OBJECT TRACKING

CS-476 Embedded System Design

Final Project

Andrea Grillo
Riccardo Lionetto

04/06/2024

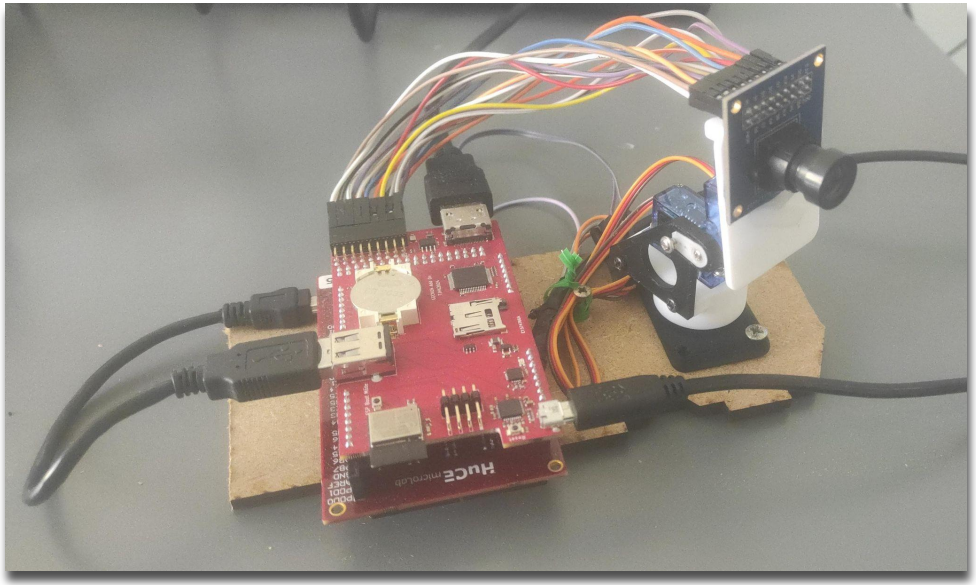
Overview

- Idea
- Hardware setup
- Preliminary sw/hw validation
- Software solution
- Hotspot Detection
- Hardware solution
- Performance analysis
- Possible Improvements

EPFL

Idea

- Object tracking
  - Object detection using camera
  - Motorized end effector to follow the object



