Edge detection optimization

December 17, 2023

Adrien Rey

1 Introduction

For the Embbeded Hardware project, we have to optimize the edge detection on a Nios II. A working project was given but the performance are not good. The given project has a perfomance of 8450 cycles per pixels. By following the theory seen during the course, the goal is to improve the performance of the system.

2 Software optimization

2.1 Compiler optimization

By modifing the compiler optimization, the performance is improved without changing the code :

- **With -O0 : 8450 cycles/pixel**
- **With -O1 : 1570 cycles/pixel**
- **With -O2 : 1337 cycles/pixel**
- **With -O3 : 545 cycles/pixel**

2.2 Sobel

2.2.1 Loop unrolling

By unrolling the inner and outer loop of the *sobel_mac* function the performance is improve. The loop was unroll 9 times so it no need loop anymore. The function has the following form.

```
short sobel_mac( unsigned char *pixels,
                 int y,
                 const char *filter,
                 unsigned int width ) {
   short dy,dx;
   short result = 0;

   //dy = -1
   result += filter[0]*
             pixels[(y-1)*width+(x-1)];
   result += filter[1]*
             pixels[(y-1)*width+x];
   result += filter[2]*
             pixels[(y-1)*width+(x+1)];
   //dy = 0
   result += filter[3]*
             pixels[y*width+(x-1)];
   result += filter[4]*
             pixels[y*width+x];
   result += filter[5]*
             pixels[y*width+(x+1)];
   //dy = 1
   result += filter[6]*
             pixels[(y+1)*width+(x-1)];
   result += filter[7]*
             pixels[(y+1)*width+x];
```

```
    result += filter[8]*
              pixels[(y+1)*width+(x+1)];

    return result;
}
```

With -O0 : 5606 cycles/pixel

2.2.2 In-lining

The function *sobel_mac* is integrated directly in the *sobel_x* and the *sobel_y* function as shown in the code snippet below. With this optimization, the performance is much improved.

```
void sobel_x(unsigned char *source) {
    int x, y;

    for (y = 1; y < (sobel_height - 1); y++) {
        for (x = 1; x < (sobel_width - 1); x++) {
            sobel_x_result[y * sobel_width + x] =
                currentRow[x - 1 - sobel_width] * gx_array[0][0] +
                currentRow[x - sobel_width] * gx_array[0][1] +
                currentRow[x + 1 - sobel_width] * gx_array[0][2] +
                currentRow[x - 1] * gx_array[1][0] +
                currentRow[x] * gx_array[1][1] +
                currentRow[x + 1] * gx_array[1][2] +
                currentRow[x - 1 + sobel_width] * gx_array[2][0] +
                currentRow[x + sobel_width] * gx_array[2][1] +
                currentRow[x + 1 + sobel_width] * gx_array[2][2];
        }
    }
}
```

With -O0 : 3855 cycles/pixel

With -O3 : 630 cycles/pixel

2.2.3 Pointer

To further improve the performance, pointers were used to access datas and write new values as in the following code.

```
void sobel_x(unsigned char *source) {
    int x, y;

    for (y = 1; y < (sobel_height - 1); y++) {
        for (x = 1; x < (sobel_width - 1); x++) {
            unsigned char *currentRow = source + y * sobel_width;

            currentRow[y * sobel_width + x] =
                currentRow[x - 1 - sobel_width] * gx_array[0][0] +
                currentRow[x - sobel_width] * gx_array[0][1] +
                currentRow[x + 1 - sobel_width] * gx_array[0][2] +
                currentRow[x - 1] * gx_array[1][0] +
                currentRow[x] * gx_array[1][1] +
                currentRow[x + 1] * gx_array[1][2] +
                currentRow[x - 1 + sobel_width] * gx_array[2][0] +
                currentRow[x + sobel_width] * gx_array[2][1] +
                currentRow[x + 1 + sobel_width] * gx_array[2][2];
        }
```

```
  }
}
```

