FPGA based accelerator for visual features detection

François Brenot, Philippe Fillatreau, Jonathan Piat

FPGA based accelerator for visual features detection

Francois Brenot
LAAS/CNRS – Team : RAP
Toulouse – France
Email: fbrenot@laas.fr

Philippe Fillatreau
LGP/ENIT – Team : DIDS
Tarbes – France
Email: philippe.fillatreau@enit.fr

Jonathan Piat
LAAS/CNRS – Team : RAP
Toulouse – France
Email: jpiat@laas.fr

***Abstract*—In the context of obstacle detection and tracking for a vision-based ADAS (Advanced Driver Assistance System), one mandatory task is vehicle localization. Vision-based SLAM (Simultaneous Localization and Mapping) proposes to solve this problem by combining the estimation of the vehicle state (localisation : position and orientation) and an incremental modelling of the environment using a perception module (feature detection and matching) in images acquired using one camera or more. Such a perception module requires an important computational load that highly affects the latency and the throughput of the system. Our goal is to implement the SLAM functionality on a low power consumption mixed hardware and software architecture (using a co-design approach) based on a Xilinx Zynq FPGA. This device includes logic cells that allows to speed-up the perception tasks to meet the real-time constraint of an ADAS. In this paper, we present the implementation of two hardware components : a FAST (Features from Accelerated Segment Test) features detector and a parametrizable corner refinement module (Non Maxima Suppression - NMS).**

I. Introduction

Advanced Driver Assistance Systems (ADAS) like cruise control, automatic high beam switching or parking assistance systems, are now widely implemented on-board commercial car models. Vision based ADAS involve vision sensors like cameras, which provide rich information allowing the development of smarter systems.

Our works deal with vision-based ADAS aiming at obstacles detection and tracking. One basic and mandatory task undertakes the vehicle localization. An efficient technique for mobile robots or vehicles localization is known in the literature as SLAM (Simultaneous Localization And Mapping). For the navigation of a mobile robot or vehicle in an unknown environment, it allows to build incrementally a map of landmarks assumed to be static; matching the landmarks in successively acquired images allows to iteratively compute the positions and orientation of the landmarks in a global map, while estimating the position and orientation of the vehicle in this map. The SLAM process is divided into 2 main steps. The first one (perception task) allows to extract landmarks (known as features) in the acquired images. A second step allows to match the perceived interest points in successively acquired images; an estimation step allows to update the landmarks positions and uncertainties, and to estimate the robot position and its uncertainty in the incremental map. In the literature and in the former works lead at LAAS, the filtering step is classically based on the use on the Extended Kalman Filter.

In our works, we intend to embed the SLAM functionalities in an ADAS on-board a car. For such applications, strong processing times constraints have to be faced, to cope with potential high vehicle speeds. Our goal is to drastically improve processing times by integrating the SLAM process on a mixed hardware architecture including an ARM DualCore Cortex A9 processor and a hardware accelerator (both included in the Xilinx Zynq FPGA) to comply with the real-time constraints. In an ADAS context, the vehicle can reach $130km/h$. If the real-time constraints are set at $30Hz$ ($30fps$), the embedded system computes 1 image every 1.2 meters. We keep our real-time constraints in this range to meet ADAS's security requirement.

In this paper, we present the progress of these works. So far, we have designed a global hardware architecture for the SLAM process, and have defined a partitioning of the SLAM functionalities on the processor and on the hardware (FPGA logic cells), using a co-design approach. We also have defined, implemented and validated two hardware modules implemented on the FPGA. The first is a corner detector (certain kind of features) module, based on the use of the FAST detector (section III-A). The second is a non maxima suppression (NMS) module, allowing to perform corner refinement by selecting local maxima among the FAST features detected in the images (section III-B).

This paper is organised as follows. In section II, we present a state of the art of vision-based SLAM, of the corner detection and of the non maxima suppression functionalities, focusing on their integration on a dedicated hardware architecture. In section III, we describe the implementation architecture that we propose in order to partition the SLAM process functionalities on our mixed target architecture. In section III-A, we propose a hardware component architecture implementing the FAST features detection on FPGA. In section III-B, we describe our approach for the design of an original hardware architecture allowing to efficiently integrate a non maxima suppression functionality on FPGA ; the processing can be configured and parameterised, and offers excellent prospects towards the integration of a versatile FPGA-based module, allowing to perform non maxima suppression as well as regions labelling and segmentation or region filtering. Finally, we present and discuss experimental results in section IV, and summarise our works and the future steps in section V.

