Università di Pisa

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Elettronica

Reconfigurable FPGA Architecture for Computer Vision Application in Smart Camera Networks

Author:
Luca Maggiani

Supervisor:
Prof. Roberto Saletti
Dr. Paolo Pagano
Prof. François Berry

Abstract

Smart Camera Networks (SCNs) is nowadays an emerging research field which represents the natural evolution of centralized computer vision applications towards full distributed and pervasive systems. In this vision, one of the biggest effort is in the definition of a flexible and reconfigurable SCN node architecture able to remotely update the application parameter and the performed computer vision application at runtime. In this respect, we present a novel SCN node architecture based on a device in which a microcontroller manage all the network functionality as well as the remote configuration, while an FPGA implements all the necessary module of a full computer vision pipeline. In this work the envisioned architecture is first detailed in general terms, then a real implementation is presented to show the feasibility and the benefits of the proposed solution. Finally, performance evaluation results underline the potential of an hardware software codesign approach in reaching flexibility and reduced processing time.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor Prof. Roberto Saletti for all I have learned from him and for his help and support during this thesis. I would like to express my gratitude to Dr. Paolo Pagano, who gave me the opportunity to work in collaboration with Scuola Superiore Sant'Anna. I am also thankful to him for his constant encouragement and for giving me advices and future research prospects. I am grateful and indebted to my co-supervisor, Prof. François Berry, for his expert, sincere and valuable guidance extended to me.

I take this opportunity to give my sincere thanks to all of the Networks of Embedded System people, for their continuous assistance during the development of the architecture prototype. In particular my appreciation to Claudio Salvadori who has been always there to listen and give me advices. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work.

Finally, my sense of gratitude to one and all who, directly or indirectly, have helped and supported me in this venture.

Contents

Chapter 1

Introduction

Smart Camera Network (SCN) is nowadays an emerging research field promoting the natural evolution of centralized computer vision applications towards full distributed and pervasive systems. Differently from Wireless Sensor Networks (WSN) which are generally supposed to perform basic sensing tasks, SCNs consist of autonomous devices (Embedded Vision Systems), performing collaborative applications, leveraging on-board image processing algorithms optimized in respect of the limited available resources. [1]

SCNs are therefore built around high-performance on-board computing and communication infrastructure, combining video sensing, processing, and communications into a single embedded device, thus enabling more challenging applications such as visual control, surveillance, and tracking.

Such vision systems are more than electronic devices, as many people perceive it. They need advanced electronics design but also software engineering, and proper networking service design and implementation. They are built following design challenges of size, weight, cost, power consumption, and limited resources in terms of computing, memory, networking capabilities. At the same time, more and more services are required by customer specifications, so that time-to-market is setting a major challenge for architects on the design of effective and powerful solutions.

From the research perspective, the European academia is also committing to accomplish a seamless integration of SCNs into the next generation networks, the Future Internet, to match emerging societal needs. Typical scenarios for deploying traditional smart cameras are applications in well-defined environments, with static sensor setups, as traffic surveillance, intelligent transport systems, human action recognition.

Thus, the main challenges on SCNs refer to the real-time processing constraints (e.g. the capability to process video frame at 25-30 fps, as the human eye cut-off frequency) set by the target application and hardware architecture. Therefore, when considering a SCN of low-end devices, a major task is that of porting complex PC-based computer vision algorithms to embedded devices. These tasks (run at the logic level) should co-exist with control and configuration tasks, targeting to fulfill flexibility and reconfigurability requirements and aimed at reducing efforts for ordinary operation and maintenance.

The decreasing cost and increasing performance of embedded smart camera systems makes it attractive to the above mentioned applications. The main research focus is directed at designing a system with sufficient com- puting power to execute well-known image processing algorithm in real-time. Typical scenarios for deploying traditional smart cameras are applications in well-defined environments, with static sensor setups, as traffic surveillance, intelligent transport systems, human action recognition [2] [3].

In this vision, we can imagine a future SCN pervasively monitoring and controlling many activities in our daily life. Small and reconfigurable smart cam- eras create a dynamic network, which can be used for a wide range of applications. Notably in smart cities:

accident detection whenever an accident occurs, the system automatically detects a threat situation and takes specific actions to provide final users by the needed informations.

smart traffic surveillance able to regulate the amount of vehicles over a specific road and determine the best traffic light configuration on the basis of real-time traffic measurement.

pedestrian detection whenever a pedestrian is detected an alert is propagated to incoming vehicles.

Similarly, we could use such a system to monitor the movements of elderly in their homes in order to improve their quality of care [4].

Another important aspect in designing a SCN is obviously the network layer. Since SCNs are focused for a real environment, a low power, low rate wireless network has been deployed for distributed applications. Such a low speed medium makes a centralised data processing infeasible [5]. Indeed the amount of local data produced by these devices will not be handled by the wireless protocol. Smart

camera networks represent a particular challenge in this regard, partly because of the amount of data produced by each camera, but also because many high level vision algorithms require data from more than one camera.

Many distributed algorithms exist that work locally to produce results from a collection of nodes, but as this number grows the algorithm's performance is quickly crippled by the resulting exponential increase in communication overhead.

In this thesis we propose and implement a new node architecture design for a general purpose Smart Camera, and demonstrates feasibility of a configurable hardware platform based on a FPGA device. This allows the formation of a dynamic reconfigurable structure, in which many computer vision pipelines can be implemented.

1.1 Thesis outline

The remainder of this document is organized as follows: in Chapter 2 we introduce and discuss the image processing tasks on embedded platforms, and in particular on FPGA devices. In Chapter 3 we describe the state-of-the-art solutions for reconfigurable architectures with respect to the image processing environments. In Chapter 4 our architecture proposal is shown, starting from the hardware design and finally considering the implementation for a SCN node. In Chapter 5 a collection of hardware modules is presented. In Chapter 6 an experimental test run on a FPGA board and shows some results eligible to be extended to a real environment application and finally in 7 we shortly summarize the results of this thesis and the proposal for a possible future work.

Chapter 2

Image processing on embedded system

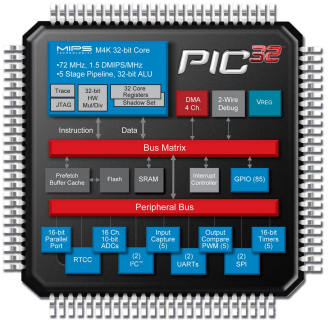
Smart cameras have been focused by the research community since the 1990s, where the first embedded devices showed up in the market. The limited computational resources available in those years were limiting real deployments of this kind of sensors. In the meantime, we have experienced a rapid growth of computational capabilities in embedded devices together with a sensitive decrease in the unit cost. This allowed to provide concreteness to applications based on smart cameras, thus triggering a special interest from the industry.

