOut->frameBufferPtr + (j + k*720)*2) = 128;
      *((unsigned char *)FBAddrOut->frameBufferPtr + (j + k*720)*2 + 1) = (unsigned char)m[j
          +width*k];
  }

    LOG_printf( &trace, "    Affichage iteration = %u", i );

        // Print changed image
    if ( IOM_COMPLETED != FVID_exchange( vid0Handle, &FBAddrOut) ) {
      return;
    }
  }
  //================FIN BOUCLE ACQUISITION & COPIE & AFFICHAGE DESIMAGES
      ======================
  FVID_free(vid0Handle,  FBAddr);
```

```
  FVID_free(ccdcHandle,  FBAddrOut);

  // Free Memory Buffers
  for( i=0; i< NO_OF_BUFFERS; i++ ) {
    FVID_free( ccdcHandle, ccdcAllocFB[i] );
    FVID_free( vid0Handle, vidAllocFB[i] );
  }

  // Delete Channels
  FVID_delete( ccdcHandle );
  FVID_delete( vid0Handle );
  FVID_delete( vencHandle );

  return;
}
```

10 Appendix C - Algorithm-architecture matched version

Listing 20: Algorithm-architecture matched version

```
/**
 Algorithm-architecture matched version of given image processing chain.

Authors: Divij Babbar, Kubicka Matej (I4-IMC)
Date: 1/6/2012
Version: algorithm-architecture matched
*/

#include <std.h>
#include <gio.h>
#include <log.h>
#include <math.h>

#include "psp_vpfe.h"
#include "psp_vpbe.h"
#include "fvid.h"

#include "psp_tvp5146_extVidDecoder.h"

#include <soc.h>
#include <cslr_ccdc.h>

#include <soc.h>
#include <cslr_sysctl.h>


//pour logger ce qui se passe avec log.h (voir DSPBIOS)
extern LOG_Obj trace;  // BIOS LOG object


/* extrait de l'exemple EDMA3 :
// 48K L1 SRAM [0x10f04000, 0x10f10000), 0xc000 length
// 32K L1 Dcache [0x10f10000, 0x10f18000), 0x8000 length
// 128K L2 SRAM [0x10800000, 0x10820000), 0x20000 length
// 128M DDR2 [0x80000000, 0x88000000), 0x8000000 length are cacheable
*/


#define ADDR(in, x,y) in[(x) + (y)*width]
#define absM(v) (v<0)?(-v):(v)

#ifndef PI
#define PI 3.14159265
#endif

#define NO_OF_BUFFERS       (2u)
#define width 300
#define height 200


// Global Variable Defined
static PSP_VPSSSurfaceParams *ccdcAllocFB[NO_OF_BUFFERS]={NULL};
static PSP_VPSSSurfaceParams *vidAllocFB[NO_OF_BUFFERS] ={NULL};

static FVID_Handle   ccdcHandle;
static FVID_Handle   vid0Handle;
static FVID_Handle   vencHandle;

static PSP_VPFE_TVP5146_ConfigParams tvp5146Params = {
  TRUE, // enable656Sync
  PSP_VPFE_TVP5146_FORMAT_COMPOSITE, // format
  PSP_VPFE_TVP5146_MODE_AUTO         // mode
};

static PSP_VPFECcdcConfigParams ccdcParams = {
  PSP_VPFE_CCDC_YCBCR_8,  // dataFlow
  PSP_VPSS_FRAME_MODE,    // ffMode
  480,                    // height
```