II. State of the art

A. Vision based SLAM

Vision based SLAM [1] allows a robot to navigate in an unknown environment; it allows to incrementally build a stochastic map of static landmarks from images acquired

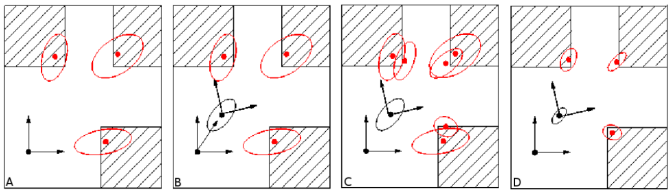

Fig. 1. Steps in the SLAM algorithm

from cameras. Matching the landmarks in successively acquired images allows to iteratively compute the positions and orientations of landmarks in a global map, while estimating the position and orientation of the vehicle in this map. A perception step allows to extract features in the acquired images. A filtering step allows to match interest points in successively acquired images, to update the landmarks positions and orientations and their uncertainties, and to estimate the robot position and orientation and their uncertainties in the incremental map.

Figure 1 illustrates the different steps of the SLAM process. The robot is represented by an Euclidean coordinate system and obstacles by hatched squares : (A) The robot observes three landmarks for the first time (the corners of the squared obstacles) together with the uncertainties of their positions, represented by ellipses (the depth is less precise than the width and the height) ; (B) the robot moves and predicts its new position ; (C) the robot predicts the landmark position and compares with the observation ; (D) merging this observation(s) with the map constructed previously reduces both robot and landmarks positions uncertainties.

In the literature, three different SLAM perception methods associated to the use of three different sensors systems can be found: stereocam-SLAM, bicam-SLAM and monocam-SLAM. Stereocam-SLAM uses a stereovision to acquire 3D landmarks. The Bicam-SLAM architecture combines two monocular cameras which allows to merge two 2D images acquisitions (one per camera). Monocam-SLAM uses a Monocular camera which acquires 2D landmarks. Our goal is to implement a Monocular-SLAM to provide a real-time and embedded system. A first system was presented in [2] and further developed and implemented at Laas [3].

B. *Features detection for SLAM*

In the perception step of the SLAM process, one of the goals is to detect features.

One of the most relevant types of features is corners defined as an intersection of two edges. In image processing applications, corner detection has become a well known method to extract stable features in images. In this field of research, two of the most commonly used algorithms are Shi-Tomasi [4] or SIFT (Scale-Invariant Feature Transform) [5] (Based on Laplacian of Gaussian algorithm). Another widely used corner detector in the literature and in the former works at LAAS is the Harris detector [6]. It is an efficient features detector according to [8] : this detector is usually used for SLAM because of its detection efficiency, its computational speed and its good repeatability.

Extracting features in an image involves to process all the pixels. Considering the huge amount of data to be processed, it makes sense to implement a hardware module to speed up this task. In this way, the corresponding hardware module processes input pixels and provides processing results of a small amount of a higher abstraction level data (relevant features in the image). Using a FPGA as the target processor allows to extract those features in high resolution images and with high processing time performances. But in [7], Berry shows that the Harris detector, due to its computational complexity, uses a lot of hardware resources, and the processing times performances reached are insufficient for our application.

One corner extraction algorithm suitable for a hardware (FPGA) implementation is FAST. It uses basics operations such as additions and comparisons, which are very efficient on hardware (in terms of resource consuming and performance). This algorithm is presented in [9] and reviewed in [8]. This method is implemented in [11] on a FPGA.

In the SLAM context, the detected corners have to be as accurate as possible to have a better localization. The result of a corner detector algorithms often leads to the detection of points clusters for each interest point in the scene. Each cluster has to be refined to find one pixel which represent the corner, classically by using a non maxima suppression algorithm.

C. *Non Maxima suppression*

Review [10] present different approaches to perform a NMS. They have been validated through a software implementation on a standard microprocessor on a standard workstation. In all cases, the whole image is explored in a global process (at least one loop to test all the image pixels).

In hardware implementation, the corner refinement module has to be achieved by using a processing pipeline fed by a pixel streaming. In order to use as less memory as possible the algorithm has to compute the non maxima suppression while storing a limited number of data. In [7], the authors present a method using a 2 pixels x 2 pixels shifting window which keep the top left detected feature when several are detected at one corner. The method presented requires to store 2 lines of pixel (to create the shifting window) and uses a FIFO per image column, storing a total of 512 (image width in pixels) x 2 (lines) x 1 bytes (pixel size).