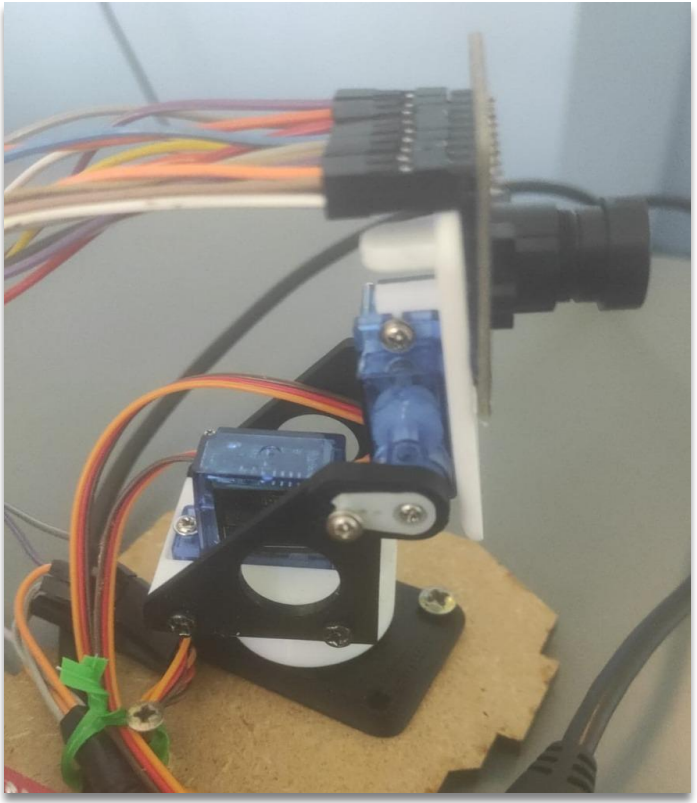
EPFL

Hardware setup - pan tilt

EPFL

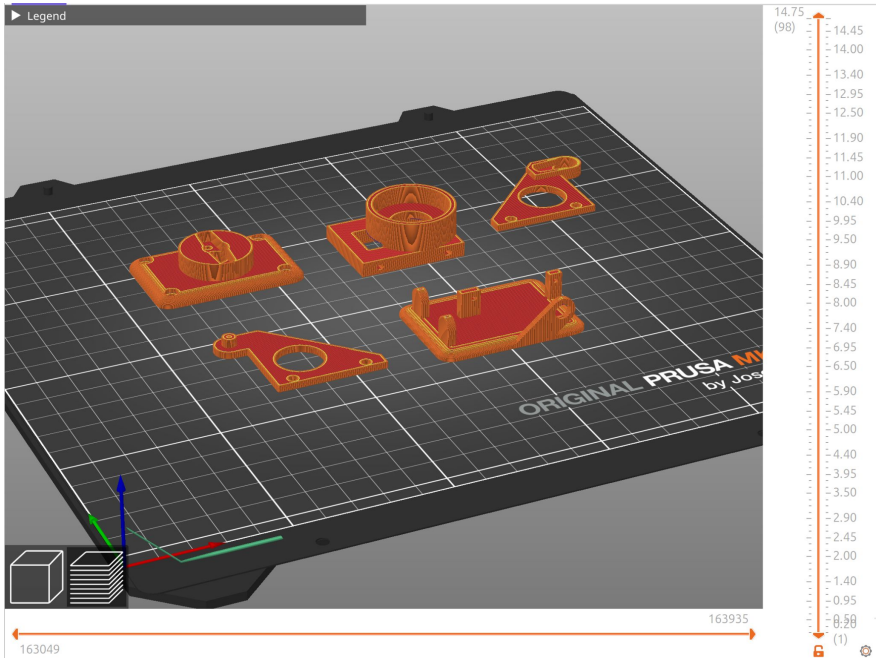
Hardware setup

- 2DoF end effector: pan tilt



Hardware setup

- **Design of the 3D structure**



EPFL

Hardware setup

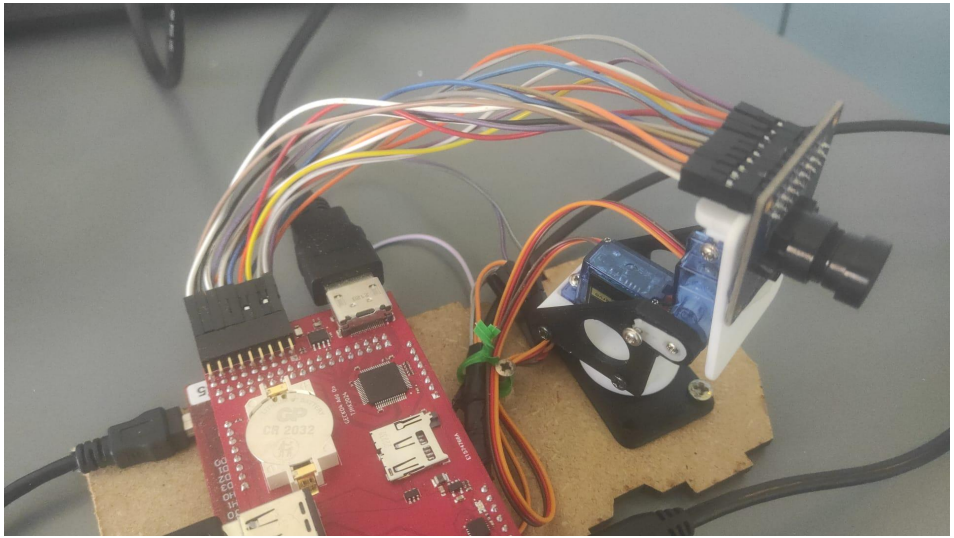
- **Use of simple servo motors**:
  - 180° rotation angle
  - PWM position controlled

EPFL

Hardware setup

- **Extended connection camera-board:**

To empower the pan tilt with more free movements, longer connections were put in place.



EPFL

Preliminary algorithm validation

EPFL

Preliminary algorithm validation

- Object detection common method:
  a. Conversion from RGB to HSV - more robust to change in lighting
  b. Thresholding to get the mask of the object
  c. Contour detection
  d. Isolation of the wanted contour
  e. Center of the contour

- Very well known library: OpenCV

EPFL

Preliminary algorithm validation

- Too complex to be run on our architecture, our idea:
  a. RGB thresholding
  b. Averaging of the points of the right color

- Development of Proof of Concept using **OpenCV** in Python:

```python
# Threshold the RGB image
mask = cv2.inRange(frame, lower_red, upper_red)

# Find the coordinates of all 1 values
coordinates = np.argwhere(mask == 255)

frame = cv2.bitwise_and(frame, frame, mask=mask)

if len(coordinates) != 0:
    # Calculate the mean of the coordinates along each axis (rows and columns)
    average_coordinates = np.mean(coordinates, axis=0)
    print("avg coord", int(average_coordinates[0]), int(average_coordinates[1]))

    cv2.circle(frame, (int(average_coordinates[1]), int(average_coordinates[0])), 5, (0, 0, 255), -1)
```

Preliminary algorithm validation

The idea works!

Preliminary algorithm validation

- Limitations of this algorithm:
  a. Only one object of that color
  b. Color distinct from the environment
  c. Not very robust to lighting perturbations due to lack of conversion to HSV

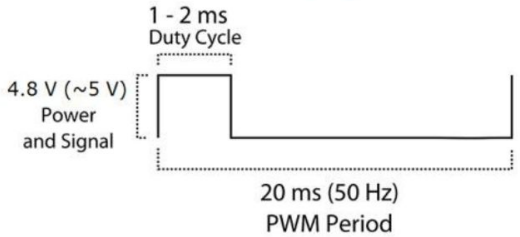
- But it worked in every environment we have tested!

Preliminary hardware testing

EPFL

Interfacing with the servos

- Controlled by PWM signal:
  a. Position control
  b. Amplitude of the high part of the signal is proportional to the angle



- Will have to generate the PWM signal in hardware
- We first developed a proof of concept using arduino

EPFL

Interfacing with the servos -preliminary tests

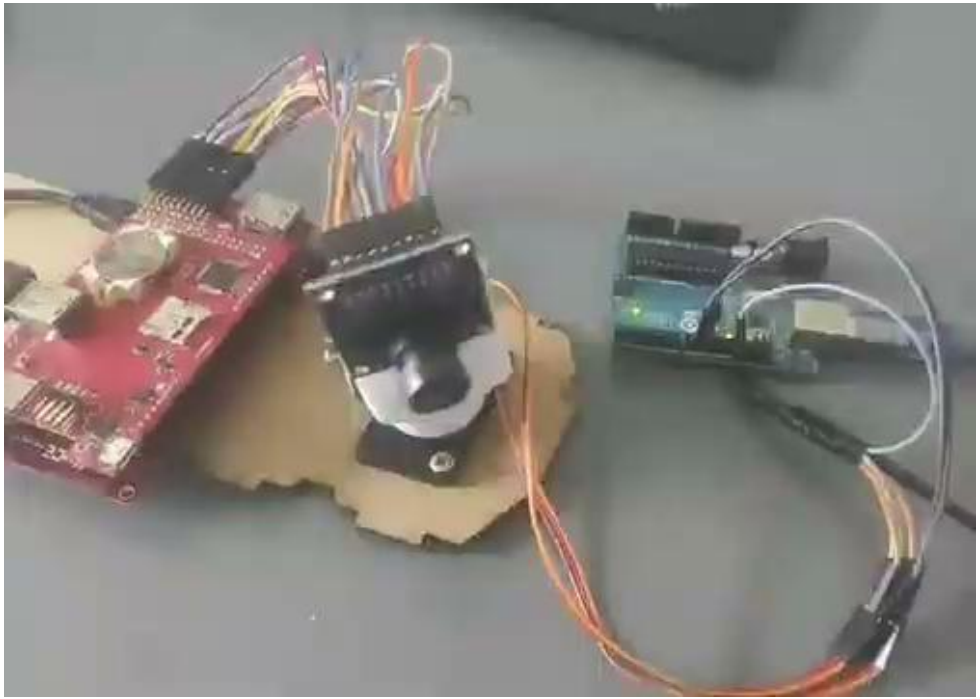
- Initial tests on the servos were prepared faster by means of Arduino

```
Servo servo1, servo2;
int pos = 90;

void setup() {
  servo1.attach(9);
  servo2.attach(10);
  servo1.write(pos);
  servo2.write(pos);
}

void loop() {
    for (pos = 55; pos <= 125; pos += 1) {
    servo1.write(pos);
    servo2.write(pos);
    delay(15);
  }
  for (pos = 125; pos >= 55; pos -= 1) {
    servo1.write(pos);
    servo2.write(pos);
    delay(15);
  }
}
```