```
  720,                    // width
  (720 *2),               // pitch
  0,                      // horzStartPix
  0,                      // vertStartPix
  NULL,                   // appCallback
  {
    PSP_VPFE_TVP5146_Open, // extVD Fxn
    PSP_VPFE_TVP5146_Close,
    PSP_VPFE_TVP5146_Control,
  },
  0                       //segId
};

static PSP_VPBEOsdConfigParams  vid0Params = {
  PSP_VPSS_FRAME_MODE,                // ffmode
  PSP_VPSS_BITS16,                    // bitsPerPixel
  //PSP_VPBE_RGB_888,  //ajout TG
 PSP_VPBE_YCbCr422,                  // colorFormat
  (720 *  (16/8u)),                   // pitch
  0,                                  // leftMargin
  0,                                  // topMargin
  720,                                // width
  480,                                // height
  0,                                  // segId
  PSP_VPBE_ZOOM_IDENTITY,             // hScaling
  PSP_VPBE_ZOOM_IDENTITY,             // vScaling
  PSP_VPBE_EXP_IDENTITY,              // hExpansion
  PSP_VPBE_EXP_IDENTITY,              // vExpansion
  NULL                                // appCallback
};

static PSP_VPBEVencConfigParams vencParams = {
  PSP_VPBE_DISPLAY_NTSC_INTERLACED_COMPOSITE // Display Standard
};

#pragma DATA_SECTION(mint, ".ExtBuffer")
float mint[300*200];
#pragma DATA_SECTION(mint2, ".ExtBuffer")
float mint2[300*200];
#pragma DATA_SECTION(h, ".ExtBuffer")
short h[360*180];

unsigned int profiling[50];

float sinT[360];
float cosT[360];
float arctanT[4000];

#pragma DATA_SECTION(l, ".L2Buffer")
float l[width];

void transpose(float *in, float *out, int w, int h)
{
  int n,m,s=0;
  for(n=0; n<h;n++)
    for(m=0; m<w;m++)
          out[m*h+n] = in[s++];
}

void deriche2(float g)
{
  float ig1 = ((1-g)*(1-g));
  float ig2 = (2*g);
  float igg = (g*g);

  int i,ii,k;
  float p1, p2, p3;

  #pragma MUST_ITERATE(width, width);
  for(i = 0 ; i < width ; i++){ // lines
    ii=i*height;
    p3=ig1*(mint2[ii++]);
    p2=(ig1*mint2[ii++]+ig2*l[0]);
```

```
    l[0]=p3; l[1]=p2;
    #pragma MUST_ITERATE(198, 198,1);
    for(k=2;k<height;k++){

      p1 = p2;
      p2 = p3;
      p3 = (ig1*(mint2[ii++]) + ig2*(p2) - igg*(p1));
      l[k] = p3;
    }

    mint2[ii--]=p3;
    mint2[ii--]=p2;
    #pragma MUST_ITERATE(198, 198,1);
    for(k=height-3;k>=0;k--){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(l[k]) + ig2*(p2) - igg*(p1));
      mint2[ii--] = (unsigned char)p3;
    }
  }

  transpose(mint2, mint, height, width);

  #pragma MUST_ITERATE(height, height);
  for(i = 0 ; i < height ; i++){ // lines
    ii=i*width;

    p3=(ig1*mint[ii++]);
    p2=(ig1*mint[ii++]+ig2*l[0]);
    l[0]=p3; l[1]=p2;
    #pragma MUST_ITERATE(298, 298,1);
    for(k=2;k<width;k++){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(mint[ii++]) + ig2*(p2) - igg*(p1));
      l[k] = p3;
    }

    mint[ii--]=p3;
    mint[ii--]=p2;
    #pragma MUST_ITERATE(298, 298,1);
    for(k=width-3;k>=0;k--){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(l[k]) + ig2*(p2) - igg*(p1));
      mint[ii--] = (unsigned char) p3;
    }
  }
}

void print_line(float intensity, int x1, int y1, int x2, int y2)
{
  int i; int d = (int)sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)); /* line size */
  float ax=(x2-x1)/(float)d, ay=(y2-y1)/(float)d; /* direction vectors */
  for(i=0; i<d; i++) /* print pixel */  mint[(x1+(int)(ax*i))+(y1+(int)(ay*i))*width] =
      intensity;
}

void print_lines(unsigned int treshold)
{
  int j,jk,k, x[2], y[2], hct;
  int sx=0, sy=0, ex=sx+width, ey=sy+height, is1_x, is2_y, is3_x, is4_y;
  float x1, y1, ax, ay, rho;