With -O3 : 545 cycles/pixel

2.2.4 Complete method

All the logic for the sobel was reunited a method called *sobel*. The contain of the *sobel_x* and *sobel_y* are embbeded in this function. The code from the threshold is also included. With this, the double for loops from the *sobel_threshold*, *sobel_x* and *sobel-y* are reuinited in one loop. The performance are better as the number of loops decrease. In this part, local variable are use to store the x and y value. So the array is acces only one time at the end for both x and y. The threshold is simplified by the use of abs. It do not change the perfomance but it improved the code lisibility

```c
void sobel(unsigned char *source, short threshold) {
  int x, y;

  for (y = 1; y < (sobel_height - 1); y++) {
    for (x = 1; x < (sobel_width - 1); x++) {
      int arrayIndex = y * sobel_width + x;
      unsigned char *currentRow = source + y * sobel_width;

      short gx = currentRow[x - 1 - sobel_width] * gx_array[0][0] +
                 currentRow[x - sobel_width] * gx_array[0][1] +
                 currentRow[x + 1 - sobel_width] * gx_array[0][2] +
                 currentRow[x - 1] * gx_array[1][0] +
                 currentRow[x] * gx_array[1][1] +
                 currentRow[x + 1] * gx_array[1][2] +
                 currentRow[x - 1 + sobel_width] * gx_array[2][0] +
                 currentRow[x + sobel_width] * gx_array[2][1] +
                 currentRow[x + 1 + sobel_width] * gx_array[2][2];

      short gy = currentRow[x - 1 - sobel_width] * gy_array[0][0] +
                 currentRow[x - sobel_width] * gy_array[0][1] +
                 currentRow[x + 1 - sobel_width] * gy_array[0][2] +
                 currentRow[x - 1] * gy_array[1][0] +
                 currentRow[x] * gy_array[1][1] +
                 currentRow[x + 1] * gy_array[1][2] +
                 currentRow[x - 1 + sobel_width] * gy_array[2][0] +
                 currentRow[x + sobel_width] * gy_array[2][1] +
                 currentRow[x + 1 + sobel_width] * gy_array[2][2];

      short sum = abs(gx) + abs(gy);

      sobel_result[arrayIndex] = (sum > threshold) ? 0xFF : 0;
    }
  }
}
```

With -O3 : 375 cycles/pixel

2.3 Grayscale

The grayscale function has also be improved by simplifing the equation. It has be reunited in a simgle line. The division by 100 has been replaced by a right shift of 128. To keep the same value at the end, the color weights have been updated following this rule new_value $= \frac{128}{100} * \text{value}$.

```
void conv_grayscale(void *picture,
                int width,
                int height) {
  int x,y,gray;
  unsigned short *pixels = (unsigned short *)picture , rgb;
  grayscale_width = width;
  grayscape_height = height;
  if (grayscale_array != NULL)
    free(grayscale_array);
  grayscale_array = (unsigned char *) malloc(width*height);
  for (y = 0 ; y < height ; y++) {
    for (x = 0 ; x < width ; x++) {
      rgb = pixels[y*width+x];

      gray = ((((rgb>>11)&0x1F)<<3)*27 +
          (((rgb>>5)&0x3F)<<2)*92 +
          (((rgb>>0)&0x1F)<<3)*9)
          >> 7;

      IOWR_8DIRECT(grayscale_array,y*width+x,gray);
    }
   }
}
```

With -O3 : 334 cycles/pixel

3 Cache

To optimize the memory use, the data and instruction caches have been activated with a size of 16k-bytes each. It made a huge difference to activate the cache.

With -O3 : 140 cycles/pixel

Tests have also been made with 32k-bytes but the performances were the same. We can conclude that the 16k-bytes caches are optimized for our system. It avoid using to much power consumption.