The specific implementation of a computer vision system is highly application dependent. However, for many vision systems, functionality can be classified as follows: in the first step, called image acquisition, an image (or more than one) is captured form a external sensors or cameras. Then the low level processing usually involves preprocessing such as noise and distortion reduction and certain important aspects of the imagery are emphasized. Then, in the intermediate level, the image is segmented. Typically, these segments are blobs, edges, lines, corners, regions, etc. The segments are usually without a specific semantic, they are not objects or physical entities but they contain spatial, geometric, angle and other types of information. It is this intermediate information that can be analyzed to extract features and information on the context.

Finally, the high-level processing stage generates the final results of the system, e.g., the position of an object, the name of a person identified by a facial recognition system, if an object has passed an optical quality inspection system, and so on.

2.1 Problems and limitations

In designing pervasive SCNs based on low-end devices(say microcontrollers), one of the biggest efforts is the porting of complex PC-based computer vision algorithms to embedded devices, as described in [6] where a Gaussian Mixture Model is indeed implemented on a PIC32. Although real-time performances can be met at 25 fps for very simple algorithms, a complex computer vision pipeline (e.g. enabling recognition) is usually requiring more powerful hardware platforms.



In the above mentioned paper a well-known background subtraction algorithm has been optimised to achieve real-time performance in a low-cost microcontroller, with fixed point operation. Even though a great effort put in optimisation, a complete image processing pipeline with real-time constrain is not feasible in such a limited device.

Thus, the main challenges on image processing on embedded devices refer to the high-rate processing constraints set by the target application and hardware architecture. Therefore, when considering a SCN of low-end devices, a major task is that of porting complex PC-based computer vision algorithms to embedded devices. Moreover image processing routines require ad-hoc hardware solutions, and careful power consuming evaluation, that make the hardware and software design more complex than other PC-based solutions.

The solution for embedded computer vision application are currently separated by different level of hardware abstraction: at low level there are the programmable logic devices represented by the FPGAs, then the Digital Signal Processors (DSP) which represent an ASIC-like solution and at the higher level the Mobile PC processor. Compared to high-end DSPs, FPGAs are more expensive, the design usually require more time and knowledge but provide a slower processing power due to the lower clock frequency of FPGAs. Mobile PC processor are commonly used in mobile computer, smartphone and netbook. They are meant for general purpose application but are easily deployed for computer vision tasks, especially

for the early stage of development.

Keeping in mind that a choise on a particular architecture brings different develompent effort, there are some premises [7] that leads towards an architecture rather than another:

- Highly repetitive processes applied to an image stream are more suitable to be implemented in hardware
- Algorithms which are independent from applications and potential subject of design reuse are more cost efficient to be implemented in hardware

Another important issue in the SCNs scenario is the realisation of a flexible and reconfigurable node architecture able to update the application parameter and/or change the target application at run-time. This aspect arise from the context where SCNs are deployed. As mentioned in the Chapter 1 SCNs are designed to be pervasive and represent an ubiquitous processing network composed by several nodes. Thus, a real smart camera node has to provide a set of possible operations, e.g. different computer vision algorithms or different communication methods, without any human actions.

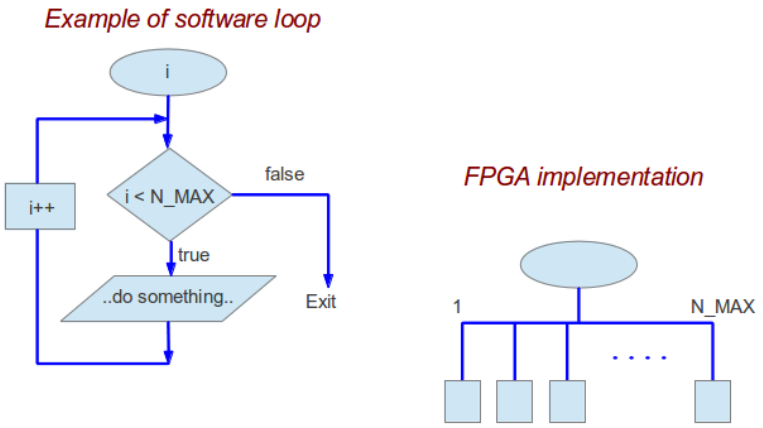
This perspective will be handled in Chapter 3 where the state of the art about reconfigurable platform is shown. In the next section there will be an evaluation of the previously exposed solutions, to explain how an altenative should be made to obtain the best results, in terms of power consuming, elaboration throughput, data latency and also accuracy [8].

2.2 Benchmark metrics

Typical image processing performance indicators are accuracy, robustness, sensitivity and efficiency [7] [8]. Based on those indexes the advantages and disadvantages of DSP, FPGA, CPU solutions are highlighted. Since most of the target applications have real-time constraints, the timing performances are extremely important to the system designer. In particular, for performance evaluation will be considered: (i) processing time, namely the time spent for a single pixel operations, (ii) the delay between new input data and the corresponding results, called processing latency and (iii) the resource used to achieve those performances.

Especially for FPGA comparison, the data latency has an important role to evaluate the optimisation level of a custom hardware IP block. Indeed with the inner parallelism that a FPGA solution provides, a single clock pixel operation is

possible. This is feasible through a pipeline elaboration, which introduces delay latency to the output results. The comparison below taking into account this aspect together with the maximum clock speed achiveable.



The comparison will be performed over well-know image processing algorithms, Sobel edge detector and SAD algorithm (Sum of Absolute Differences). The execution time per pixel is still dependent on the size of the image (in particular for CPUs), thus all tests were done on input images with equal size for all platform.

The data results for CPU and DSP were taken from [7], while FPGA comparison has significant updates from our development experience.

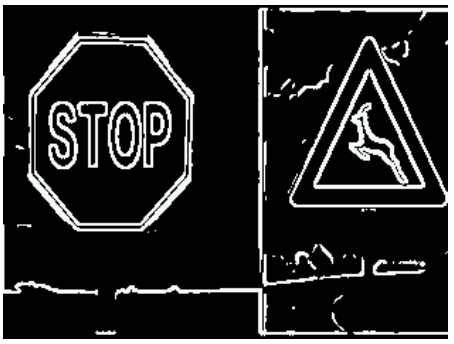
2.3 FPGA, DSP, CPU comparison

2.3.1 Sobel edge detector

The Sobel operator is widely used in image processing, particularly within edge detection algorithms. It is based on convolving the image with separated filter kernels in horizontal and vertical direction. Typically, the Sobel filter kernel consists of a pair of 3x3 coefficients, reported below.

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Region with high spatial frequency correspond to edges and reported in the binarized image as high values (typically 255 for 8bits pixel).

