ESTENSIONE DEL PROGETTO DI COMPUTER VISION INERENTE IL RICONOSCIMENTO DI BIELLE (ATTIVITÀ PROGETTUALE 4CFU)

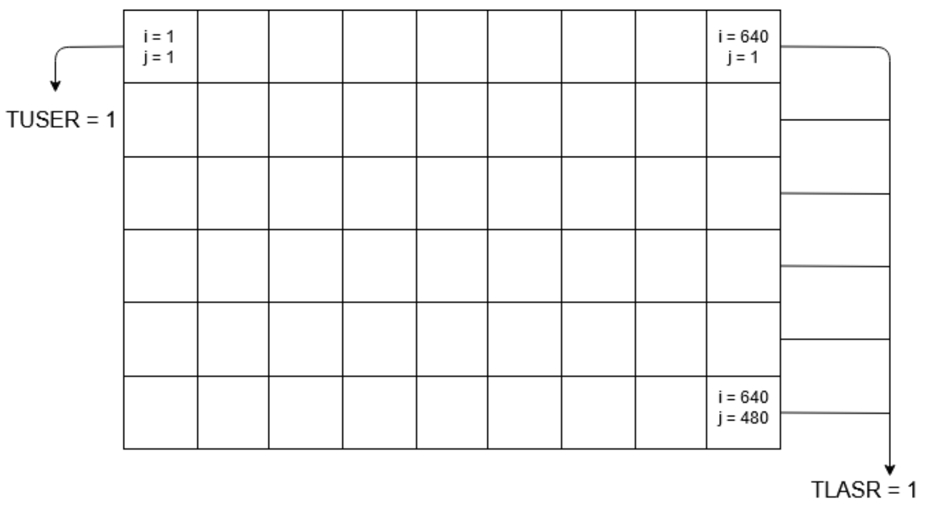
IMAGE PROCESSING ON FPGA AS OPENCV-PYTHON PROGRAM ACCELERATOR

Relazione e progetto di Lorenzo Balducci

1) Introduction

The aim of this project is to propose a way for implementing the features required by the connected rods recognition program, in an FPGA. The entire pipeline of the pc program, that is already implemented in python, takes advantage of random access over the whole photogram stored in memory and sequential algorithms. For taking advantage of an FPGA capability we should process, as much as possible, all the data flowing in the video stream without a random access over an entirely stored image. (This approach is also not possible at all for big resolutions and small sized FPGA).
The FPGA that will be used as reference is the xilinx zybo z7-10; this board is one of the cheapest available by xilinx, the size of the FPGA is just 240KB BRAM and 17K LUT, and the ARM processor is just a dual-core 650mhz fanless. However, the size of the FPGA will be enough for implementing the hardware needed for the performance boost we are looking for, and the performance of the ARM processor is enough because we will just handle the ethernet communication in the PS side.

All the processing steps works upon an AxiVideoStream that is an AxiStream protocol in which every data is a pixel and the tlast and tuser signals are used to indicate last pixel of a row and first pixel of a photogram.