EPFL

Interfacing with the servos -preliminary tests

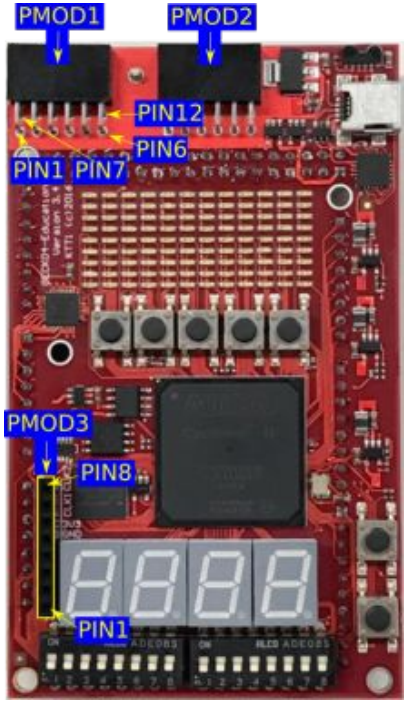
- First movements!



EPFL

Interfacing with the servos -FPGA implementation

- Servos PWMs were generated on the FPGA PMOD pins



Problem: servos required 5V, pins output is 3.3V

Solution: we were ready to create an additional level shifter circuit to make the tensions compatible, but it was not necessary

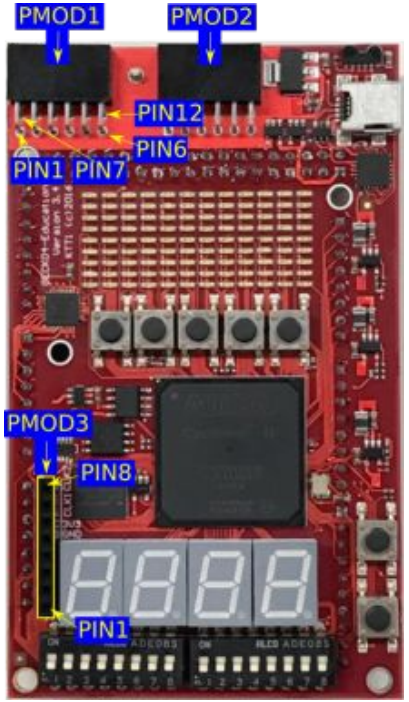
→ servos are able to correctly interpret high level already at 3.3V, which is then above the threshold

EPFL

Interfacing with the servos -FPGA implementation

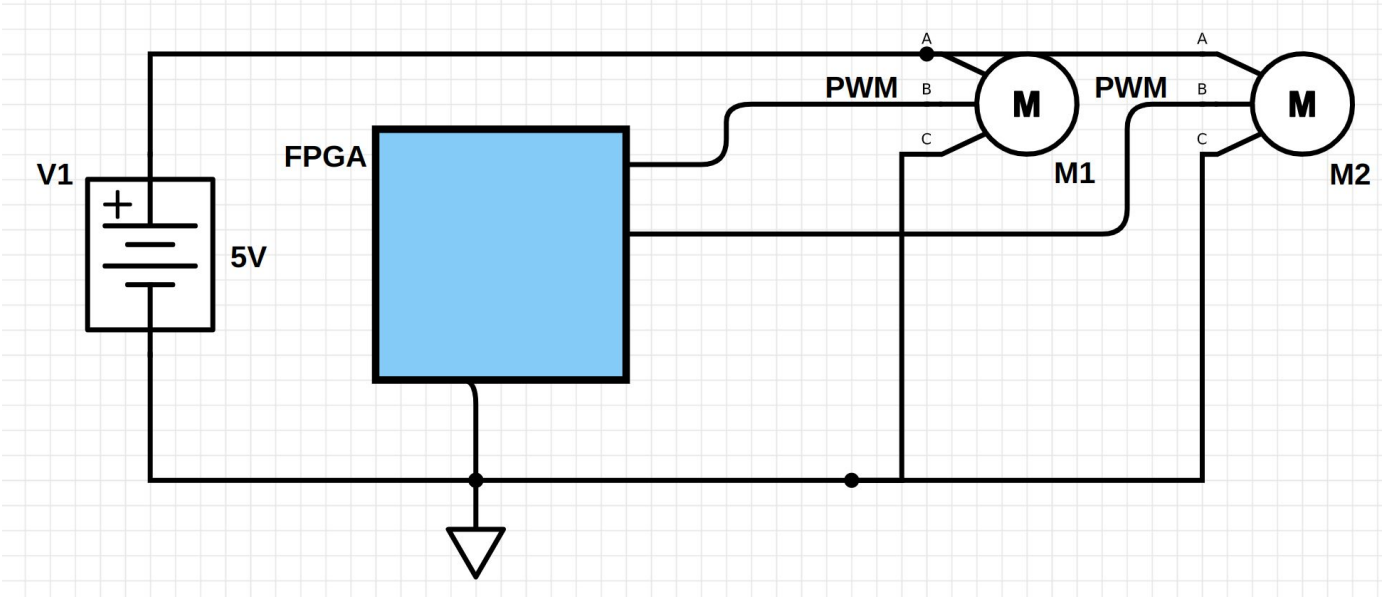
- PMOD1 connector
- TCL file pin assignment
- Definition in the Top Level Entity

Pin number:	1	2	3	4	5	6	7	8	9	10	11	12
Function:	I/O 1	I/O 2	I/O 3	I/O 4	GND	3V3	I/O 7	I/O 8	I/O 9	I/O 10	GND	3V3
FPGA Pin:	PIN_F2	PIN_E3	PIN_C2	PIN_B2			PIN_F1	PIN_E4	PIN_C1	PIN_B1		



Interfacing with the servos

- External power supply: adapted USB cable
- Common ground



EPFL

Interfacing with the servos

- PWM Generator Custom instruction:

1. write the needed duty cycles to control the two motors position
2. internal module counter: when count < duty cycle → high pin output
   when count > duty cycle → low pin output

Working frequency:

- ideal → 50Hz
- actual → rounded up to `72MHz/1.048.576 = 69Hz` to use a counting register that didn't need an active operation of reset