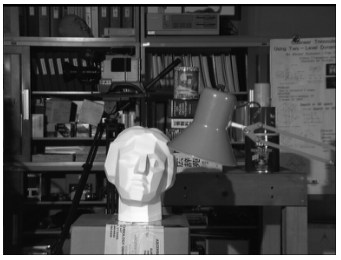


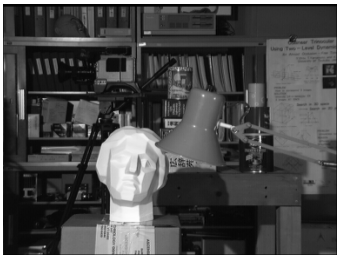
A magnitude is extracted from both vertical and horizontal gradients and then modulus are sum up to generate an index. This value is then compared with a threshold to estimate the edge presence.

2.3.2 Sum of Absolute Differences

The SAD is used to solve the correspondence problem between two image areas by calculating matching costs for blocks of pixels. In brief, the matching process involves computation of the similarity measure for each disparity value, followed by an aggregation and optimization step. Two stereo vision images, left and right, are compared using SAD algorithm to compute the dept map information. The quality of the result depends on the SAD block size (referred in Eq.2.1 as W). The mathematical operation is described in the equation below.

$$SAD(x,y) = \sum_{(i,j)\in W} \left| I\big(x,y\big) - I\big(x-i,y-j\big) \right| \tag{2.1}$$





2.4 FPGA implementation

FPGA technology has matured considerably over the past decade and special features and development tools have simplified the task of designing for FPGAs. FPGAs are targeted to the embedded devices where a hardware design is able to refine and upgrade a wide range of embedded designs with a limited development effort [9]. The processing power of an FPGA is directly proportional to the pro-

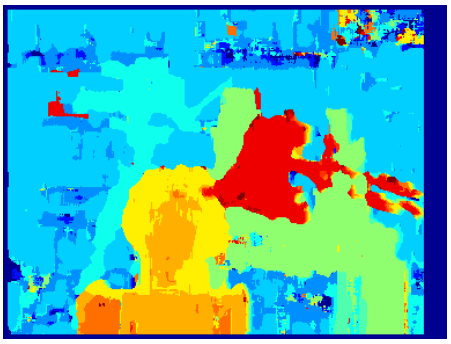


Figure 2.1: SAD disparity map over Tsukuba.

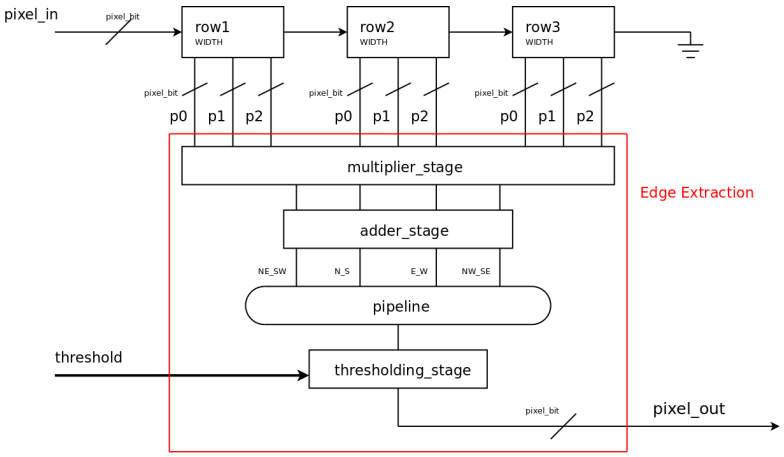


Figure 2.2: FPGA-based Sobel edge detector.

cessing capabilities of its logic blocks and the total number of logic blocks available in the array. More often, most commercial FPGAs have thousands of elaboration blocks (namely Logic Elements) which make programmable hardware a viable and efficient solution for accelerationg complex image-processing and computer vision applications [10] [11] [12].

Since FPGAs usually have limited on-board memory locations, optimised and memory aware image processing algorithms have been implemented. Typical applications use various on-board RAM blocks as distributed memory for parallel computations. Since a single RAM block has a separated control register, parallel operation can be performed on a set of RAM cells.

2.4.1 Sobel edge detector

In Fig. 2.2 an optimised FPGA circuit for Sobel edge detector is shown. The filter is composed by several multiplier stages (one for each Sobel kernel coefficients) which are feeded by a Multi-Row buffer. This buffer is made by two onboard RAM blocks, each one contains an image row. By using this buffering structure,

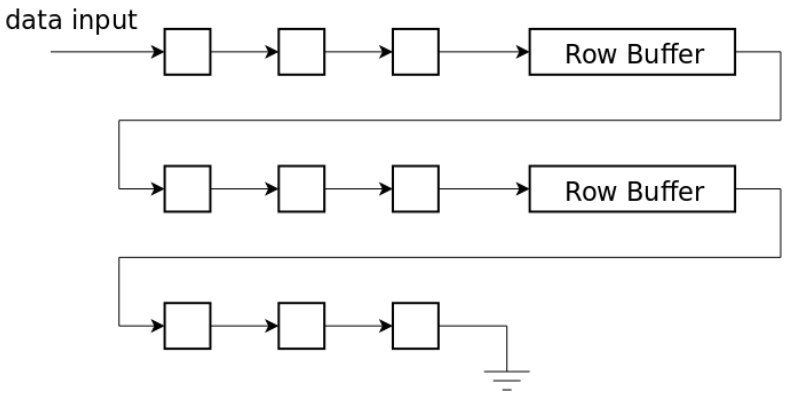


Figure 2.3: Sobel buffering schema.

Image Resolution	Delay latency	FPGA resource occupancy
QVGA (320x240)	4 T clk	$\sim$ 650 bytes + 1200 LEs (8%)

Table 2.1: Sobel hardware resources.

the circuit can handle a new data every clock cycle.

Fig.2.3 shows the internal buffering structure. The 3x3 matrix is used by the multipliers to compute the vertical and horizontal gradient values.

The above presented circuit is able to process a new input data every clock cycle, without computation delays. Thus introduces a constant latency output delay, defined by the pipeline nature of the architecture. The Sobel's timing and resource data is shown in Tab. 2.1. The percentages are referred to the Cyclone IV FPGA which has 22000 LEs and 144 DSP modules (9x9bits).

Applying the coefficient matrix, calculation of the absolute value and the final saturation is performed in one clock cycle. Hence, every time a new pixel is received from data in[7:0], a Sobel-filtered pixel is generated on the output interface signal data out[7:0], which results in a processing time for a single pixel (without considering any latency) of $t = \frac{1}{fCLK}$ where $fCLK$ is the clock frequency of the So- bel module. By using an Altera Cyclone IV EPCE22 FPGA a clock frequency of 77,7 MHz can be achieved.

As reported by [7], the processing time can be evaluated as:

$$t_{IMG} = T_{ROW} + \frac{1}{F_{CLK}} \cdot columns \cdot rows \tag{2.2}$$

The latency T_{ROW} between data in$[7{:}0]$ and data out$[7{:}0]$ is equal to the time needed to shift in a complete row plus two additional pixels into the Sobel module.