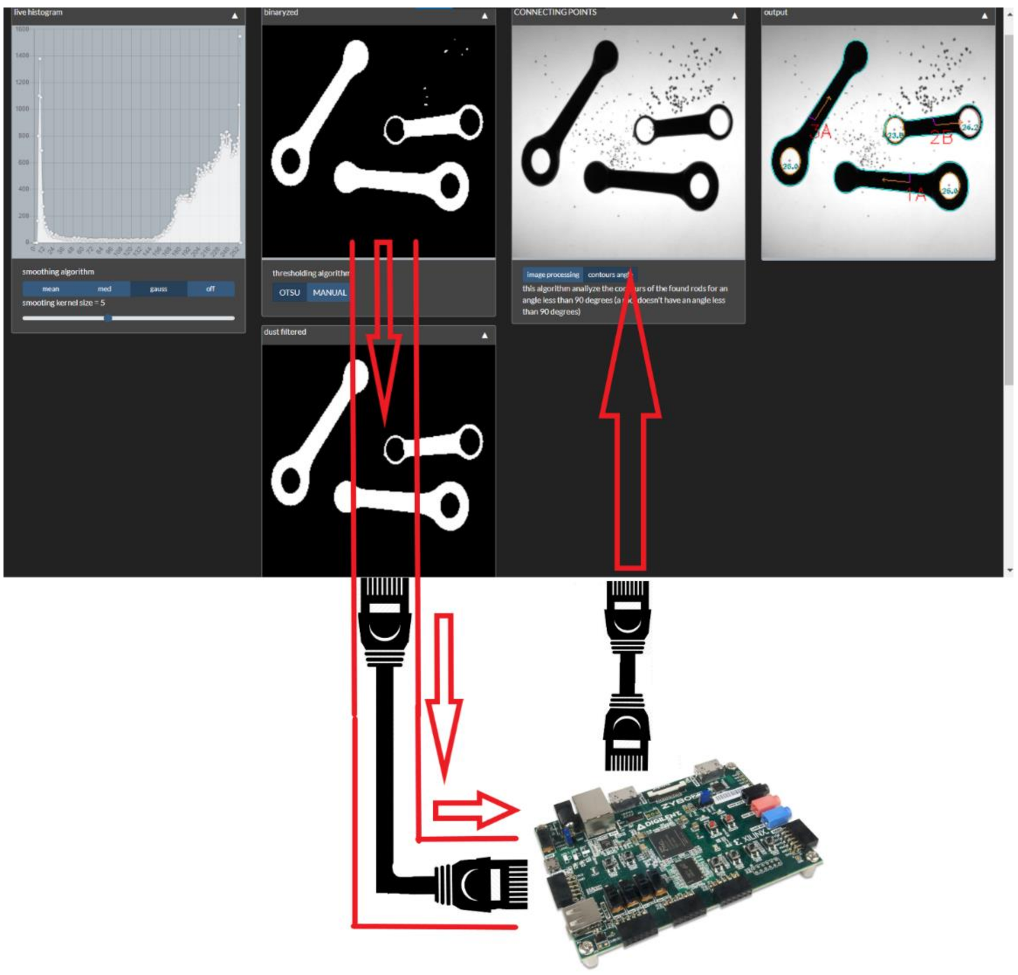


This stream will be generated by input video signal taken from an HDMI input (more suitable for the final product), or will be written by the PS layer of the FPGA that can receive a video stream from the Ethernet cable (useful for testing purposes). In the second case a DMA will be used.
Every processing step is implemented by an ip-core developed in vivado HLS: a language for describing hardware with a c-like high level language. No VHDL nor VERILOG languages had been used in this project.

The main processing step involved in this project (each one implemented by a separate ip-core) are:

1. The gaussian smoothing filter
2. The otsu thresholding finder filter
3. The binarization
4. Connected components labeling and final metadata extractor

Moving the entire pipeline into the FPGA has left some open problems that can be hardly overcome without random access over the image pixels: connected rods detaching. Even if the localization of critical connected points could be implemented with a FAST corner detection, “the drawing of a black line” between 2 pixels is hardly implementable. Because of this a different approach is also proposed: the FPGA could be used just for the image processing parts of the pipeline on the base of the image received by the python program (running on a normal computer and connected with an ethernet cable to the fpga, or directly in the arm processor of the soc board).