  #pragma MUST_ITERATE(360, 360, 1);
  for(k=0; k<360;k++) { rho = k*2-360;
    #pragma MUST_ITERATE(180, 180);
    #pragma UNROLL(20);
    for(jk=0; jk<180;jk++)
      if(h[jk+180*k] > treshold) {
        j=jk-90;
        hct = 0;
        x1 = rho*cosT[absM(j)]; y1 = ((j<0)?-1:1)*rho*sinT[absM(j)]; /* get a point on line
```

```
          */
      ax = y1; ay = -x1; /* get direction vector */
      is1_x = x1-sx+(sy-y1)*ax/ay; is2_y = y1-sy+(ex-x1)*ay/ax; /* calculate intersections */
      is3_x = x1-sx+(ey-y1)*ax/ay; is4_y = y1-sy+(sx-x1)*ay/ax;
      if((is1_x>=sx)&&(is1_x<ex)) { x[hct]=is1_x; y[hct++]=sy; } /* find hits */
      if((is3_x>=sx)&&(is3_x<ex)) { x[hct]=is3_x; y[hct++]=ey; }
      if((is2_y>=sy)&&(is2_y<ey)) { x[hct]=ex; y[hct++]=is2_y; }
      if((is4_y>=sy)&&(is4_y<ey)) { x[hct]=sx; y[hct++]=is4_y; }
      if(hct==2) print_line(255, x[0], y[0], x[1], y[1]); /* print */
    }
  }
}

void roberts(int treshold) {
   /* Run-in-vars */ int k,j;
   int convy, convx; int res;
   register int p1,p2,p3,p4;

   #pragma MUST_ITERATE(height, height);
   for(k=0; k<height-1;k++) { // Lines
     p1 = mint[k*width];
     p4 = mint[(k+1)*width];
     #pragma MUST_ITERATE(width, width);
     for(j=0; j<width-1;j++) { // Columns
       //ROBERTS
       p2 = mint[j+1+k*width];
       p3 = mint[j+1+(k+1)*width];
       convy = -p1+p4-p2+p3;
       convx = -p1+p2-p4+p3;
       p1 = p2; p4 = p3;
       res = ((absM(convx) + absM(convy)) >= treshold)?255:0;
       mint2[j+k*width]=res;
     }
   }
}

void hough_lines_slow() {
  /* Run-in-vars */ int k,j, a; float rho;

  for(k=0; k<360*180;k++)
  h[k] = 0;

  for(k=1; k<height;k++) // Lines
    for(j=1; j<width;j++) // Columns
      if(mint2[j + k*width]) for(a=0;a<180;a++) { // Angles
        rho = (j*cos(a*PI/180.0) + k*sin(a*PI/180.0));
        if(rho) h[a+180*(((int)rho+360)/2)]++;
      }
}

void start_boucle() {
  PSP_VPBEChannelParams beinitParams;
  PSP_VPFEChannelParams feinitParams;
  GIO_Attrs gioAttrs = GIO_ATTRS;
  PSP_VPSSSurfaceParams *FBAddr = NULL;
  PSP_VPSSSurfaceParams *FBAddrOut = NULL;

  Uint32 j = 0;
  Uint32 k = 0;
  Uint32 l = 0;

  int i,v = 0;

  for(i=0;i<359;i++) {
  sinT[i] = sin(i*PI/180.0);
    cosT[i] = cos(i*PI/180.0);
  }

  for(i=-2000;i<2000;i++)
  arctanT[i+2000] = atan(i/10.0);

  //Init CSL du DMA
```

```
edma3init();