FPGA	LEs	M4K blocks	F_{clk}	$T_{LAT,IMG}$	$T_{LAT,LINE}$	$T_{SAD-block}$
EP2S120	75452	104 (17 %)	110 MHz	66 μ s	1,27 μ s	0,111 ns

Table 2.2: SAD algorithm resources related to the EP2C130 FPGA.

2.4.2 SAD algorithm

The SAD algoritm has been implemented by [7] for an Altera Stratix II EP2S130 using a 9x9 block size. The latency time is shown by two expression: $T_{LAT,IMG}$ and $T_{LAT,LINE}$. The first one is necessary on each image, while the other represents the latency over a single row of pixels. Both values are required to evaluate the system performance on a single disparity image. The SADs timing and resource data is shown in Tab. 2.2 for a 800x600 pixels image, with 100 Hardware SAD blocks.

2.5 CPU implementation

The Altera NIOSII CPU has been deployed for comparison. It is a 32-bit RISC CPU, with Data and Instruction Cache and a single precision Floation point co-processor. It runs at 100MHz and can achive up to 1,16 DMIPS MHz. Although is a softcore, the fastest version has similar performance than other commercial MCU.

2.5.1 Sobel edge detector

The Sobel filtering operation of an image requires multiple MAC operations on every pixel with a sliding window over the image. The filter coefficients are derived from an array which is accessed periodically. All coefficients are multiplied with the corresponding pixel intensity value.

```
                                                                    1
for (i = 1; i<( HEIGHT*WIDTH-1 ); i++) {                           2
        sobel_hor[i] = computeSobel_hor(&img[i]);                   3
        sobel_ver[i] = computeSobel_ver(&img[i]);                   4
        edge[i] = threshold(sobel_hor[i] + sobel_ver[i]);           5
}                                                                   6
```

In the algorithm presented above, the functions *compute_Sobel_hor*, *compute_Soblel_ver*, *threshold* are called for every pixel of the image. Both *compute_Soblel_hor* and

NiosII CPU	Operation per pixel	Memory footprint (Bytes)	fps
100MHz	24 (240ns)	3 · HEIGHT*WIDTH	up to 12

Table 2.3: The CPU Sobel performances.

compute_Soblel_ver perform 8 memory read cycles for coefficients retrieving, hence a high data bandwidth is requested.

A simple latency evaluation is reported on Eq. 2.3:

$$T_{IMG} = columns \cdot rows \cdot \frac{1}{fCLK} \cdot 8 \cdot T_{acc-mem} + T_{thresh} \tag{2.3}$$

where the image latency time is a linear function of the image resolution plus a algorithmic terms for computation. This formula does not consider a possible cache management, which could increase the memory access time. Morover care must be taken about the edge of a frame. Min and max operations are used to ensure that a memory address doesnt exceed the edges of a frame, thus leaving the address range.

2.5.2 SAD algorithm

Similar results can be obtained on SAD algorithm and other low-level operations because are based all on a convolutional arithmetic. Such repetitive tasks always require a great memory bandwidth to store the intermediate results and multiple clock periods for each iteration. Algorithm optimisations are available expecially when there is not data dependency between close pixel and the elaboration can be performed line-by-line. This is the case of SAD algorithms, where the images displacement can be evaluated over two lines at same time.

2.6 DSP implementation

In [7] DSP device has been deployed for convolution tasks. The authors used a Texas Instruments TMS320C6414T-1000 [13] DSP which runs with a clock of 1 GHz and provides up to 8,000 million MAC (multiply-accumulate) operations per second. By leveraging on the parallel ALU architecture, can produce four 16-bit multiply-accumulates (MACs) per cycle for a total of 4000 million MACs per second, or eight 8-bit MACs per cycle for a total of 8000 MMACS.

The TMS320C6414T device is based on a *Very Long Instruction Word* (VLIW) architecture (called VelociTI.2) developed by Texas Instruments, that makes these

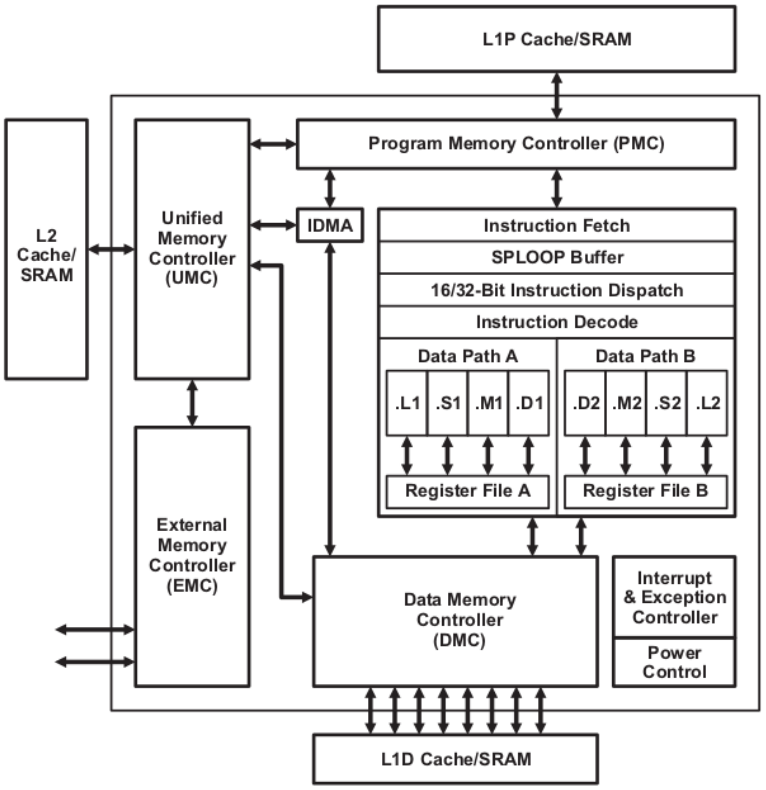


Figure 2.4: TMS320C64X architecture internal.

DSPs expecially targeted to the image processing tasks. With respect to a CPU-based approach, a DSP offers a higher ALU parallelization, extended data cache and vectorial instruction for heavy data processing.

By leveraging on the vectorial-based instruction, a DSP can process multiple input data in parallel using a dedicated memory management. For instance, by storing a multiple pixel lines into a multiport data memory, a 3x3 Sobel kernel computation can be archived in one clock cycle. In the Fig. the TMS320C64x architecture is shown. Rather than a General Purpose CPU (GP-CPU), this architecture provide two parallel data paths, each with four functional units. Each clock cycle this architecture handles up to 8 ALU operations, that result in a speeding elaboration by a factor of eight compared to a GP-CPU.

The TMS320C64X can gain execution time performance by deploying the VLIW operations. These operations are featured as an instruction-level-parallelism using the parallel elaboration data paths. Software functions on DSPs typically can have their performance improved by using specific intrinsics. Intrinsics are built-in functions which the compiler can directly translate into an assembler code. However, once intrinsics are used in the code it is not ANSI C-compliant anymore.