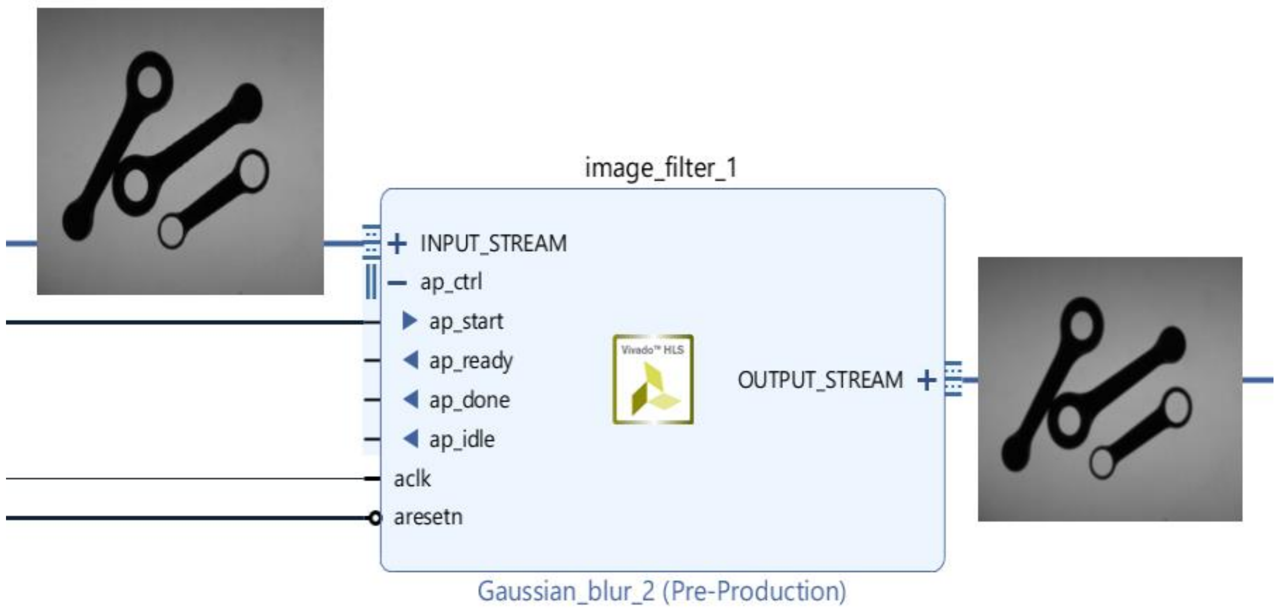


this algorithm anallyze the contours of the found rods for an angle less than 90 degrees (a rod doesn't have an angle less than 90 degrees)

Visual representation of fpga usage in case of image processing accelerator

Image processing ip cores

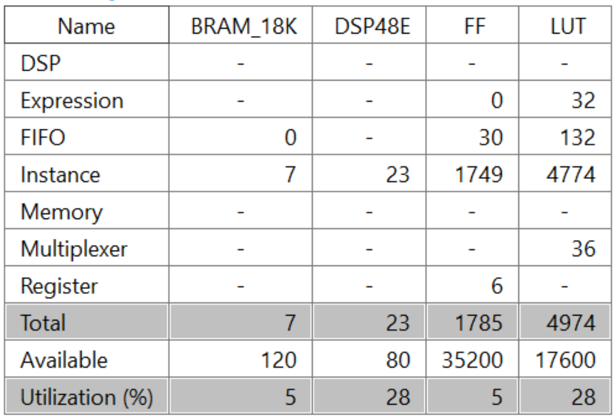
Gaussian filter ip-core



This Ip-core provides a gaussian blur smoothing with a kernel of 7x7. Is implemented with Vivado hls opencv gaussian blur function that works over the AxiVideoStream in dataflow mode. Is implemented with sliding buffer of required lines and the whole image is not stored inside the block ram. For sliding line buffering of pixels in BRAM please refer to (1).

```
void image_filter(AXI_STREAM& input, AXI_STREAM& output) {
#pragma HLS RESOURCE variable=input core=AXIS metadata="-bus_bundle INPUT_STREAM"
#pragma HLS RESOURCE variable=output core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> img_input(MAX_WIDTH, MAX_HEIGHT);
    hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> filtered_1(MAX_WIDTH, MAX_HEIGHT);
#pragma HLS DATAFLOW
    hls::AXIvideo2Mat(input, img_input);
    hls::GaussianBlur<7,7>(img_input, filtered_1);
    hls::Mat2AXIvideo(filtered_1, output); }
```

This will result in a synthetized design not so huge:



Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	32
FIFO	0	-	30	132
Instance	7	23	1749	4774
Memory	-	-	-	-
Multiplexer	-	-	-	36
Register	-	-	6	-
Total	7	23	1785	4974
Available	120	80	35200	17600
Utilization (%)	5	28	5	28