EPFL

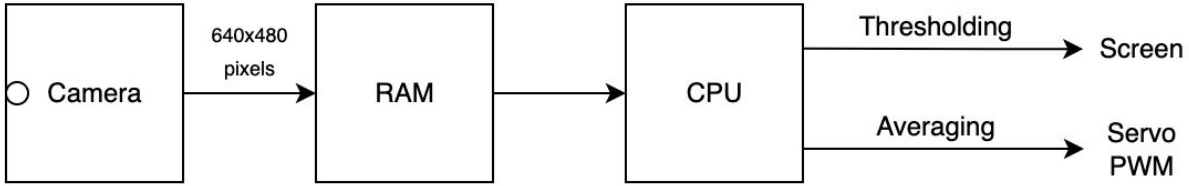
Interfacing with the servos

- First tests of movement controlled by the Gecko Board

Software solution

EPFL

Software solution



- Steps:
  a. Acquire single frame
  b. Thresholding and conversion to grayscale
  c. Average of the coordinates of the masked points
  d. Visualization of center point
  e. Position controller and setting of the PWM duty cycle

EPFL

Software solution

- For thresholding, detection boundaries were fine-tuned to maximize the performances of the object detection.

First idea: use values from OpenCV for the onboard camera.

Solution:

a. print the values of the pixel in the middle of the image
b. put the sheet in front of the camera
c. choose an interval around the visualized point

EPFL

Software solution

- Thresholding and grayscale conversion

```
for (int line = 0; line < camParams.nrOfLinesPerImage; line++) {
  for (int pixel = 0; pixel < camParams.nrOfPixelsPerLine; pixel++) {
    uint16_t rgb = swap_u16(rgb565[line*camParams.nrOfPixelsPerLine+pixel]);

    uint8_t red1 = ((rgb >> 11) & 0x1F) << 3;
    uint8_t green1 = ((rgb >> 5) & 0x3F) << 2;
    uint8_t blue1 = (rgb & 0x1F) << 3;

    if(lowerRed < red1 && red1 < upperRed  &&
       lowerGreen < green1 && green1 < upperGreen &&
       lowerBlue < blue1 && blue1 < upperBlue){
      sum++;
      sum_line += line;
      sum_pixel += pixel;
    } else {
      uint8_t gray = ((red1*54+green1*183+blue1*19) >> 8) & 0xFF;
      uint16_t gray16 = (gray & 0xF8) << 8 | (gray & 0xFC) << 3 | (gray & 0xF8) >> 3;
      rgb565[line*camParams.nrOfPixelsPerLine+pixel] = swap_u16(gray16);
    }
  }
}
```

thresholding

grayscale conversion

EPFL

Software solution

- Averaging and object's center displaying

```
if(sum > 100){
  int avg_line = sum_line/sum;
  int avg_pixel = sum_pixel/sum;
  for(int i = avg_line - 3; i < avg_line + 3; i++){
   for(int j = avg_pixel - 3; j < avg_pixel + 3; j++){
      rgb565[j*camParams.nrOfPixelsPerLine+i] = 0xFFFF;
    }
  }
}
```

Hotspot detection

EPFL

Analysis of the software detection

- Profiling of the thresholding and averaging sections (started after the frame acquisition):

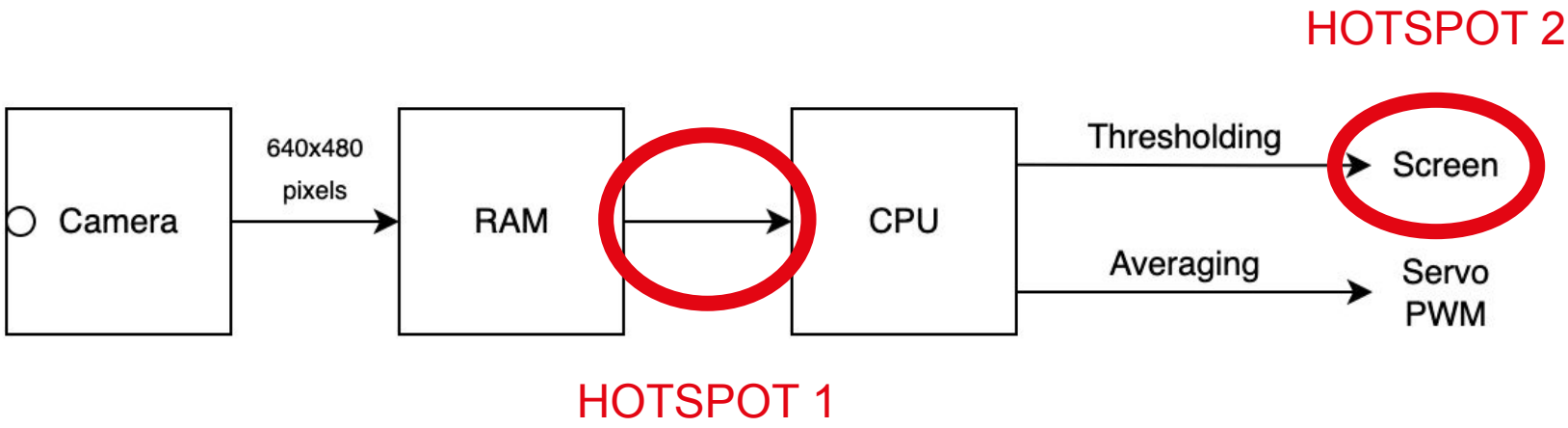
CPU cycles	~44 millions
CPU stall cycles	~26 millions
BUS idle cycles	~23 millions

Execution time	~0.6 seconds
CPU waiting time	~60%
BUS busy time	~48%

The CPU is waiting for 60% of the time and working for only 40% [18 million cycles]..