III. Design flow - Embedded SLAM architecture

Our works aim at embedding a vision-based EKF-SLAM process in a real-time system, in the context of an ADAS. Such an embedded platform requires low power consumption and high computational performances. In order to do so, we propose to integrate the SLAM process on a mixed hardware architecture involving a microprocessor and hardware cells (both included on Zynq FPGA).

The perception tasks involves the processing of a large amount of pixels while the EKF filtering step involves more complex processings on sparser information. This EKF tasks requires a huge amount of DSP blocks and memories.

Our problem is thus to partition the functionalities between both processors. To deal with these constraints, and to propose an efficient partitioning, we use a co-design approach. Thus,

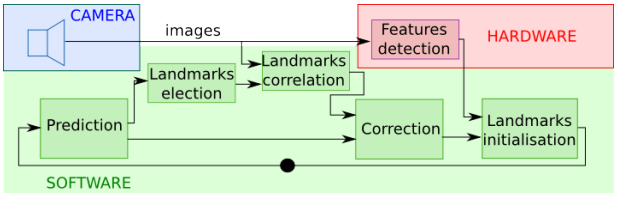

Fig. 2. Generic architecture to solve the SLAM problem

the FPGA is used to accelerate the perception tasks to release some bottlenecks in the microprocessor computation as : corner detection, distortion correction, NMS, ... The hardware architecture allows to speed up the processing times to meet our ADAS real-time constraints.

Figure 2 shows a general view of a generic SLAM architecture realising this partitioning. It is divided into two parts. The front-end which includes the perception tasks in red and the back-end which concerns the EKF filtering stepin green on figure 2.

The vision front-end is implemented on a FPGA. It includes, among others, the corner detector and NMS modules which are described later in this article. It computes high level features which feed the back-end EKF step.

The back-end EKF algorithm is running on a microprocessor.

To accelerate the perception task, we first propose to use an implementation of a corner detector : FAST [11].

A. *Design of a hardware architecture for FAST corner detector*

The FAST corner detection is composed of two main steps. The first one computes the corner score of the current candidate (pixel), named pixel p in figure 3. The second step is the corner validation task. It uses a 16 pixels Bresenham circle to classify whether a candidate pixel p is actually a corner (see figure 3). The pixels on the circle are identified using from numbers 1 to 16 clockwise. The corner validation step compares each pixel of the circle with the current (central) pixel p. If a set of N contiguous pixels in the circle are all brighter than the candidate pixel p (denoted by Ip) plus a threshold (for instance $t = 10$) value t or all darker than the candidate pixel p minus threshold value t, then p is classified as corner.

Figure 3 shows an example of a corner with the Bresenham circle.

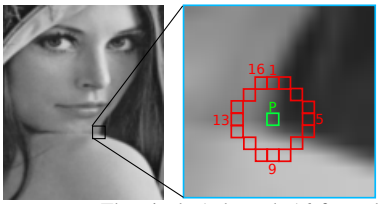

Fig. 3. Fast segment test. The pixels 1 through 16 form the circle

In the example given on figure 3 pixels 1 to 5 pixels have the same value that pixel p. Pixels 6 to 16 are brighter than p. As they are all contiguous, it means than the pixel p is a corner and its score is validated.

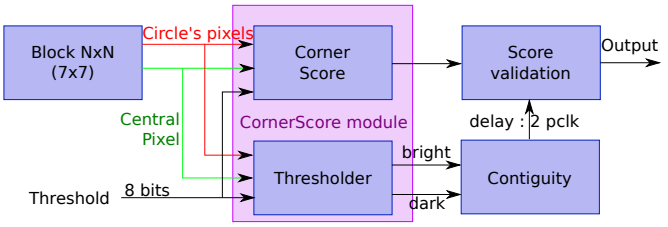

Fig. 4. FAST architecture

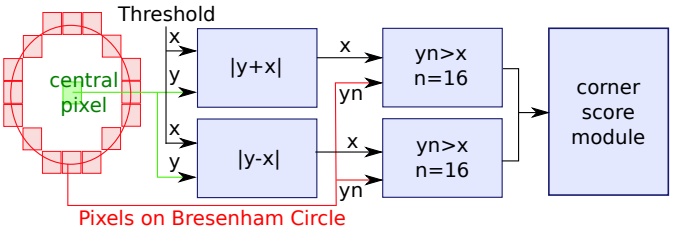

Fig. 5. Thresholder and comparator module

The corner score is computed according to equation 1 where t is a threshold, $I_{p\to x}$ is the intensity of the segment test pixel and I_p is the pixel's intensity under test.