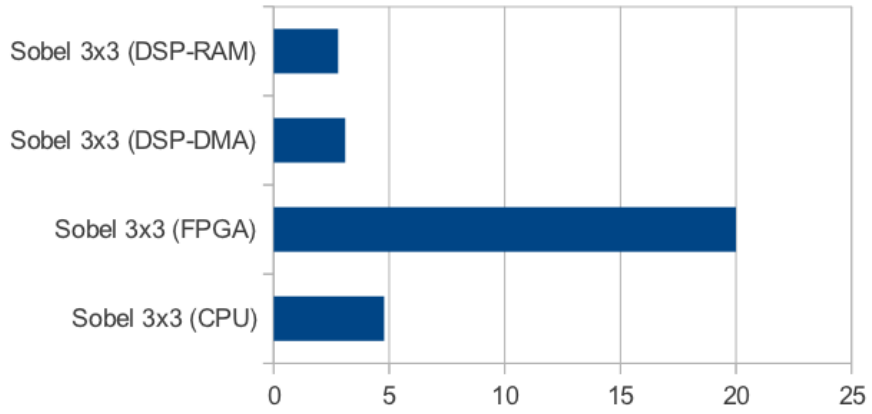


Figure 2.5: Sobel time performances (ns/pixel) [7].

2.7 Results

In Fig. 2.5 and Fig. 2.6 the performances for CPU, DSP and FPGA are shown. In Fig. 2.5 two implementation variants of DSP are presented. In the first one the DSP uses the internal RAM memory while in the other uses a DMA-like controller with an enternal memory chip. For the FPGA, performances are evaluated by a cycle accurate simulator and Altera Quartus tool for bitstream generation and static timing analysis.

The extreme differences between FPGA timing performances and the others are due to the different image processing schema deployed. Indeed low-level FPGA processing operation is done by directly receiving data form a camera. As long as the hardware algorithm is able to cope with the data bus speed, higher processing speed is not needed. Thus a speed up in the FPGA master clock does not make any sense since the camera is not able to deliver the pixel data anyway.

Basically, the FPGA time processing per pixel is equal to a period of the FPGA master clock. This constrast with the CPU implementation, where a higher clock frequency increases both computing performance and data transfer to the memory device.

In Fig. 2.6 the different implementations of SAD are exposed. Rather than the Sobel case, here FPGA performs considerably better than software solution. In this context, the low-level SAD processing operations are suited for a high parallelism and then ideal for FPGA implementation. In particular, the 9x9 SAD block match core is implemented as a row of one hundred cores, enabling a parallel match comparison within an entire line.

To sum up, in Fig. 2.7 a comparison between different architectures is shown.

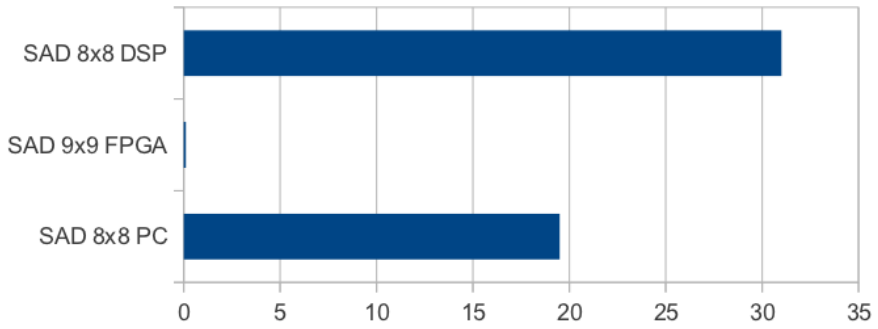


Figure 2.6: SAD time performances (ns/blockmatch) [7].

CPUs are generally meant for general application and indeed shown the high degree of programmability, as long as DSPs are focused on signal processing. By leveraging on the flexible nature of its internal architecture, the FPGA offers a trade off between full-custom ASIC design and a general purpose approach. Further, the FPGA low-level approach is particulary suited for image processing deployment, where thousands of repetitive operations are requested every seconds.

In this chapter the implementation of several low-level image processing algorithms were evaluated and compared. In particular, FPGA implementation outperform the CPU-based implementations when a large number of operations can be parallelised. As already reported by [15] and [9], the FPGA is well suited for low-level image processing tasks. This results are the starting point for our proposed architecture thath will be exposed on Chapter 4.

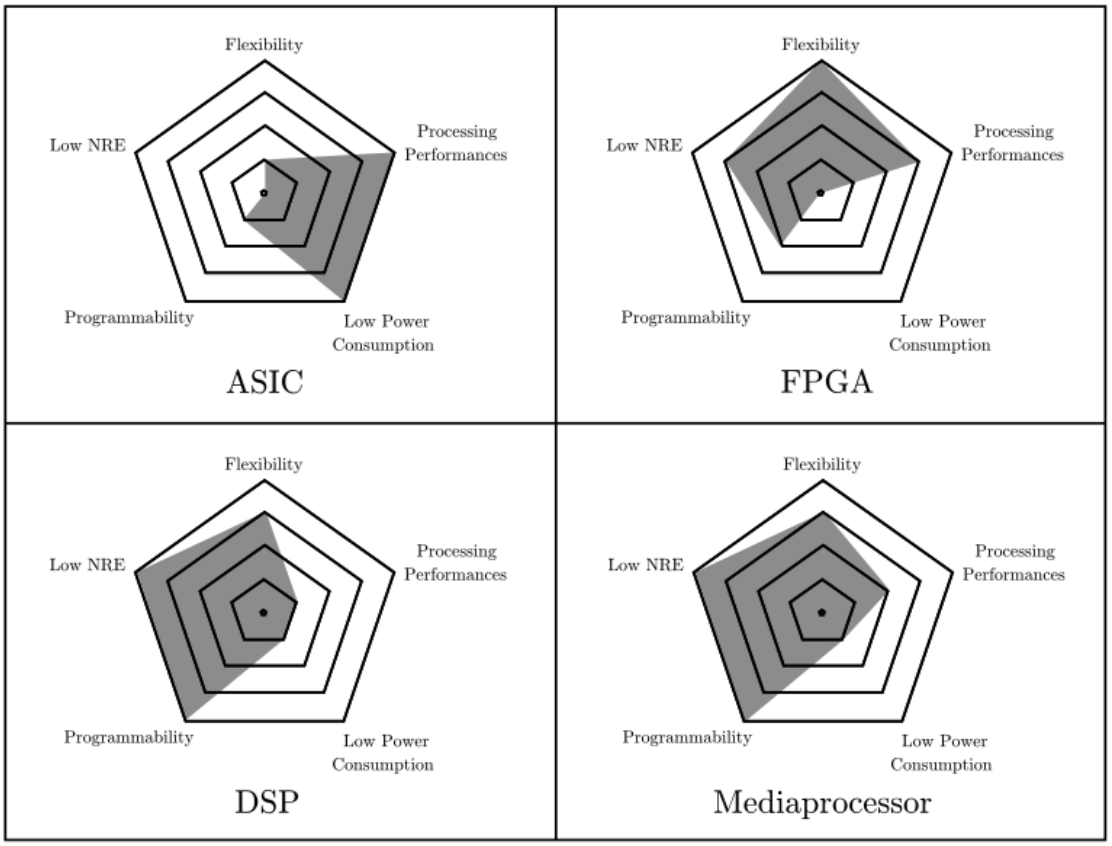


Figure 2.7: ASIC, FPGA, DSP, CPU comparison [14].