EPFL

Analysis of the software detection

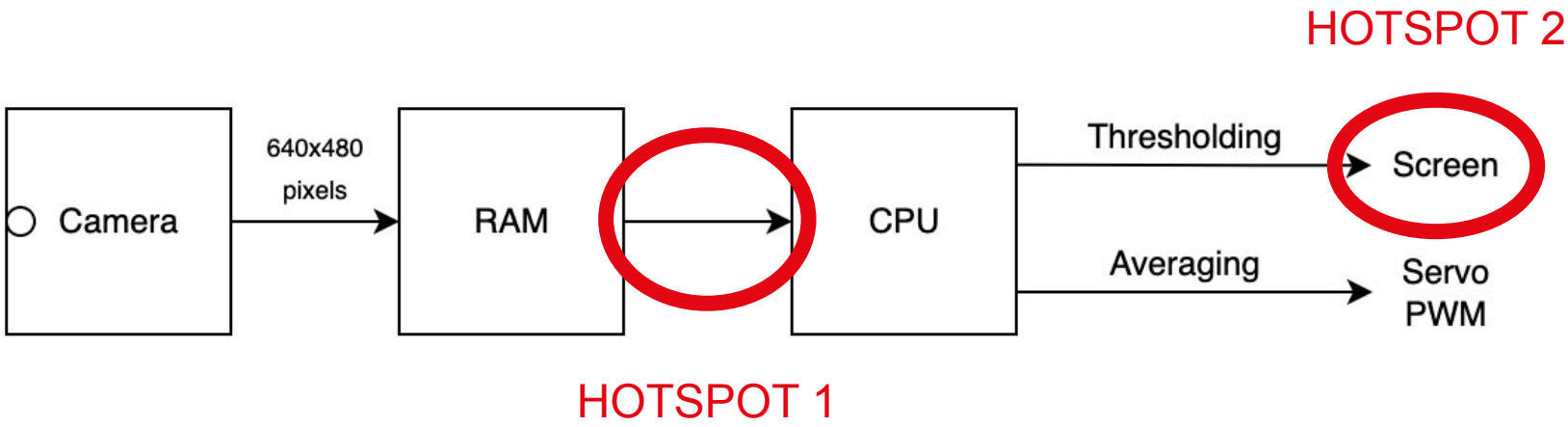


The CPU has to read the whole frame from the RAM and then write it back.

The transfers to and from the RAM are the hotspot in our program.

EPFL

Analysis of the software detection



The frame is big in size: `640x480pixels * 16bits = 614,4 Kbyte`

even with a data cache multiple evictions and refill of the lines are needed

Hardware solution

EPFL

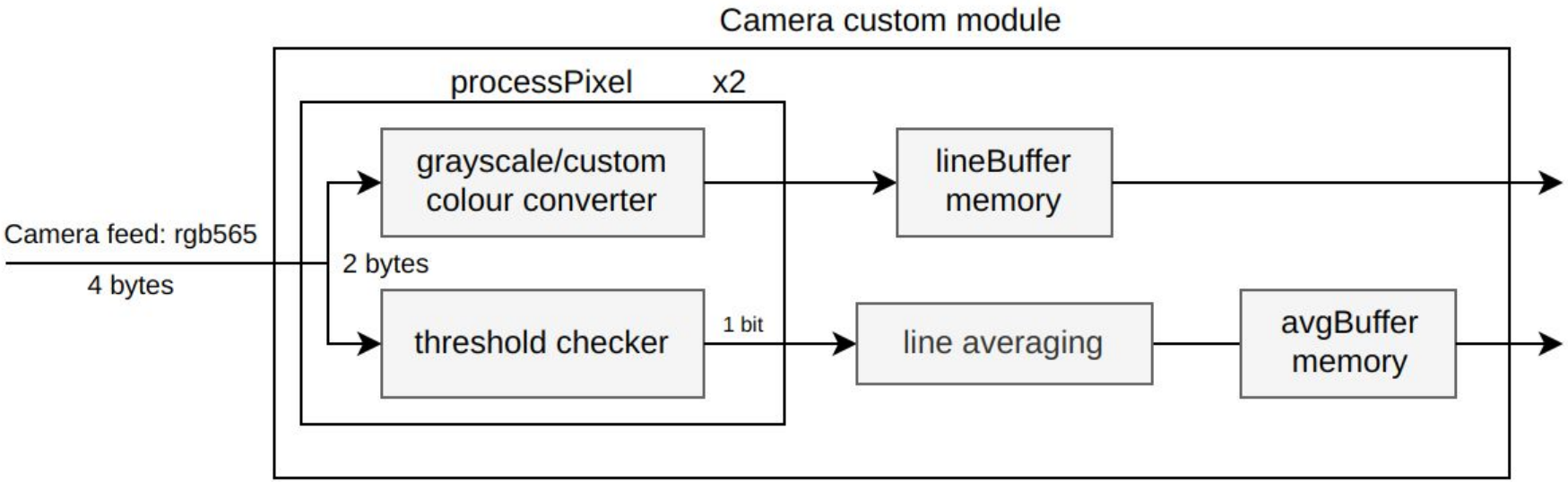
Hardware solution

- Key features we wanted to implement:
  a. keep camera visualization
  b. grayscale background with detected object still coloured [powerful debugging feature]
  c. center of object masked with white pixels
  d. PWM servo control

- How it was achieved:
  a. **camera custom module** edited to have in-stream thresholding and grayscale conversion
  b. **sw averaging** done by the CPU
  c. **custom module** used for the generation of PWM signals

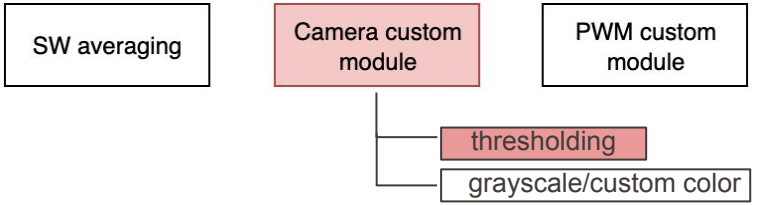
Hardware solution

Structure of the in-stream pixel processing:



EPFL

Hardware solution



In hardware **thresholding**:

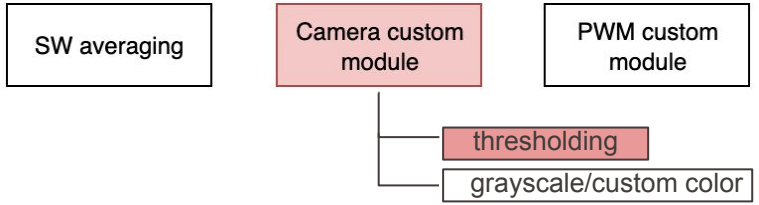
- **1° version**: the previously found fine-tuned values were firstly used

```
assign s_rok = s_red > 5'd15 & s_red < 5'd25;
assign s_gok = s_green < 6'd10;
assign s_bok = s_blue > 5'd5 & s_blue < 5'd15;
```

However, it required the use of comparators to compare rgb channel values with threshold, as we did in software → expensive logic

EPFL

Hardware solution



In hardware thresholding:

- **2° version**: eliminate comparators to simplify and lower used logic

```
// Check with no use of comparison operators
assign s_rok = ~s_red[4] & s_red[3];
assign s_gok = ~s_green[5] & ~s_green[4] & ~s_green[3];
assign s_bok = (s_blue[2] | (s_blue[1] & s_blue[0])) & ~s_blue[4] & ~s_blue[3];
```

Fine-tuned values were rounded up to the closest values that allowed logic simplification.

e.g. - before: `green_value < 10`

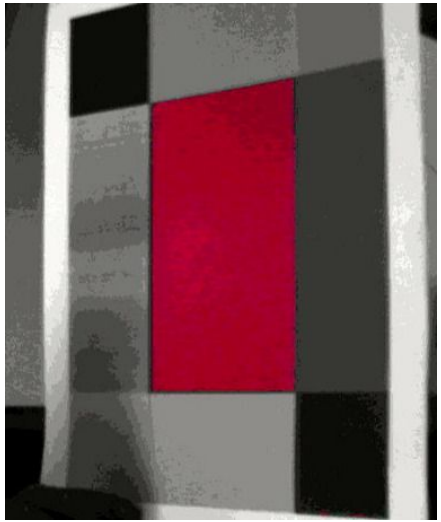
- now: `green_value < 7`

EPFL

Hardware solution

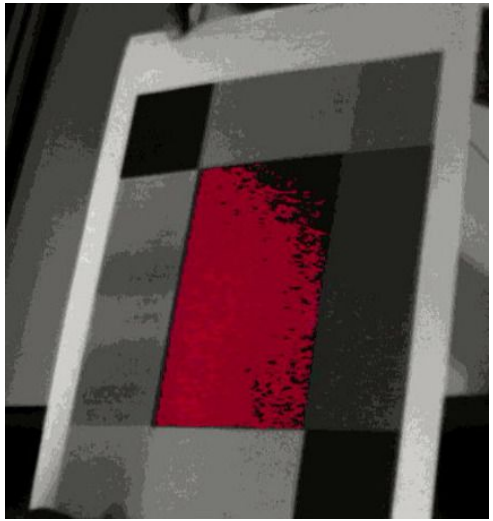
SW averaging | Camera custom module | PWM custom module

- **Fine-tuned limits:**



- more complex logic
- better performances

- **Approximated limits**



- less robust
- space optimization

EPFL

Hardware solution

SW averaging

Camera custom module

PWM custom module

The question is:

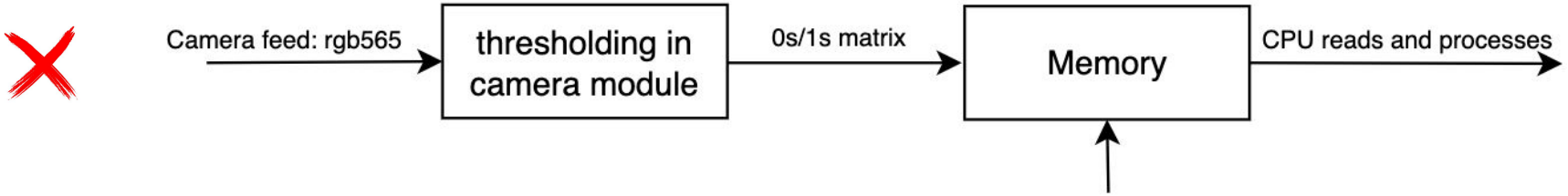
How to do the averaging?

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- How to do the averaging, ideas:



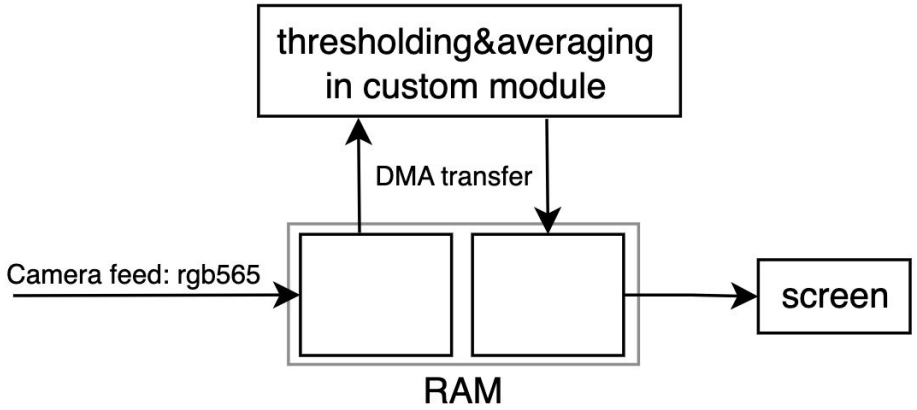
`640x480 = 307200 pixels → 307200/32bit words = 9600lines → 38k memory` ⟶ too big!

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- How to do the averaging, ideas:



Inspired by PW6: would result in intense use of bus by the DMA transfers, which need to go back and forth

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- Do the thresholding in-stream 
- Cannot save the whole mask 
- Need to find an intermediate solution for the averaging part

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- Averaging to find the center of the object:

given the list of points that belong to the object: $(x_i, y_i) \quad \forall i \in 1, .., n$

the coordinates of the center $(\bar{x}, \bar{y})$ are found as:

$$\bar{x} = \frac{\sum x_i}{n} \qquad \bar{y} = \frac{\sum y_i}{n}$$

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- Ideas to simplify this operation:

a. save the list of points as a sparse matrix instead of the mask

- the memory needed to save the points has varying size for each frame, difficult to handle

b. split the averaging over the rows and the columns and save only intermediate results:

for each line (j) compute: number of points in the object n_j, sum of the indexes of the points s_j

the coordinates of the center are then found as:

$$\bar{x} = \frac{\sum_j s_j}{n} \qquad \bar{y} = \frac{\sum_j j \cdot n_j}{n} \qquad n = \sum_j n_j$$

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- Idea:
  a. Line computation in-stream in hardware
  b. Final averaging in software

$$n_{pix} = 640$$

$$n_{lin} = 480$$

- Intermediate results to store:
  a. $n_j \in 0, .., n_{pix}$ fits in 10 bits
  b. $s_j \in 0, .., k$ $k = \frac{n_{pix}(n_{pix}+1)}{2} = 205120$ fits in 18 bits