$$V = max(\sum_{x \in Sbright} |I_{p\to x} - I_p| - t, \sum_{x \in SDark} |I_p - I_{p\to x}| - t) \quad (1)$$

The corner score function V, given in equation 1, is defined as the sum of absolute differences between the intensity of the central pixel and the intensities of the pixels on the Bresenham circle.

The FAST hardware implementation is composed of two modules, the corner score and the corner validation modules. Both can be computed in parallel. Those modules are fed by the Bresenham circle stored in BRAM. A $N \times N$ pixels Bloc stores $(N-1)lines+(N-1)pixels$ to make the circle available for the FAST processing. Figure 4 shows the FAST hardware architecture.

The validation module computes a contiguity test by processing 8 comparisons. Thus, Eight comparators are used to test arcs of 9 contiguous pixels (a segment test). The 9 pixels input segment tests are given by $Inputs[1+m$ to $9+m]$ with $m \in [0$ to $7]$. The 8 comparisons are processed in parallel.

The architecture of the hardware implementation of the thresholder and comparator module is presented in figure 5, and figure 6 summarizes the hardware architecture of the corner score computation (according to equation 1).

The main hardware acceleration is provided by the adder tree instantiation which computes pipelined additions. It allows to perform, in our case, additions of 8 inputs per clock cycle instead of 8 clock cycles using a basic processor.

The proposed global hardware architecture for the FAST algorithm is generic. The circle size can be easily parameterized and the design can also be implemented on a Xilinx or Altera FPGA. We have set the size of the Bresenham circle

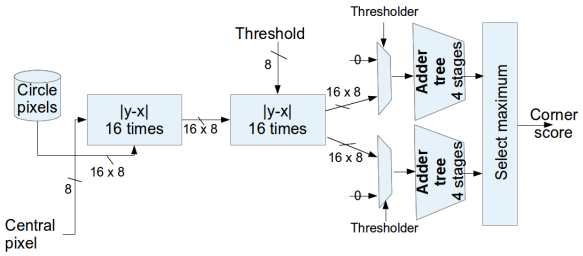

Fig. 6. Corner score module

to 16 pixels and the contiguity test to an arc of 9 pixels. This configuration is the most efficient one according to [9].

Due to its logic resources cost (in terms of memory and LUT consumption), the implementation also allows a parallelism by duplicating the architecture. Combined to image downscaling or circle parameterisation, it can provide a multi-scaling FAST corner detector. This can be a good way to solve the FAST non scale invariance problem.

B. Design of an hardware architecture for NMS

The output score image is composed by regions which correspond to each corner found. To refine those corners' locations, we propose an original hardware implementation of a NMS algorithm.

Our NMS algorithm can be performed considering a 4 connected or a 8 connected pixels connectivity. In this document, we present the 4-connected pixels connectivity configuration.

The small FPGA memory size obliges to compute algorithm on a pipelined architecture where lines are acquired one line at the time. Figure 7 presents the data flow in which P is the current pixel. We can see a first half of the image which was acquired previously and an unknown future part which will be sent by the camera later. NMS is performed upon data reception while keeping track of past computations.

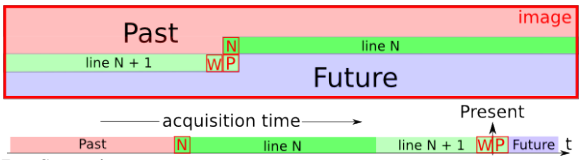

Fig. 7. Streaming process

Our pipelined NMS algorithm is based on State Machine (figure 9) and three main different cases can be found:

1) A new region is found, a new memory allocation is created which will store the maximum score of this region;
2) An already known region (defined as an unclosed region) is found , the maximum of the previous line in this region is read from the memory and is updated if needed;
3) A known region is closed, the maximum can be released.

Figure 10 shows an overview of our proposed hardware NMS architecture. Two modules can be seen : a "memory manager" and a "maximum comparator".