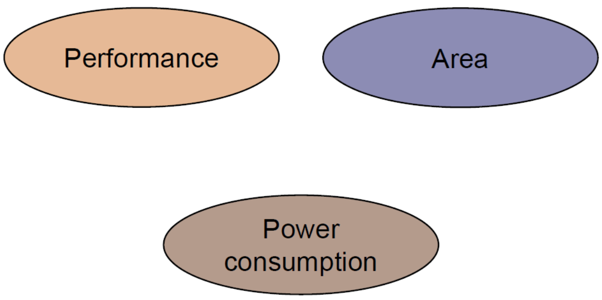

Figure 1: Performance - Area - Power consumption

4 Custom instructions

A custom instruction was writed and used for the threshold. It allow to make the threshold calul in a only cycle. The vhdl code is the following.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity sobel_threshold_ci is
  port (
    signal sum          : in std_logic_vector(31 downto 0);
    signal threshold    : in std_logic_vector(31 downto 0);
    signal thresholdResult : out std_logic_vector(31 downto 0)
  );
end sobel_threshold_ci;

architecture behavioral of sobel_threshold_ci is
begin
  process(sum)
  begin
    if sum > threshold then
        thresholdResult <= (others=>'1');
    else
        thresholdResult <= (others=>'0');
    end if;
  end process;
end behavioral;
```

In the *sobel* function, this line of code

```
 sobel_result[arrayIndex] = (sum > threshold) ? 0xFF : 0;
```

is replaced by this one using the macro created along with the new composant based on the vhdl added in Qsys

```
 sobel_result[arrayIndex] = (sum > threshold) ? 0xFF : 0;
sobel_result[arrayIndex] = ALT_CI_THRESHOLD_CI_0(sum, threshold);
```

🚀 **With -O3 : 134 cycles/pixel**

5 Memory access

The last improvements were to optimiz the use of the cache memory. To do so, in the *sobel* function, instead of using a pointer to access the data, we made local copy of each of the three lines used for the calcul. We are using a *memcpy* to copy with the cache the contain of the array into the local array.

```
void sobel(unsigned char *source, short threshold) {
    int x, y;

    for (y = 1; y < (sobel_height - 1); y++) {
        unsigned char l1[sobel_width];
        unsigned char l2[sobel_width];
        unsigned char l3[sobel_width];

        memcpy(l1, source + (y - 1) * sobel_width, sobel_width);
        memcpy(l2, source + y * sobel_width, sobel_width);
        memcpy(l3, source + (y + 1) * sobel_width, sobel_width);


        for (x = 1; x < (sobel_width - 1); x++) {
            int arrayIndex = y * sobel_width + x;

            short gx =  l1[x - 1] * gx_array[0][0] +
                  l1[x] * gx_array[0][1] +
            l1[x + 1] * gx_array[0][2] +
            l2[x - 1] * gx_array[1][0] +
            l2[x] * gx_array[1][1] +
            l2[x + 1] * gx_array[1][2] +
            l3[x - 1] * gx_array[2][0] +
            l3[x] * gx_array[2][1] +
            l3[x + 1] * gx_array[2][2];

            short gy =  l1[x - 1] * gy_array[0][0] +
                  l1[x] * gy_array[0][1] +
            l1[x + 1] * gy_array[0][2] +
            l2[x - 1] * gy_array[1][0] +
            l2[x] * gy_array[1][1] +
            l2[x + 1] * gy_array[1][2] +
            l3[x - 1] * gy_array[2][0] +
            l3[x] * gy_array[2][1] +
            l3[x + 1] * gy_array[2][2];

            short sum = abs(gx) + abs(gy);