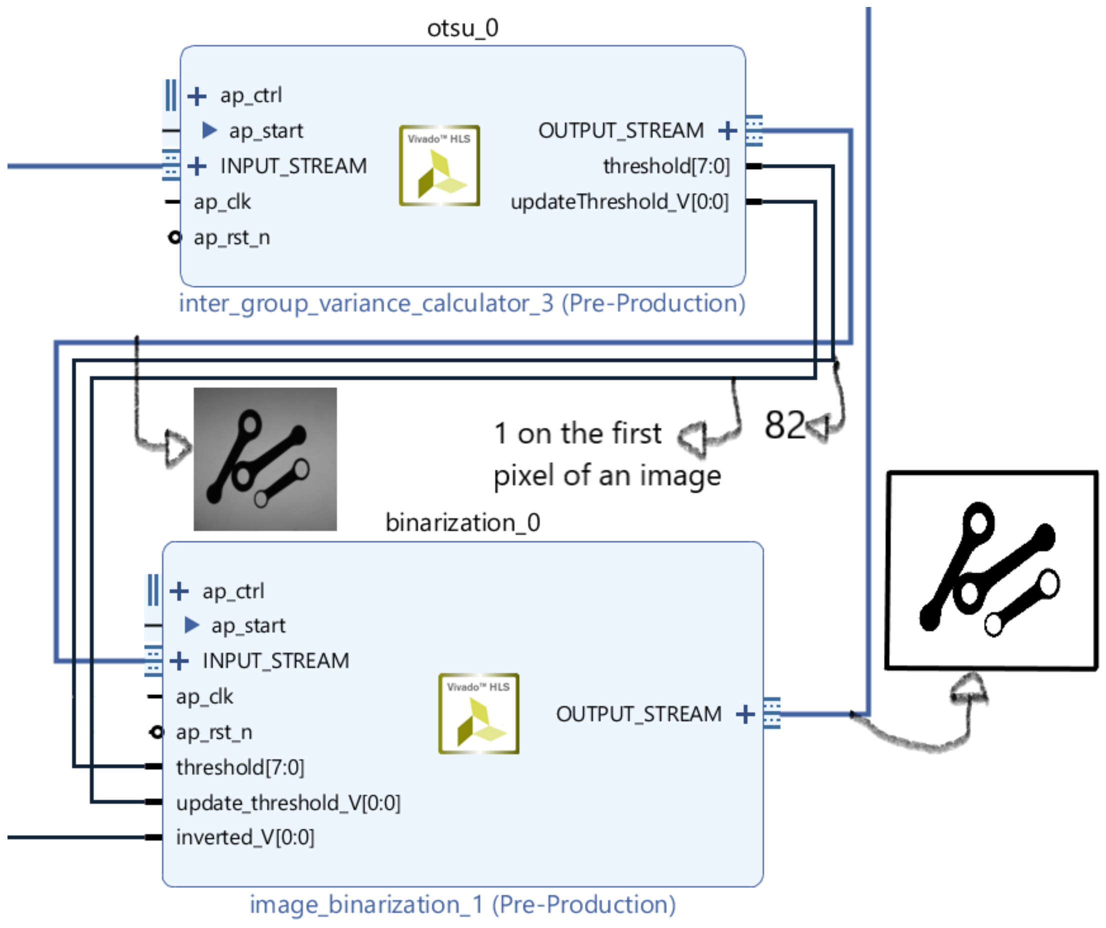
Resource utilization for this ip core referred to zybo-z7 10 XILINX FPGA

Otsu thresholding search and binarization

From the moment that the Otsu thresholding search require the entire photogram in order to compute any kind of variance based on the whole photogram, we have no possibility to compute the otsu thresholding and apply the binarization in the same stream.

One solution is of course to keep the whole picture in memory and delay the stream by the time of one picture, but we can’t afford this in an fpga. 2 alternative solutions had been explored:

1) Based on the consideration of the environment in which the system will work: a conveyor backlighted belt in which rods are moving from left to right, and based on the fps of the camera used, one solution could be the possibility to use the threshold computed for the previous photogram in the actual photogram.
2) The network can be adapted to work with photograms replicated 2 times in the stream: the first is for computing the threshold and the rest of the network would simply ignore the signal, in the second the threshold will be performed. The photograms replicated 2 times in the stream can be achieved in different ways:
    a. A 30fps 60hz signal is used in the HDMI input signal (most camera use this frequency) = in this case we already have the same photogram 2 times in the stream.
    b. A 25fps 50hz interlaced signal is used in the HDMI input signal = in this case the computation of the variance over the even lines can be applied for the photogram of the even odd lines.
    c. In case the input signal is taken from the DMA in which the PS has wrote the photograms, a simple mod to the arm code can allow to stream each photogram twice.