1) Maximum comparator: This module allows to detect regions (in our case the FAST scores image) and test each pixel of a given region to find the maximum. it relies on a pattern of 3 pixels, shown in figure 8. Where, N is the maximum found in the previous lines of the current region. C is the current score send by the camera. W is the maximum score of the whole known region (including the previous lines). Those scores are compared and the result is pushed to W.

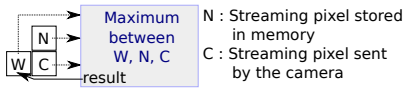

Fig. 8. How the NMS pattern of comparison is fed

Figure 9 shows the state machine of this NMS algorithm.

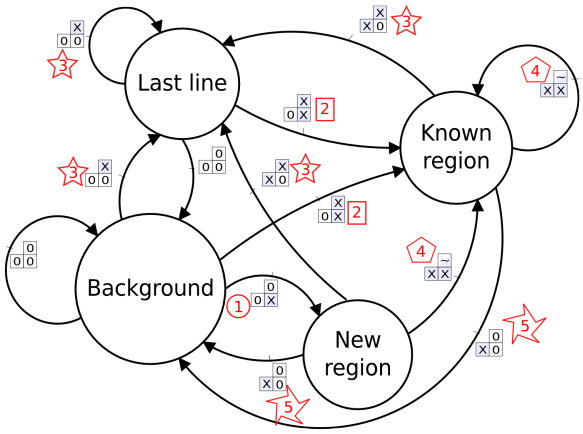

Fig. 9. State Machine

5 main cases determine the state's transition conditions (refer to red numbers in figure 9 and letters in figure 8):

- Conditions 1 : $C \neq 0$ ($C = X$), C has a non-null score, means that the current score is a part of a new region. If this new area appears to belong to an already known region in a future shift, the oldest region will be merge with the newest one;

- Condition 2 : As $N \neq 0$ and $C \neq 0$, we can assume that the region has been discovered (opened) during the previous line. The highest score is kept and pushed to W;

- Conditions 3 : The pattern is closing the current region. During the process, a reference counter is used and updated to know how many scores are belonging to this region. This counter is decreased each time $N \neq 0$ and $C = 0$. If it falls to zero, the region is closed (exploration of the current region is over) and the definitive maximum is now known.

- Conditions 4 : This state shows that the pattern is in a known, unclosed, region. If N and C belong to different regions, both regions are merged, the oldest region (of N) is merged with the newest (of C);

- Condition 5 : This condition means that the maximum can be updated. N and C belong to different regions, both regions are merged

2) Memory manager: In order to explain the memory manager module, let I be a $w \times h$ sized image. The coordinates of the current pixel position in the image is given by (m, n) with m the row index and n the column index. (m, n) refers also to the current computed score named $I_{P(m,n)}$.

We have chosen to store some data to keep track of past computations. First, this module uses a memory (RAM_{score}) which stores the highest score of each region. It also stores the related (m, n) position. For each score, a label is stored by a second memory according to the region to which it belongs (it will be presented in details later). This second memory is called RAM_{index}. A third memory $ref\_counter$, which contains the region size, is a reference counter for each region.

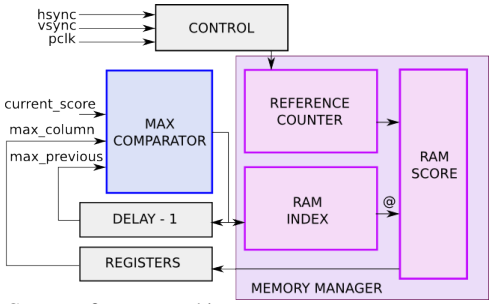

Fig. 10. Corner refinement architecture

Our architecture uses 2 memories. One pointers memory, RAM_index, which stores the region numbers (also known as labels). Those labels point to the second memory which stores maximum (RAM_{score}). Both shown in figure 10.

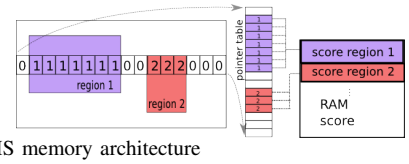

Fig. 11. NMS memory architecture

The RAM_index memory (in figure 10) stores an entire line of pointers as shown in figure 11. The column numbers of the image are the addresses of the memory in order to know in which region each pixel of the current line is. For instance, in figure 11, the region 1 is 7 pixels wide and region 2 is 3 pixels wide on this current line.