Chapter 3

State-of-the-art on reconfigurable architecture

As reported in Chapter 2, a complete computer vision pipeline is made up by a set of operations, which are computed sequentially on the image data. Even though the sequential nature of the computer video algorithms, the image processing phase is not well suited for general purpose CPU.

Low-level image processing such as color transformations and filtering operates on individual pixels in regular patterns. These low-level operations process the complete image data at the sensors frame rate, but typically offer a high data parallelism. On the other hand, high level image processing operates on a reduced set of features reducing the data bandwidth but increases the complexity of the operations significantly. This high level tasks often exibits strong data dependency thus programmable architecture and programmable processors are deployed in the state of the art solutions.

The rest of this Chapter will be organised as follow: at first three existing reconfigurable solutions are presented, either based on a reconfigurable processor architecture. Then different flexible datapath solutions will be shown, underlining the differences with the previous solutions.

3.1 Reconfigurable processor

3.1.1 ConvNet

In [16] and [17] a configurable processor is implemented on a FPGA device. This processor is specifically targeted towards the convolutional network, but can be

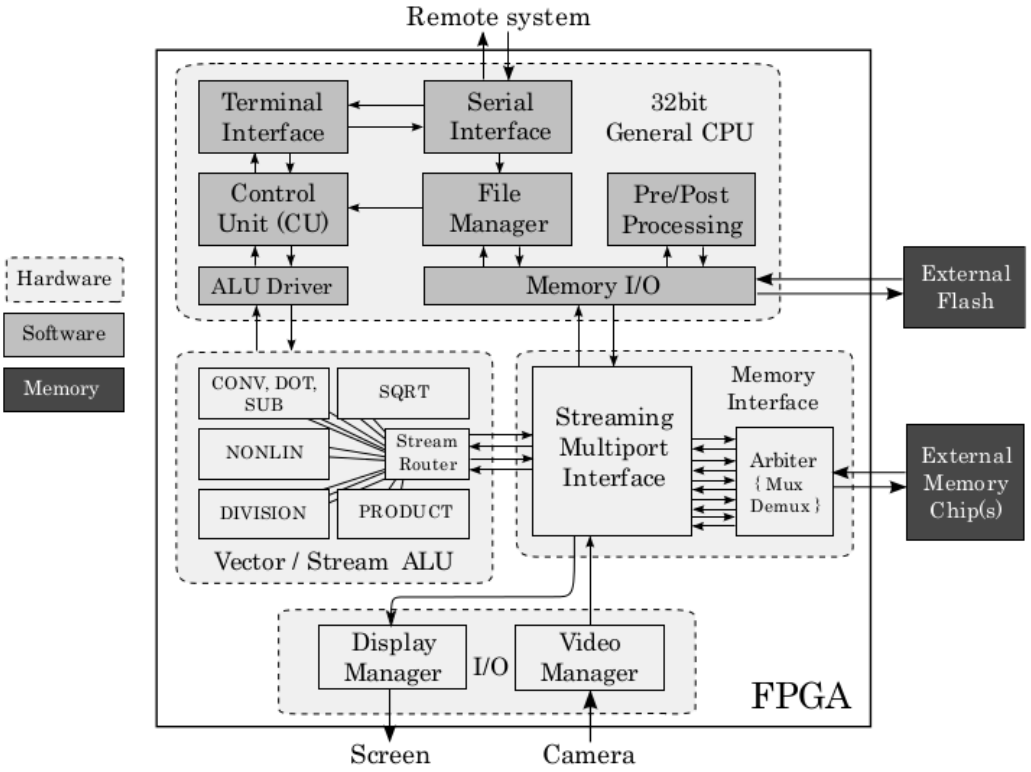


Figure 3.1: ConvNet architecture [16].

used also for other algorithms and tasks. Convolutional Networks (ConvNets) are feed-forward architectures composed of multiple layers of convolutional filters and non linear operations.

In Fig. 3.1 the internal architecture is shown. It embeds a 32bit RISC CPU and a configurable structure, which is able to manage the one or more continuous datastream. The second key component is the Vector/Stream ALU. All the basic operations of ConvNet are implemented at the hardware level and provided as SIMD instruction. The order between operations is managed by the softcore CPU, which acts as an instruction scheduler.

In [17] an updated version of the architecture has been presented by the authors. In Fig. 3.2 the reconfigurable neuFlow behaviour is shown. Every processing block is feeded by a DMA, which is controlled by the softcore CPU. The grid provides a flexible processing framework, due to the software reconfigurable connections. Indeed, any path can be configured on the grid an each operator uses FIFO buffers to compensate the processing speed variations between close blocks.

Both architectures have been implemented on a single Xilinx Virtex-4 FPGA and both can be run up to 200MHz.

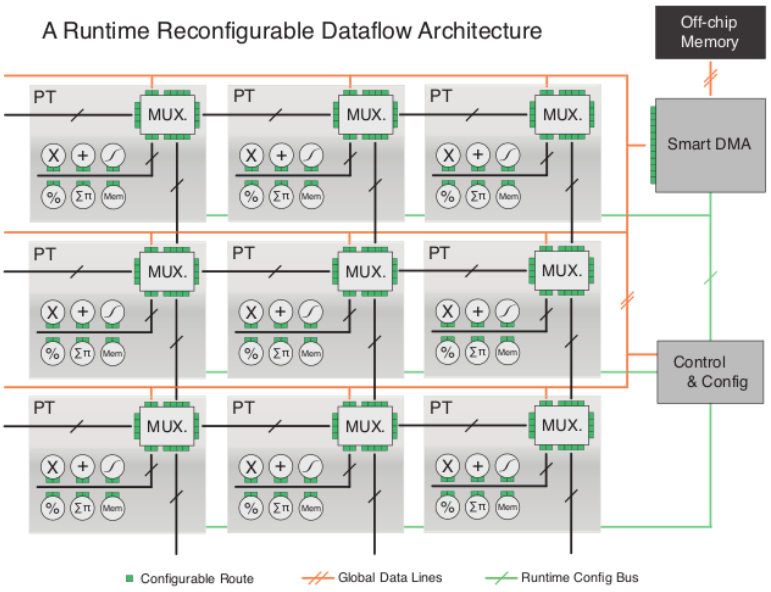


Figure 3.2: Reconfigurable neuFlow architecture [17].