→ fit in a 32 bit word! ✓

- We need to store 480 32 bit words → fit in a 2k memory! ✓

- Added a 2k memory in the camera module

EPFL

Hardware solution

SW averaging | Camera custom module | PWM custom module

- Intermediate results computation in hardware:

```
reg [11:0]  sum_1Pixels;
reg [19:0] sum_idxPixels_line;
wire [19:0] idx_Pixel= {10'd0, s_pixelCountReg[10:1]};

always @(posedge pclk)
begin
  sum_1Pixels <= (reset == 1'b1 || s_hsyncNegEdge) ? 12'd0 :
                 s_weLineBuffer ? (sum_1Pixels + {11'd0,outputMask1} + {11'd0, outputMask2}) : sum_1Pixels;
  sum_idxPixels_line <= (reset == 1'b1 || s_hsyncNegEdge) ? 20'd0 :
                 s_weLineBuffer ? (sum_idxPixels_line +
                                   (outputMask1 ? (idx_Pixel-20'd1) : 20'd0) +
                                   (outputMask2 ? idx_Pixel : 20'd0)) : sum_idxPixels_line;
end
```

EPFL

Hardware solution

SW averaging

Camera custom module

PWM custom module

- Final averaging and center computation in software:

```
for (int i = 0; i < camParams.nrOfLinesPerImage; i++) {
  asm volatile ("l.nios_rrc %[out1],%[in1],%[in2],0x7":[out1]"=r"(result):[in1]"r"(16),[in2]"r"(i));
  index_pixels_per_line = result >> 12;
  pixels_per_line = result & 0xFFF;
  sum_pixels += pixels_per_line;
  sum_index_pixels += index_pixels_per_line;
  sum_lineindex_pixels_per_line += pixels_per_line * i;
}

if(sum_pixels > 100){
  int avg_line  = sum_index_pixels / sum_pixels;
  int avg_pixel = sum_lineindex_pixels_per_line / sum_pixels;
}
```

EPFL

Hardware solution

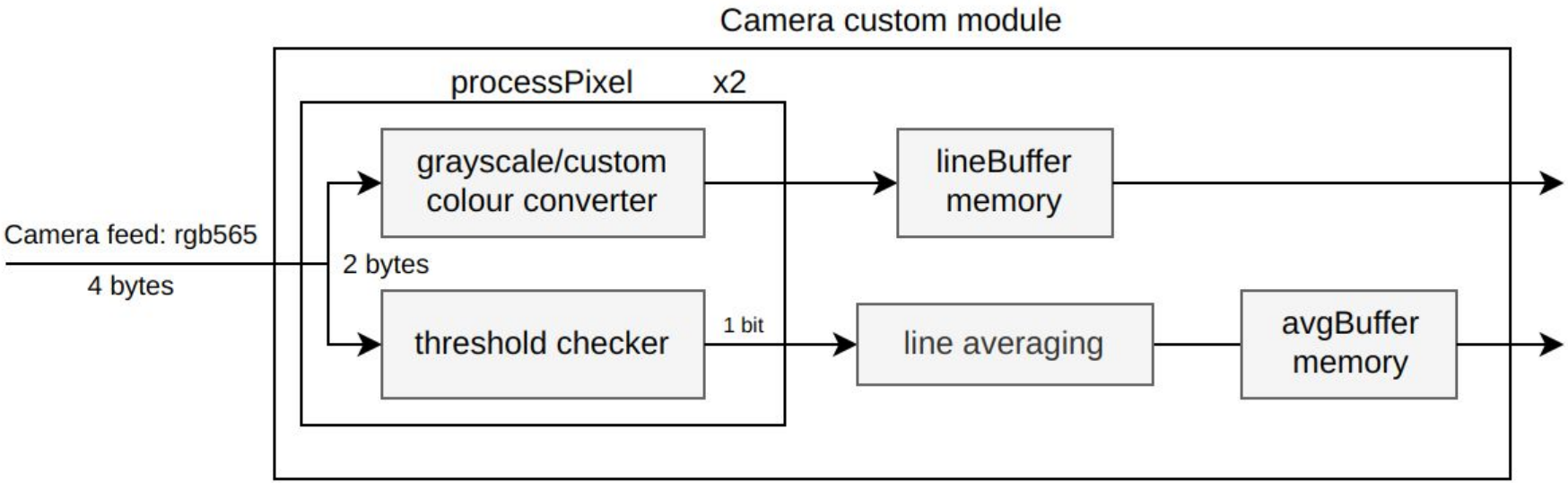
SW averaging | Camera custom module | PWM custom module

- **PWM custom module:**

The module is capable of generating two PWM signals at the same time, to singularly control the two servos of the pan tilt.

Hardware solution

Structure of the in-stream pixel processing:



Performance analysis

EPFL

Performance analysis

- **Assembly code analysis:**

Analysing the assembly code allowed us to spot implementation errors in the C code.

e.g. useless access to memory due to *volatile* qualifiers for variables that did not need it

EPFL

Performance analysis

- **Assembly code analysis:**

```
for (int i = 0; i < camParams.nrOfLinesPerImage; i++) {
  asm volatile ("l.nios_rrc %[out1],%[in1],%[in2],0x7":[out1]"=r"(result):[in1]"r"(16),[in2]"r"(i));
  index_pixels_per_line = result >> 12;
  pixels_per_line = result & 0xFFF;
  sum_pixels += pixels_per_line;
  sum_index_pixels += index_pixels_per_line;
  sum_lineindex_pixels_per_line += pixels_per_line * i;
}
```

```
.L4:
    l.sfne  r17, r22
    l.bf    .L5
    l.ori   r19, r0, 16

.L5:
#  61 "src/project.c" 1
    l.nios_rrc r21,r28,r17,0x7
#  0 "" 2
    l.ori   r25, r0, 12
    l.andi  r19, r21, 4095
    l.add   r2, r2, r19
    l.sra   r21, r21, r25
    l.mul   r19, r17, r19
    l.add   r3, r3, r21
    l.add   r23, r23, r19
    l.j .L4
    l.addi  r17, r17, 1
```

The code is optimized by the compiler to keep all the values in registers and not write them into memory.