A reference counter ($ref\_counter$) is updated to know how many scores are belonging to this region. When it falls to zero, the memory allocation in RAM_score can be released, the region has been fully explored.

Finally, both RAM_index and RAM_score memories lead to a latency. It takes 2 clock cycles to update a maximum. In order to be able to update and read this maximum in the next clock cycle, we implemented registers for feeding the W register of the pattern (Figure 8). During the score update, the memory and the registers are both updated.

Moreover, The NMS architecture can be easily parametrized to perform a 4 or 8-connected pixels connectivity Non Maxima Suppression. This functionality will only affect the comparison module. As shown in figure 12 it will compare the Northern, previous, current and North-eastern scores for

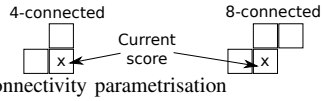

Fig. 12. Pixels connectivity parametrisation

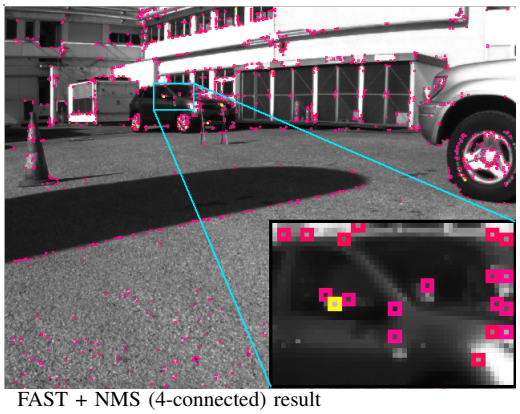

Fig. 13. FAST + NMS (4-connected) result

taking into account 8 neighbours (it will merge each related regions).

IV. Results

We presented the implementation of 2 perception modules in III-A and III-B. We first validated those architectures by using SystemC before describing them in VHDL.

SystemC has semantic similarities to VHDL, has a syntactical overhead compared to VHDL when used as a hardware description language. It offers a greater range of expression, similar to object-oriented design partitioning and template classes. It can be used for system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis.

The sequence of images used to validate our proposed hardware modules were acquired using a mobile robot navigating in an outdoor scene at LAAS. Figure 13 shows the results obtained after processing one of this images by our FAST and NMS hardware modules. The original image is visible in the background. Each detected corners appears as surrounded by a square (generated by an OpenCV software). The squares' colors depend on scores' values : from dark red to yellow for a very good feature. It shows that our architecture provides good corners despite the quality of the foreground (gravel surface).

On our FPGA target (Xilinx Zynq zc7z20), the processing times can reach the maximum frequency of 134 MHz. This means the algorithm can deal with a maximum of 134 Mpixels/s. For example, this bandwidth corresponds to 60 frames/second for a HD resolution image (1920*1080 pixels) with a maximum power consumption of 2.5 Watts (estimated by the Xilinx Xpower Analyser tool). Those results highlight the very good ratio computation/power consumption of an FPGA architecture on this application. Table I summarizes the slice logic utilization.

TABLE I. Zynq xc7z20-1clg484 - Logic Slices Utilization

Slice Logic Utilization	Used	Available	Utilization
Slice LUTs	5963	53200	11%
BRAM (18k/36K)	3/2	140 (38K)	2%
Slice registers	8281	106400	7%

Moreover, in [9], it is shown that FAST+NMS algorithms use 26.5% of a Pentium III 850MHz during 5.29ms to process a 768 x 288 image. Our implementation can handle this task to free up this host's processing time with a latency of only 4 pixels (thanks to the streaming pipeline).

Unlike the hardware implementation of FAST presented in [11], the non-maxima suppression presented here is not based on the storage of 5 lines using FIFOs. It stores the important information : the maximum score of each current known but unclosed regions (opened regions). The memory increasing corresponds to the storage of those unclosed regions. The wider is the image, the higher is the number of simultaneously unclosed regions. In the worst case, on one line, every other pixel belongs to different regions : $ImageWidth/2$ opened regions. The memory usage is processed as follow : $(ImageWidth/2) \times (@range\_score + @range\_posX + @range\_posY + @range\_indexTab)$ Where $@range\_$ is the range of the address vector.