3.1.2 Acadia

Sarnoff Corporation has developed Acadia [18], a custom CPU targeted to video processing tasks. The Processing Elements are connected to each other through a Crossbar switch, which acts as a data stream controller. Using such a structure, the processor is able to process up to 80 GOPS (Giga Operations per Second). Acadia processor can handle correlations, motion estimation, SAD algorithms and non-linear operation.

In Fig. 3.3 the second version of Acadia processor has been shown. Acadia II performs real-time contrast enhancement, stabilization, multi-sensor fusion, and tracking [19]. Equipped with ARM11, Acadia II functions as a CPU for the entire system and provide customized processing interfaces.

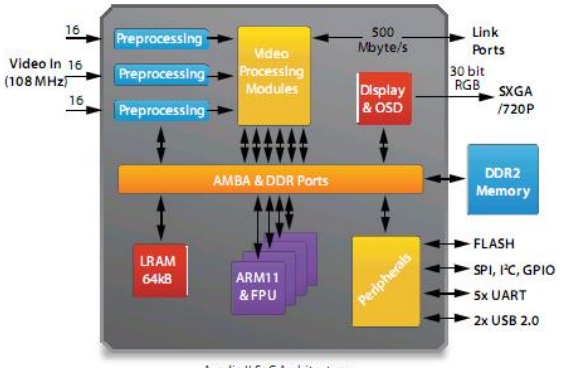


Figure 3.3: Acadia II SoC.

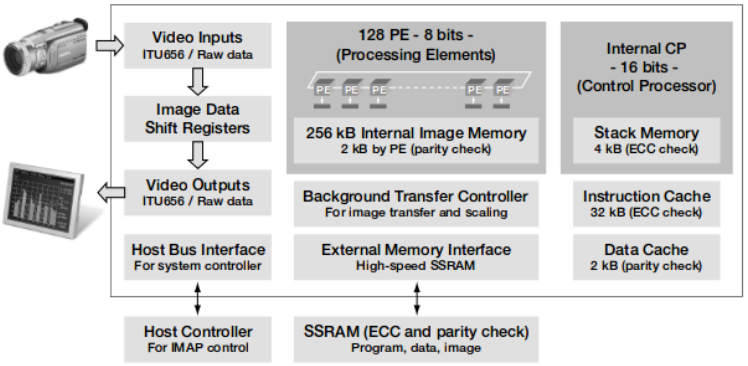


Figure 3.4: IMAPCAR architecture.

3.1.3 IMAPCAR

NEC Electronics has developed an SIMD processor called IMAPCAR, which stands for Integrated Memory Array Processor for CAR. This processor is designed specifically to be deployed as an in-vehicle vision processors for ADAS (Advanced Driver Assistance Systems), as described in [20]. This processor embeds 128 8-bit processing elements and a 16-bit control processor (CP). Like the previously shown Acadia architecture, is based on a grid of processing elements which are connected toghether with a configurable interconnection switch.

Each processing element (PE) receives column of image or partial data and processes them internally. The processing elements are forming a one dimension connected linear array to execute all vector operations. The CP supervises the whole operation while executing sequential tasks like loop control, global conditions or programming of peripherals.

The processor is connected to two SSRAMs which contain program memory, data memory for the control processor and external memory to save images. As reported by [21], the IMAPCAR is capable to perform 100 GOPS (8bit equivalent) with only two watts of power.

3.1.4 SeePROC

The SeePROC architecture has been proposed by [14] in 2012 and represents one of the most recent reconfigurable solution in the SCN scenario. Actually is not a real processor, but works as a co-processor environment devoted to the image processing tasks.

In Fig. 3.5 the internal of SeePROC architecture is shown. The internal structure is divided in two functional blocks: SeePROC_Decoder and SeePROC_DPR.

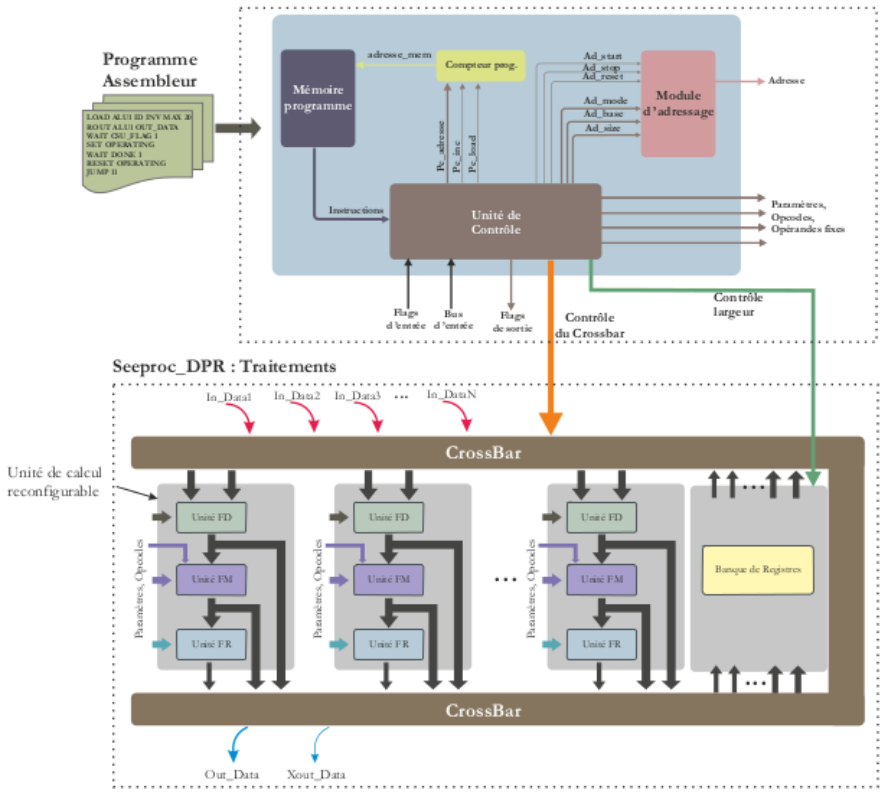


Figure 3.5: SeePROC architecture.

The SeePROC_Decoder is made up by a RISC processor and some custom logic for external interface while SeePROC_DPR is dedicated to stream processing.

The SeePROC_DPR is composed by several elaboration modules and some interconnection switch, which are controlled by the SeePROC_Decoder. The configurable datapath controller, represented in Fig. 3.6, contains a configurable ALU, called ALUM (Matricial ALU). Each ALUM block can handle two data stream and is connected to a interconnection multiplxer, as happens on ConvNet processor.

In other word, SeePROC_DPR acts as a SeePROC_Decoder coprocessor, taking the heavy image processing operations out of the sequential execution. Indeed, given a defined computer vision algorithm, the SeePROC_Decoder decodes the instruction and configures the interconnection. A custom language has been developed to control the machine and process the datastream.