            sobel_result[arrayIndex] = ALT_CI_THRESHOLD_CI_0(sum, threshold);
        }
    }
}
```

With -O3 : 103 cycles/pixel

6 Optimization processor

We have modified the parameters of the processor in Qsys as shown in the following picture.

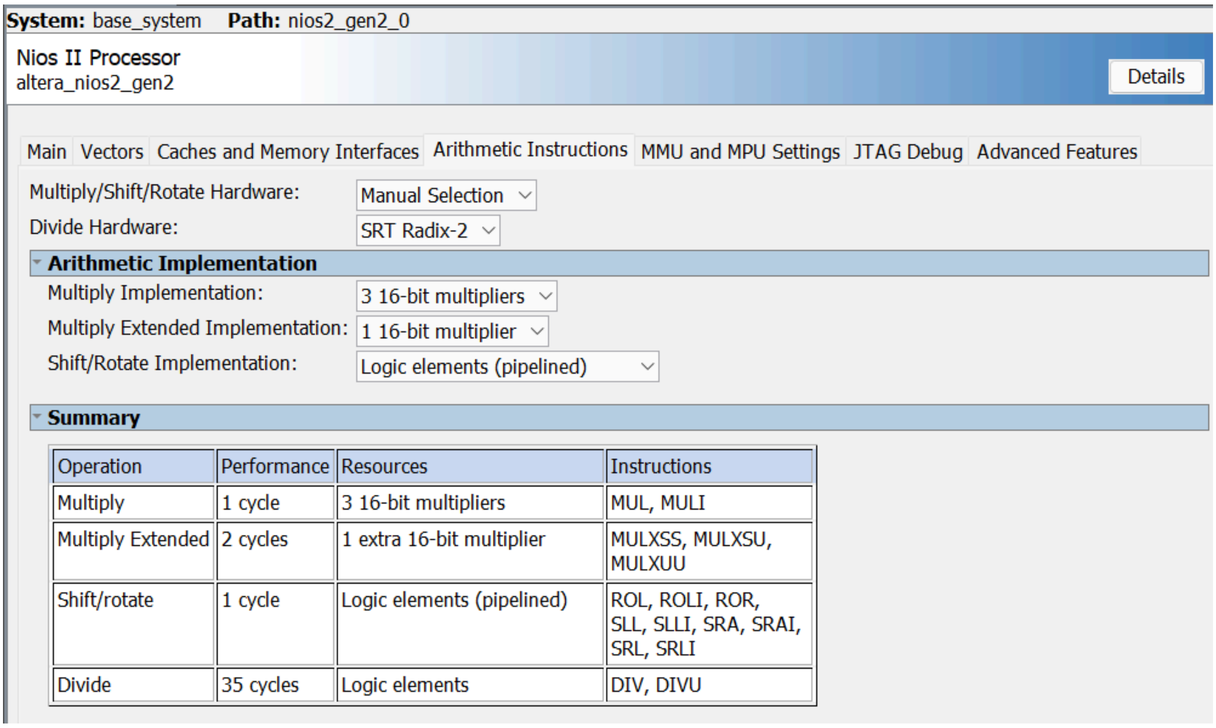

Figure 2: Settings optimization processor

With -O3 : 92 cycles/pixel

7 Conclusion

The final performance of the system are the following :

With -O3 : 97 cycles/pixel (31 for grayscale and 66 for the sobel)
With -O3 : 2.59 FPS

This performance are a mean of 100 measures.

All the performances measured during the optimization process are detailled in the following chart and graphe. All modifications made are included in the following tests.

Test	Compiler option	Grayscal	Sobel complet	Sobel	Sobel	Threshold	ycles/p	FPS
Without optimization	-O0	600		3673	3674	520	**8467**	
Without optimization	-O1	118		642	642	168	**1570**	
Without optimization	-O2	102		541	541	153	**1337**	
Without optimization	-O3	112		146	134	153	**545**	
Loop unrolling sobel_mac (inner loop)	-O0	661		2806	2809	531	**6807**	
Loop unrolling sobel_mac (outer loop)	-O0	664		2196	2209	537	**5606**	
In-lining sobel_x and sobel_y	-O3	107		219	220	64	**630**	**0,40**
Sobel optimization	-O3	112		146	134	153	**545**	**0,47**
Sobel optimization	-O3	112		146	134	153	**545**	**0,47**