Here is the one proposed by [11] by using FIFOs to store and compute 5×5 neighbourhood NMS : $(5\times ImageWidth)\times @range\_score$

Our implementation is even more efficient when a descriptor is associated to the detected corner. The descriptor output is stored in the RAM_score memory like the corner score, the x and y positions. If the descriptor is BRIEF, a 128 bit vector is generated to describe the detected corner thanks to its neighbourhood. Figure 14 shows the memory consumption comparison between two architectures (FAST + BRIEF + NMS) : our architecture and another one based on [11]. For the same algorithm, our architecture needs 45% less RAM memory. Our implementation computes FAST, BRIEF and then NMS. To perform the same algorithms, implementation presented in [11] needs to compute FAST, NMS (with a storage of the intensity in FIFOs) and then BRIEF.

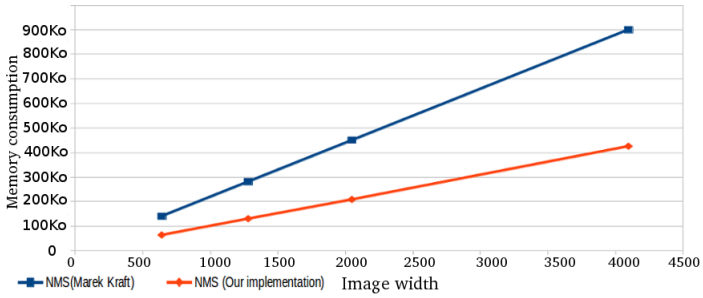

Fig. 14. Memory consumption comparison

By following the same logic, each informations related to the pixel (intensity, descriptor, RGB values, image scale...) can be stored in the RAM_score memory with a very small memory consumption cost.

V. Conclusion & Future Work

In this article, we proposed an efficient SLAM architecture partitioning. The perception task is computed by a hardware implementation while the EKF filtering is processed by a microprocessor. The hardware part deals with the processings involving large amount of data (pixels) while the software part undertakes the complex computations on sparser data.

We also presented 2 hardware architectures allowing to efficiently integrate a corner detector and non maxima suppression functionality on FPGA.

The first architecture presented is an implementation of the FAST algorithm. It will become a multi-scale module by duplicating the FAST architecture to involve several Bresenham circle size. Finally, we will add a dynamically adjusted threshold. All these future upgrades will allow to find better corners in order to increase the estimation precision and then the robot position.

The second proposed hardware module integrates an NMS processing. it will be upgraded to be versatile. With very small modifications, it can perform region filtering and labelisation. One configuration can perform region filtering. A simple pixel counter measures the area of each regions allows to removes all the smallest regions. The second one, labelisation, allows to assign a same label (number) to each related pixels which have homogeneous parameters. Afterwards, these object properties or object features can be associate to a classification algorithm.

Our future work will also concern the challenging SLAM implementation by integrating the FAST and NMS algorithms. This work will lead to the detection and obstacle tracking.

This work is related to DICTA, a project funded by the "Midi-Pyrenees" region, and involving the LAAS-CNRS, the LGP-ENIT and the DTSO (Delta Technologies Sud-Ouest) company. F. Brenot's PhD work is funded by the "Midi-Pyrenees" region and DTSO.

References

[1] Davison. Real-time simultaneous localisation and mapping with a single camera. pages 1403–1410 vol.2. IEEE, 2003.

[2] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. MonoSLAM: real-time single camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, June 2007.

[3] Cyril Roussillon, Aurélien Gonzalez, Joan Solà, Jean-Marie Codol, Nicolas Mansard, Simon Lacroix, and Michel Devy. RT-SLAM: a generic and real-time visual SLAM implementation. In *Computer Vision Systems*, volume 6962, pages 31–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[4] Jianbo Shi and Tomasi. Good features to track. pages 593–600. IEEE Comput. Soc. Press, 1994.

[5] David G. Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, page 1150–1157. Ieee, 1999.

[6] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[7] BIREM Merwan and BERRY Francois. Fpga-based real time extraction of visual features. In *Circuits and Systems (ISCAS)*, Seoul, South Korea, May 2012.

[8] Mohammad Awrangjeb, Guojun Lu, and Clive S. Fraser. Performance comparisons of contour-based corner detectors. *IEEE Transactions on Image Processing*, 21(9):4167–4179, September 2012.

[9] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 2, page 1508–1515. IEEE, 2005.

[10] A. Neubeck and L. Van Gool. Efficient non-maximum suppression. pages 850–855. IEEE, 2006.

[11] Marek Kraft, Adam Schmidt, and Andrzej J. Kasinski. High-speed image feature detection using FPGA implementation of fast algorithm. In *VISAPP (1)*, page 174–179, 2008.