// Create ccdc channel
feinitParams.id = PSP_VPFE_CCDC;
feinitParams.params = (PSP_VPFECcdcConfigParams*)&ccdcParams;
ccdcHandle = FVID_create( "/VPFE0", IOM_INOUT, NULL, &feinitParams,
                          &gioAttrs);
if ( NULL == ccdcHandle) {
  return;
}

// Configure the TVP5146 video decoder
if( FVID_control( ccdcHandle,
                  VPFE_ExtVD_BASE + PSP_VPSS_EXT_VIDEO_DECODER_CONFIG,
                  &tvp5146Params) != IOM_COMPLETED ) {
      return;
} else {
  for ( i=0; i < NO_OF_BUFFERS; i++ ) {
    if ( IOM_COMPLETED == FVID_alloc( ccdcHandle, &ccdcAllocFB[i] ) ) {
      if ( IOM_COMPLETED != FVID_queue(ccdcHandle, ccdcAllocFB[i] ) ) {
        return;
      }
    }
  }
}

// Create video channel
beinitParams.id = PSP_VPBE_VIDEO_0;
beinitParams.params = (PSP_VPBEOsdConfigParams*)&vid0Params;
vid0Handle = FVID_create( "/VPBE0", IOM_INOUT,NULL, &beinitParams,
                          &gioAttrs );
if ( NULL == vid0Handle ) {
  return;
} else {
  for ( i=0; i<NO_OF_BUFFERS; i++ )  {
    if ( IOM_COMPLETED == FVID_alloc( vid0Handle, &vidAllocFB[i] ) ) {
      if ( IOM_COMPLETED != FVID_queue( vid0Handle, vidAllocFB[i]) ) {
        return;
      }
    }
  }
}

// Create venc channel
beinitParams.id = PSP_VPBE_VENC;
beinitParams.params = (PSP_VPBEVencConfigParams *)&vencParams;
vencHandle = FVID_create( "/VPBE0", IOM_INOUT, NULL, &beinitParams,
                          &gioAttrs);
if ( NULL == vencHandle ) {
  return;
}

//Allocation memoire et la structure qui contiendra l'image
FVID_alloc( ccdcHandle, &FBAddr );
FVID_alloc( ccdcHandle, &FBAddrOut );


//================BOUCLE ACQUISITION & COPIE & AFFICHAGE DESIMAGES============
// 1)Acquisition
for( i = 0; i < 1000000; i++ ) {

      // Load image
  if ( IOM_COMPLETED != FVID_exchange( ccdcHandle, &FBAddr ) ) {
       return;
  }

v = i%10;

// Make the Y matrix transposed
l=0; for(k=0; k<height;k++) // Lines
  for(j=0; j<width;j++)  // Columns
    mint2[k+j*height] = (float)(*((unsigned char *)FBAddr->frameBufferPtr +
      (j*2 + k*2*720)*2 + 1));
```

```

  // integer
  profiling[5*v] = C64P_getltime();
  deriche2(0.2);
  profiling[5*v+1] = C64P_getltime();
  roberts(75);
  profiling[5*v+2] = C64P_getltime();
  hough_lines_slow();
  profiling[5*v+3] = C64P_getltime();
  print_lines(100);
  profiling[5*v+4] = C64P_getltime();


  // Print the Y matrix
  l=0; for(k=0; k<height;k++) // Lines
    for(j=0; j<width;j++) { // Columns
      *((unsigned char *)FBAddrOut->frameBufferPtr + (j + k*720)*2) = 128;
      *((unsigned char *)FBAddrOut->frameBufferPtr + (j + k*720)*2 + 1) =
              (unsigned char)(mint[l]);
      l++;
  }

    LOG_printf( &trace, "    Affichage iteration = %u", i );

        // Print changed image
    if ( IOM_COMPLETED != FVID_exchange( vid0Handle, &FBAddrOut) ) {
      return;
    }
  }
  //================FIN BOUCLE ACQUISITION & COPIE & AFFICHAGE DESIMAGES=======
  FVID_free(vid0Handle,  FBAddr);
  FVID_free(ccdcHandle,  FBAddrOut);