Schema for the binarization steps: the binarized image is created with the threshold computed upon the previous image in the stream

Identification of the threshold with the usage of inter group variance

The algorithm for the identification of the best threshold for the image binarization compute the inter group variance of 253 values (0 and 255 grayscale values are excluded a priori) and find the maximum value (2). The computation of the inter group variance is lightweight with respect to the computation of intra group variance. We are going to explain the main logic of an HLS implemented ip core meanwhile we refers to the actual code.

Definition of function and variables

```
void otsu (AXI_STREAM &in_stream, AXI_STREAM &out_stream, unsigned char *threshold,
      ap_uint<1> *updateThreshold) {
//#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
#pragma HLS INTERFACE axis port=out_stream name=OUTPUT_STREAM
#pragma HLS INTERFACE axis port=in_stream name=INPUT_STREAM
    static unsigned width_count = 0;
    static unsigned height_count = 0;
    static long scores[256];
    static unsigned int values[256];

    AXI_VALUE aValue;
    unsigned char readedVal
    unsigned char outputVal;
    union {    unsigned char ival; char oval; } converter;
```

The 2 AXI_STREAM in the definition will create INPUT_STREAM and OUTPUT_STREAM on the ip core, the threshold will be a 8 bit output signal referring to the computed best threshold, and the updateThreshold will be to 1 when the threshold should be changed by the binarization ip core (is like a signal_valid). The static variables are initialized at the startup of the ip-core and will keep their values between each call. The c function will be hardware-implemented and will be called each time a new pixel is available on the stream and will be read from the stream.

```
in_stream.read(aValue);
```

width_count and height_count are used between each call for identifying the last pixel of an image. Whenever a new pixel is received, the count of the relative intensity is incremented.

```
values[aValue.data]++;
updateThreshold = false;
```

When the whole image is received the computation of the best threshold starts.

This parameters are needed:

- the global mean of the pixels values

for every of the 253 thresholds:

  - under threshold mean
  - over threshold mean
  - probability to belong to the over threshold group
  - probability to belong to the under threshold group

This because we are applying this formula for the inter group variance

$$\sigma(threshold) = \sum_{i=0}^{threshold} (u_1 - u)^2 p(i) \; + \sum_{i=threshold+1}^{255} (u_2 - u)^2 p(i)$$

The mean is computed once for all:

```
for(scoreIter = 0; scoreIter < 256; scoreIter++){
    sum = sum + (values[scoreIter] * scoreIter);
}
globalMean = sum/(ALTEZZA*LARGHEZZA);
```

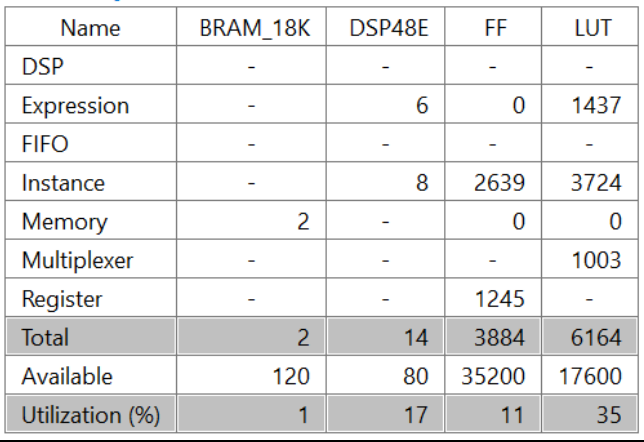
After this, the under threshold and over threshold mean are computed for every threshold between 1 and 254.

```
for(scoreIter = 0; scoreIter <= threshold; scoreIter++){
      sum_under_threshold = sum_under_threshold + (values[scoreIter] * scoreIter);
      count_pixels_under_threshold += values[scoreIter];
}
for(scoreIter = threshold + 1; scoreIter <= 255; scoreIter++){
      sum_over_threshold = sum_over_threshold + (values[scoreIter] * scoreIter);
      count_pixels_over_threshold += values[scoreIter];
}
if(count_pixels_over_threshold != 0 && count_pixels_under_threshold != 0){
      meanUnderThreshold = sum_under_threshold/count_pixels_under_threshold;
      meanOverThreshold = sum_over_threshold/count_pixels_over_threshold;
```