Sobel complete	-O3	115	260				**375**	**0,68**
Grayscale opti	-O3	75	259				**334**	**0,76**
Cache I/D 16k	-O3	32	108				**140**	**1,82**
CI threshold	-O3	30	104				**134**	**1,90**
Use memcpy	-O3	32	71				**103**	**2,47**
Optimization param. Processor	**-O3**	**32**	**61**				**97**	**2,59**

Figure 3: Performance chart

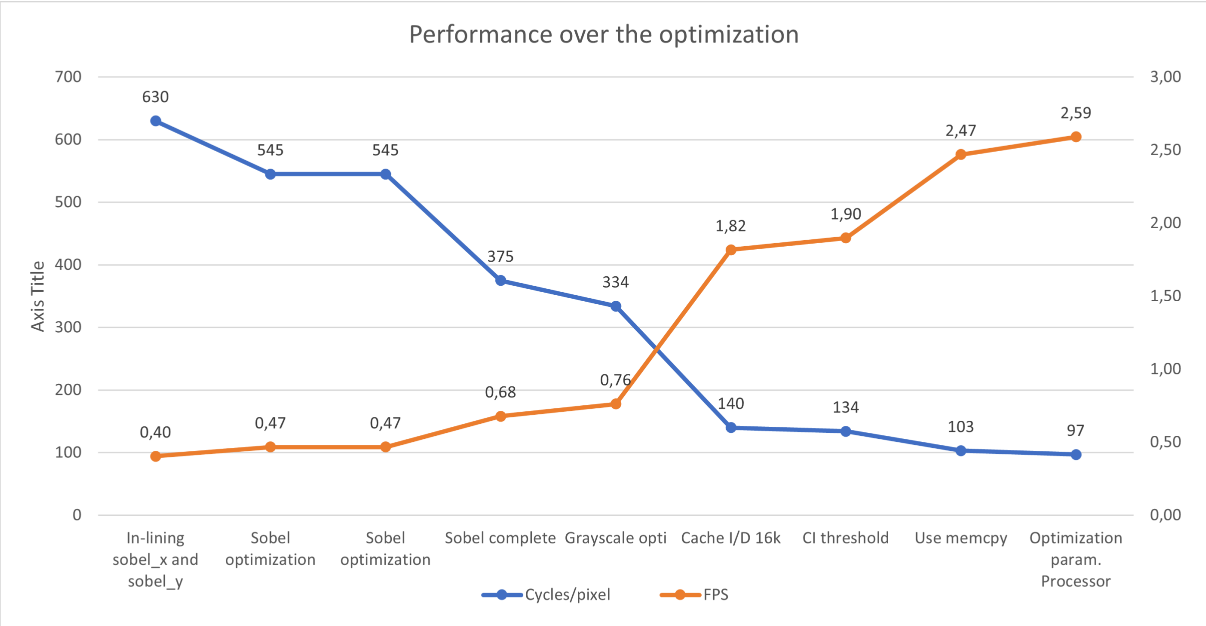

Figure 4: Performance graph

We can see that using the compiler optimization -O3, only one function and the I/D caches made great improvement.

7.1 Further optimization

To continue improving performance, it is possible to create a custom instruction for the grayscale and the sobel. A pipeline can be put in place. It is necessary to write a vhdl code. This code contains different module :

- Instruction fetch
- Decode
- Execute
- Memory
- Align
- Writeback

All these modules need to communicate with the other pipeline stages using registers or buffers.

If the project has to be remake from the beggining, it will be intresting to investigate other kind processors with a higher clock frequency. In general, increasing the frequency of a processor can lead to improved performance. The used processor is a one core processor. using a processor with multiple core can also increase the performances. Finally, to save optimization time, it will be better to start directly the implementation with the pipeline in vhdl.