  // Free Memory Buffers
  for( i=0; i< NO_OF_BUFFERS; i++ ) {
    FVID_free( ccdcHandle, ccdcAllocFB[i] );
    FVID_free( vid0Handle, vidAllocFB[i] );
  }

  // Delete Channels
  FVID_delete( ccdcHandle );
  FVID_delete( vid0Handle );
  FVID_delete( vencHandle );

  return;
}
```

11 Appendix D - Optimized version

Listing 21: Optimized version

```
/**
 Optimized version of given image processing chain.

Authors: Divij Babbar, Kubicka Matej (I4-IMC)
Date: 1/6/2012
Version: Naive

*/

#include <std.h>
#include <gio.h>
#include <log.h>
#include <math.h>

#include "psp_vpfe.h"
#include "psp_vpbe.h"
#include "fvid.h"

#include "psp_tvp5146_extVidDecoder.h"

#include <soc.h>
#include <cslr_ccdc.h>

#include <soc.h>
#include <cslr_sysctl.h>


//pour logger ce qui se passe avec log.h (voir DSPBIOS)
extern LOG_Obj trace;  // BIOS LOG object


/* extrait de l'exemple EDMA3 :
// 48K L1 SRAM [0x10f04000, 0x10f10000), 0xc000 length
// 32K L1 Dcache [0x10f10000, 0x10f18000), 0x8000 length
// 128K L2 SRAM [0x10800000, 0x10820000), 0x20000 length
// 128M DDR2 [0x80000000, 0x88000000), 0x8000000 length are cacheable
*/


#define ADDR(in, x,y) in[(x) + (y)*width]
#define absM(v) (v<0)?(-v):(v)

#ifndef PI
#define PI 3.14159265
#endif

#define NO_OF_BUFFERS       (2u)
#define width 300
#define height 200


// Global Variable Defined
static PSP_VPSSSurfaceParams *ccdcAllocFB[NO_OF_BUFFERS]={NULL};
static PSP_VPSSSurfaceParams *vidAllocFB[NO_OF_BUFFERS] ={NULL};

static FVID_Handle   ccdcHandle;
static FVID_Handle   vid0Handle;
static FVID_Handle   vencHandle;

static PSP_VPFE_TVP5146_ConfigParams tvp5146Params = {
  TRUE, // enable656Sync
  PSP_VPFE_TVP5146_FORMAT_COMPOSITE, // format
  PSP_VPFE_TVP5146_MODE_AUTO         // mode
};

static PSP_VPFECcdcConfigParams ccdcParams = {
  PSP_VPFE_CCDC_YCBCR_8,  // dataFlow
  PSP_VPSS_FRAME_MODE,    // ffMode
```

```
  480,                      // height
  720,                      // width
  (720 *2),                 // pitch
  0,                        // horzStartPix
  0,                        // vertStartPix
  NULL,                     // appCallback
  {
    PSP_VPFE_TVP5146_Open, // extVD Fxn
    PSP_VPFE_TVP5146_Close,
    PSP_VPFE_TVP5146_Control,
  },
  0                         //segId
};

static PSP_VPBEOsdConfigParams  vid0Params = {
  PSP_VPSS_FRAME_MODE,                  // ffmode
  PSP_VPSS_BITS16,                      // bitsPerPixel
  //PSP_VPBE_RGB_888,  //ajout TG
 PSP_VPBE_YCbCr422,                    // colorFormat
  (720 *  (16/8u)),                     // pitch
  0,                                    // leftMargin
  0,                                    // topMargin
  720,                                  // width
  480,                                  // height
  0,                                    // segId
  PSP_VPBE_ZOOM_IDENTITY,               // hScaling
  PSP_VPBE_ZOOM_IDENTITY,               // vScaling
  PSP_VPBE_EXP_IDENTITY,                // hExpansion
  PSP_VPBE_EXP_IDENTITY,                // vExpansion
  NULL                                  // appCallback
};

static PSP_VPBEVencConfigParams vencParams = {
  PSP_VPBE_DISPLAY_NTSC_INTERLACED_COMPOSITE // Display Standard
};