The final computation of the variance can be simplified with respect to the formula because in the end we are not interested in computing the real variance, but just a score for a good thresholding. First of all we don't consider the summation but we bring the sum of all the probability of every threshold value outside of the sum, and we don't consider the summation at all.

```
float prob_under_thresh = count_pixels_under_threshold / (float)(LARGHEZZA*ALTEZZA);
float prob_over_thresh = count_pixels_over_threshold / (float)(LARGHEZZA*ALTEZZA);
computedScore = probability_under_threshold *
      (meanUnderThreshold - globalMean) * (meanUnderThreshold - globalMean)  +
      probability_over_threshold *
      (meanOverThreshold - globalMean)*(meanOverThreshold - globalMean);
```

However, this would bring to an implementation in the FPGA that uses a huge space



Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	6	0	1437
FIFO	-	-	-	-
Instance	-	8	2639	3724
Memory	2	-	0	0
Multiplexer	-	-	-	1003
Register	-	-	1245	-
Total	2	14	3884	6164
Available	120	80	35200	17600
Utilization (%)	1	17	11	35

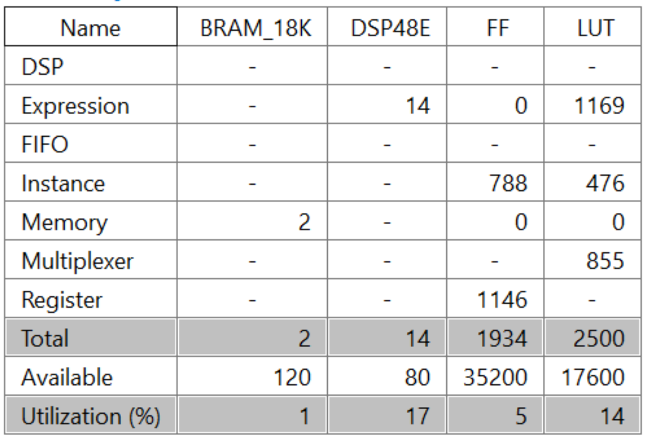
Resource utilization for the implementation of the best threshold finder with float values (huge LUT usage)

The algorithm can be simplified (we are just searching for the score of a threshold).

For the computation of the probability we can just use the count of the pixel and then rescale the final result (or even not!).

```
unsigned int probability_under_threshold = count_pixels_under_threshold;
unsigned int probability_over_threshold = count_pixels_over_threshold;
computedScore = (probability_under_threshold *
      (meanUnderThreshold - globalMean)*(meanUnderThreshold - globalMean)  +
            probability_over_threshold *
      (meanOverThreshold - globalMean)*(meanOverThreshold - globalMean))
      /(LARGHEZZA*ALTEZZA);
```

This would bring to a smaller size implementation that will find the same best thresholds as the previous one.



Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	14	0	1169
FIFO	-	-	-	-
Instance	-	-	788	476
Memory	2	-	0	0
Multiplexer	-	-	-	855
Register	-	-	1146	-
Total	2	14	1934	2500
Available	120	80	35200	17600
Utilization (%)	1	17	5	14

Resource utilization for the implementation of the best threshold finder with integer values

The performance gain given by an FPGA is due to the fact the all those loops are unrolled in hardware design.

In the end, a cycle for finding the maximum score for the various computedScore of every threshold is involved.

Binarization ip core

The core related to the binarization is really simple, with respect to the computed best threshold value, and with respect to an inverted parameter (that allow us to have the foreground white and the background black instead of the contrary), computes the binarization over every pixel flowing in the axi stream. The main problem is that we are working on a stream, so the threshold computed by the otsu ip core is used starting from the successive photogram.

……

```
        in_stream.read(aValue);
        if(inverted){
            if(aValue.data > threshold_memorized){
                aValue.data = 0;
            }else{
                aValue.data = 255;
            }
        }else{
            if(aValue.data > threshold_memorized){
                aValue.data = 255;
            }else{
                aValue.data = 0;
            }
        }
        out_stream.write(aValue);
```

……

The resource utilization of this ip core is of course small.

References

1) Otsu overview with inter class variance optimization explaination
2) Convolutions filters implemented in FPGA with the usage of “sliding line buffers”
   https://en.wikipedia.org/wiki/Otsu%27s_method#Improvements
3) https://amslaurea.unibo.it/11311/1/TESI_Mingarelli_Simone.pdf