EPFL

Performance analysis

- **Assembly code analysis - averaging:**

```
for (int i = 0; i < camParams.nrOfLinesPerImage; i++) {
  asm volatile ("l.nios_rrc %[out1],%[in1],%[in2],0x7":[out1]"=r"(result):[in1]"r"(16),[in2]"r"(i));
  index_pixels_per_line = result >> 12;
  pixels_per_line = result & 0xFFF;
  sum_pixels += pixels_per_line;
  sum_index_pixels += index_pixels_per_line;
  sum_lineindex_pixels_per_line += pixels_per_line * i;
}
```

```
.L4:
    l.sfne  r17, r22
    l.bf    .L5
    l.ori   r19, r0, 16

.L5:
#  61 "src/project.c" 1
    l.nios_rrc r21,r28,r17,0x7
#  0 "" 2
    l.ori   r25, r0, 12
    l.andi  r19, r21, 4095
    l.add   r2, r2, r19
    l.sra   r21, r21, r25
    l.mul   r19, r17, r19
    l.add   r3, r3, r21
    l.add   r23, r23, r19
    l.j .L4
    l.addi  r17, r17, 1
```

13 instructions repeated 480 (number of lines) times -> **6240** instructions

The custom instruction that reads from the 2k memory stalls the cpu for 1 cycle -> at least **480** stall cycles

Measured values for this part of the code: Total cycles: **7745** Stall cycles: **536**

As we have no access to memory those cycles are constant at each execution step.

EPFL

Performance analysis

- **Code analysis - center computation, square on the image, pwm:**

```
if(sum_pixels > 100){
  int avg_line  = sum_index_pixels / sum_pixels;
  int avg_pixel = sum_lineindex_pixels_per_line / sum_pixels;

  // printf("avg line %d avg pixel %d \n", avg_line, avg_pixel);

  for(int i = avg_line - 3; i < avg_line + 3; i++){
   for(int j = avg_pixel - 3; j < avg_pixel + 3; j++){
     rgb565[j*camParams.nrOfPixelsPerLine+i] = 0xFFFF;
    }
  }

  if(avg_line < MIDDLE_LINE) pwm_x = pwm_x < MIN ? MIN : (pwm_x - DEG);
  else                       pwm_x = pwm_x > MAX ? MAX : (pwm_x + DEG);

  if(avg_pixel < MIDDLE_PIXEL) pwm_y = pwm_y < MIN ? MIN : (pwm_y - DEG);
  else                         pwm_y = pwm_y > MAX ? MAX : (pwm_y + DEG);

  asm volatile("l.nios_rrr r0,%[in1],%[in2],21" :: [in1] "r"(ENABLE_BOTH_PWM | SET_PWM_1), [in2] "r"(pwm_y));
  asm volatile("l.nios_rrr r0,%[in1],%[in2],21" :: [in1] "r"(ENABLE_BOTH_PWM | SET_PWM_2), [in2] "r"(pwm_x));
}
```

for this part is harder to predict how many cycles will take the execution, as there are accesses to memory and a few branches that depend on the execution.

EPFL

Performance analysis

- **Profiling trend analysis:**

  Profiling results were analysed on ***Excel*** to extract trends:

  [based on *700 samples*]

	Avg	Max	Min	Std Dev
CPU cycles	10070	16522	7806	1958
CPU stall cycles	2213	8157	592	1491
BUS idle cycles	5258	9274	1028	2358

EPFL

Final performance comparison

- **Software solution**

CPU cycles	~44 millions
CPU stall cycles	~26 millions
BUS idle cycles	~23 millions

Execution time	~0.6 seconds
CPU waiting time	~60%
BUS busy time	~48%

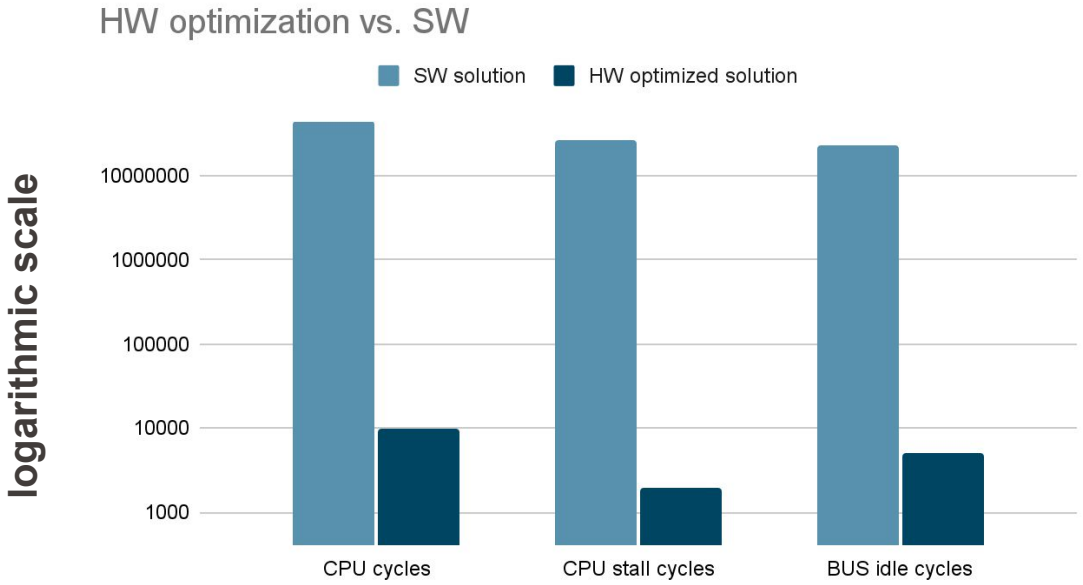
- **Optimized hardware solution**

CPU cycles	~10k
CPU stall cycles	~2k
BUS idle cycles	~5k

Execution time	**~240 microseconds**
CPU waiting time	~20%
BUS busy time	~50%

EPFL

Final performance



4400x faster

Possible improvements

EPFL

Possible improvements

- **Achieve even higher speed:**

Current implementation: one of the goals is the real time visualization of the camera feed, with a custom grayscale filter that keeps colored only the detected object. To achieve this a 2K memory was added.

To have an even faster system and to still see the pan tilt tracking, it'd be enough to get rid of the camera feed and make all computations on stream!

As soon as the frame is received by the camera module, the center coordinates can be already computed.

EPFL

Possible improvements

- **Connections between camera and board:**

Current implementation: it can occur that with some movements the connection between the camera and the board, because of the presence of additional external wires, fail to reliably transport the signal.

Better connectors and wires with lower resistance and parasitic capacitance could help to make the system more reliable.

EPFL

Possible improvements

- **Achieve even higher speed:**

Current implementation: one frame is transferred to the RAM memory and then the center of the object is computed. When the computation is finished the next frame is requested.

During the computation we do not request access to the bus. In this time the next frame could be transferred to memory, we would need to add another memory to the camera module and use a ping-pong strategy.

Thank You