#pragma DATA_SECTION(mint, ".L2Buffer")
unsigned char mint[300*200];
#pragma DATA_SECTION(mint2, ".ExtBuffer")
unsigned char mint2[300*200];
#pragma DATA_SECTION(h, ".ExtBuffer")
short h[360*180];

unsigned int profiling[50];

float sinT[360];
float cosT[360];
float arctanT[4000];

#pragma DATA_SECTION(l, ".L2Buffer")
int l[width];

void transpose(unsigned char *in, unsigned char *out, int w, int h)
{
  int n,m,s=0;
  for(n=0; n<h;n++)
    for(m=0; m<w;m++)
          out[m*h+n] = in[s++];
}

void deriche2(float g)
{
  int ig1 = ((1-g)*(1-g))*1024;
  int ig2 = (2*g)*1024;
  int igg = (g*g)*1024;

  int i,ii,k;
  int p1, p2, p3;

  #pragma MUST_ITERATE(width, width);
  for(i = 0 ; i < width ; i++){ // lines
    ii=i*height;
    p3=ig1*(mint2[ii++])>>10;
```

```
    p2=(ig1*mint2[ii++]+ig2*l[0])>>10;
    l[0]=p3; l[1]=p2;
    #pragma MUST_ITERATE(198, 198,1);
    for(k=2;k<height;k++){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(mint2[ii++]) + ig2*(p2) - igg*(p1))>>10;
      l[k] = p3;
    }

    mint2[ii--]=p3;
    mint2[ii--]=p2;
    #pragma MUST_ITERATE(198, 198,1);
    for(k=height-3;k>=0;k--){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(l[k]) + ig2*(p2) - igg*(p1))>>10;
      mint2[ii--] = (unsigned char)p3;
    }
  }

  transpose(mint2, mint, height, width);

  #pragma MUST_ITERATE(height, height);
  for(i = 0 ; i < height ; i++){ // lines
    ii=i*width;
    p3=(ig1*mint[ii++])>>10;
    p2=(ig1*mint[ii++]+ig2*l[0])>>10;
    l[0]=p3; l[1]=p2;
    #pragma MUST_ITERATE(298, 298,1);
    for(k=2;k<width;k++){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(mint[ii++]) + ig2*(p2) - igg*(p1))>>10;
      l[k] = p3;
    }

    mint[ii--]=p3;
    mint[ii--]=p2;
    #pragma MUST_ITERATE(298, 298,1);
    for(k=width-3;k>=0;k--){
      p1 = p2;
      p2 = p3;
      p3 = (ig1*(l[k]) + ig2*(p2) - igg*(p1))>>10;
      mint[ii--] = (unsigned char) p3;
    }
  }
}

void print_line(float intensity, int x1, int y1, int x2, int y2)
{
  int i; int d = (int)sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)); /* line size */
  float ax=(x2-x1)/(float)d, ay=(y2-y1)/(float)d; /* direction vectors */
  for(i=0; i<d; i++) /* print pixel */  mint[(x1+(int)(ax*i))+(y1+(int)(ay*i))*width] =
      intensity;
}

void print_lines(unsigned int treshold)
{
  int j,jk,k, x[2], y[2], hct;
  int sx=0, sy=0, ex=sx+width, ey=sy+height, is1_x, is2_y, is3_x, is4_y;
  float x1, y1, ax, ay, rho;

  #pragma MUST_ITERATE(360, 360, 1);
  for(k=0; k<360;k++) { rho = k*2-360;
    #pragma MUST_ITERATE(180, 180);
  #pragma UNROLL(20);
    for(jk=0; jk<180;jk++)
      if(h[jk+180*k] > treshold) {
        j=jk-90; hct = 0;
        x1 = rho*cosT[absM(j)]; y1 = ((j<0)?-1:1)*rho*sinT[absM(j)]; /* get a point on line
            */
        ax = y1; ay = -x1; /* get direction vector */
```

```
        is1_x = x1-sx+(sy-y1)*ax/ay; is2_y = y1-sy+(ex-x1)*ay/ax; /* calculate intersections */
        is3_x = x1-sx+(ey-y1)*ax/ay; is4_y = y1-sy+(sx-x1)*ay/ax;
        if((is1_x>=sx)&&(is1_x<ex)) { x[hct]=is1_x; y[hct++]=sy; } /* find hits */
        if((is3_x>=sx)&&(is3_x<ex)) { x[hct]=is3_x; y[hct++]=ey; }
        if((is2_y>=sy)&&(is2_y<ey)) { x[hct]=ex; y[hct++]=is2_y; }
        if((is4_y>=sy)&&(is4_y<ey)) { x[hct]=sx; y[hct++]=is4_y; }
        if(hct==2) print_line(255, x[0], y[0], x[1], y[1]); /* print */
      }
  }
}

void hough_lines(int treshold) {
   /* Run-in-vars */ int k,j; float rho;
   int convy, convx; float conv ; int res, a;
   register int p1,p2,p3,p4;
   int v;

   #pragma MUST_ITERATE(360*180, 360*180, 1);
   for(k=0; k<360*180;k++)
     h[k] = 0;

   #pragma MUST_ITERATE(height, height);
   for(k=0; k<height-1;k++) { // Lines
     p1 = mint[k*width];
     p4 = mint[(k+1)*width];
     #pragma MUST_ITERATE(width, width);
     for(j=0; j<width-1;j++) { // Columns

      //ROBERTS
      p2 = mint[j+1+k*width];
      p3 = mint[j+1+(k+1)*width];
      convy = -p1+p4-p2+p3;
      convx = -p1+p2-p4+p3;
      p1 = p2; p4 = p3;
      res = ((absM(convx) + absM(convy)) >= treshold)?255:0;

      if(res!=0) {
        conv = (convy/(float)convx);
        if(conv >= 200) a = 90;
        else if(conv < -200) a=-90;
        else a = (int)(arctanT[(int)(conv*10)+2000]*57.295);
        rho = (j*cosT[absM(a)] + ((a<0)?-1:1)*k*sinT[absM(a)]);
        if(rho) h[a+90+180*(((int)rho+360)>>1)]++;
      }
    }
  }
}

void start_boucle() {
  PSP_VPBEChannelParams beinitParams;
  PSP_VPFEChannelParams feinitParams;
  GIO_Attrs gioAttrs = GIO_ATTRS;
  PSP_VPSSSurfaceParams *FBAddr = NULL;
  PSP_VPSSSurfaceParams *FBAddrOut = NULL;

  Uint32 j = 0;
  Uint32 k = 0;
  Uint32 l = 0;


  int i,v = 0;

  for(i=0;i<359;i++) {
  sinT[i] = sin(i*PI/180.0);
    cosT[i] = cos(i*PI/180.0);
  }

  for(i=-2000;i<2000;i++)
  arctanT[i+2000] = atan(i/10.0);

  //Init CSL du DMA
  edma3init();
```

```
// Create ccdc channel
feinitParams.id = PSP_VPFE_CCDC;
feinitParams.params = (PSP_VPFECcdcConfigParams*)&ccdcParams;
ccdcHandle = FVID_create( "/VPFE0", IOM_INOUT, NULL, &feinitParams,
                          &gioAttrs);
if ( NULL == ccdcHandle) {
  return;
}

// Configure the TVP5146 video decoder
if( FVID_control( ccdcHandle,
                  VPFE_ExtVD_BASE + PSP_VPSS_EXT_VIDEO_DECODER_CONFIG,
                  &tvp5146Params) != IOM_COMPLETED ) {
      return;
} else {
  for ( i=0; i < NO_OF_BUFFERS; i++ ) {
    if ( IOM_COMPLETED == FVID_alloc( ccdcHandle, &ccdcAllocFB[i] ) ) {
      if ( IOM_COMPLETED != FVID_queue(ccdcHandle, ccdcAllocFB[i] ) ) {
        return;
      }
    }
  }
}

// Create video channel
beinitParams.id = PSP_VPBE_VIDEO_0;
beinitParams.params = (PSP_VPBEOsdConfigParams*)&vid0Params;
vid0Handle = FVID_create( "/VPBE0", IOM_INOUT,NULL, &beinitParams,
                          &gioAttrs );
if ( NULL == vid0Handle ) {
  return;
} else {
  for ( i=0; i<NO_OF_BUFFERS; i++ )  {
    if ( IOM_COMPLETED == FVID_alloc( vid0Handle, &vidAllocFB[i] ) ) {
      if ( IOM_COMPLETED != FVID_queue( vid0Handle, vidAllocFB[i]) ) {
        return;
      }
    }
  }
}

// Create venc channel
beinitParams.id = PSP_VPBE_VENC;
beinitParams.params = (PSP_VPBEVencConfigParams *)&vencParams;
vencHandle = FVID_create( "/VPBE0", IOM_INOUT, NULL, &beinitParams,
                          &gioAttrs);
if ( NULL == vencHandle ) {
  return;
}

//Allocation memoire et la structure qui contiendra l'image
FVID_alloc( ccdcHandle, &FBAddr );
FVID_alloc( ccdcHandle, &FBAddrOut );


//================BOUCLE ACQUISITION & COPIE & AFFICHAGE DESIMAGES============
// 1)Acquisition
for( i = 0; i < 1000000; i++ ) {

        // Load image
  if ( IOM_COMPLETED != FVID_exchange( ccdcHandle, &FBAddr ) ) {
    return;
  }

  v = i%10;

  // Make the Y matrix transposed
  l=0; for(k=0; k<height;k++) // Lines
    for(j=0; j<width;j++)  // Columns
      mint2[k+j*height] = (unsigned char)(*((unsigned char *)FBAddr->frameBufferPtr
        + (j*2 + k*2*720)*2 + 1));

```

```
    // integer
    profiling[5*v] = C64P_getltime();
    deriche2(0.2);
    profiling[5*v+1] = C64P_getltime();
    profiling[5*v+2] = C64P_getltime();
    hough_lines(10);
    profiling[5*v+3] = C64P_getltime();
    print_lines(20);
    profiling[5*v+4] = C64P_getltime();


    // Print the Y matrix
    l=0; for(k=0; k<height;k++) // Lines
      for(j=0; j<width;j++) { // Columns
        *((unsigned char *)FBAddrOut->frameBufferPtr + (j + k*720)*2) = 128;
        *((unsigned char *)FBAddrOut->frameBufferPtr + (j + k*720)*2 + 1) =
                (unsigned char)(mint[l]);
        l++;
    }

    LOG_printf( &trace, "    Affichage iteration = %u", i );

        // Print changed image
    if ( IOM_COMPLETED != FVID_exchange( vid0Handle, &FBAddrOut) ) {
      return;
    }
  }
  //================FIN BOUCLE ACQUISITION & COPIE & AFFICHAGE DESIMAGES==========
  FVID_free(vid0Handle,  FBAddr);
  FVID_free(ccdcHandle,  FBAddrOut);

  // Free Memory Buffers
  for( i=0; i< NO_OF_BUFFERS; i++ ) {
    FVID_free( ccdcHandle, ccdcAllocFB[i] );
    FVID_free( vid0Handle, vidAllocFB[i] );
  }

  // Delete Channels
  FVID_delete( ccdcHandle );
  FVID_delete( vid0Handle );
  FVID_delete( vencHandle );

  return;
}
```