3.2 EFFEX processor

With support from the National Science Foundation and the Gigascale Systems Research Center in [22] a new multicore video processor has been developed in 2011. This processor, called EFFEX, is specifically designed to increase the speed

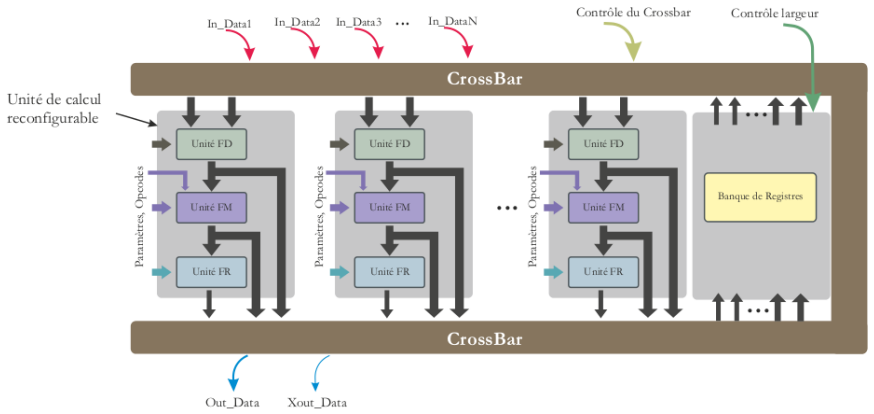


Figure 3.6: SeePROC_DPR internal.

of feature-extraction algoritms.

The EFFEX CPU usesa a programmable architecture able to handle different image processing algorithms, such as Scale Invariant Feature Transform (SIFT), Histogram of Oriented Gradients (HOG) and other convolutional operations.

With respect to the previously shown architecture, EFFEX processor has been thought with specific power supply constraints. The simulation results shows that can outperform GPU and CPU solutions by a factor of two while presents a considerable increasing in performance in terms of fps [22].

3.3 Xetal Micropocessor

Xetal Micropocessor has been developed by NXP (former Philips Semiconductor) since 2001 [23]. The version run at 18MHz with 320 Processing Elements (PEs) and 16 line memories. Since each of the PEs can perform one operation per clock cycle the performance results of 5.7 GOPS. As a result, combined with a CMOS image sensor at QVGA resolution running at 15 frames per second the Xetal 1 could essentially perform 5000 operations per pixel with a low power consumption (about 2 Watt). In more recent years, newer version of Xetal has been presented. In [24] the Xetal-II processor is presented and is compared to the previous version. The new version is capable of 107 GOPS at 84MHz, with 10Mbit of on board memory, 27 billions of MAC units and a power consumption above 600mW. This processor can be used for high speed image processing using a dedicated languages, which is an extended C (XTC). One of the major extensions is the introduction of a vector data type (vint) to represent the 320-element wide memory in the Processing Element array. It is worth to mention the last version of Xetal, called Xetal-Pro [25], which supports ultrawide supply voltage scaling to optimise the

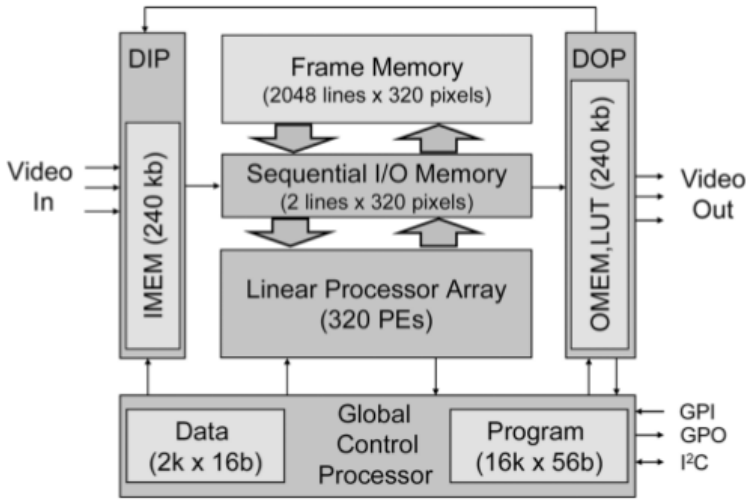


Figure 3.7: Xetal-II architecture.

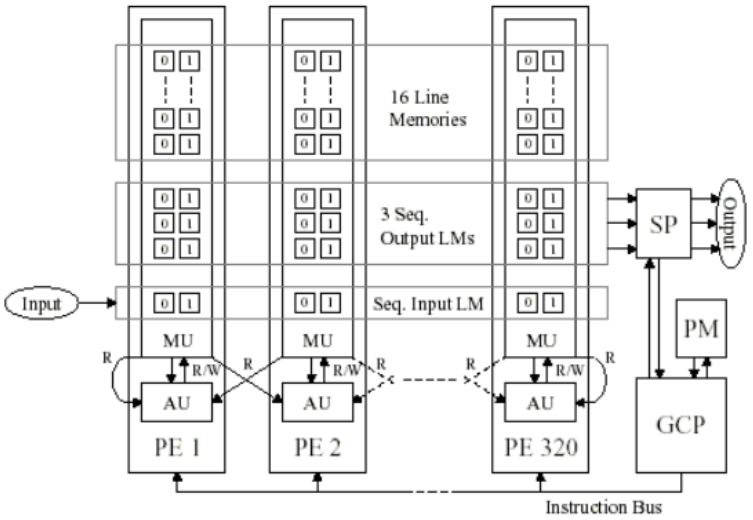


Figure 3.8: Xetal Processing Elements line.

power consumption. Wide voltage variation reflect a decrease in performance due to the limited operative frequency near to subthreshold voltage, thus the authors have specifically designed a massive parallelism to mitigate the voltage limitation drawbacks.

Chapter 4

Our architectural proposal

This Chapter presents the proposed modular reconfigurable architecture, called CameraOneFrame. CameraOneFrame is a data-path based flexible architecture for signal processing. It is composed by a set of hardware modules that process the data and a configurable path controller, which defines the connection through the elaboration blocks.

4.1 Overview

The main idea is based on the model based design assumption, where processing tasks are executed as a sequential flow of operations. If a target application can be divided in a defined number of elaboration steps, the proposed architecture is well-suited to give an optimised solution.

In a SCN context, imagine that a CMOS camera is connected as an input video peripheral. This device usually works at high frame rate, above 25 fps with a defined resolution. The amount of datas to be processed is directly dependent on the video bus throughput, i.e. at 25 fps, 320x240 pixels, YUV422 format brought up to 4 Megabytes per second. Moreover, due to the limited resources available, usually an embedded system can not save a whole frame in the on-board memory.

As a result, embedded video processing tasks (with high speed channel and time constraints) require a higly optimised architecture, capable to process datas at the maximum speed allowable by device technology.

Dedicated hardware solution provides best results compared to the general purpose design, such as commercial microcontroller. Hardware design exploits the the image processing tasks taking advantages of parallel computation and dedicated data structures. Despite a general purpose approach, which is designed

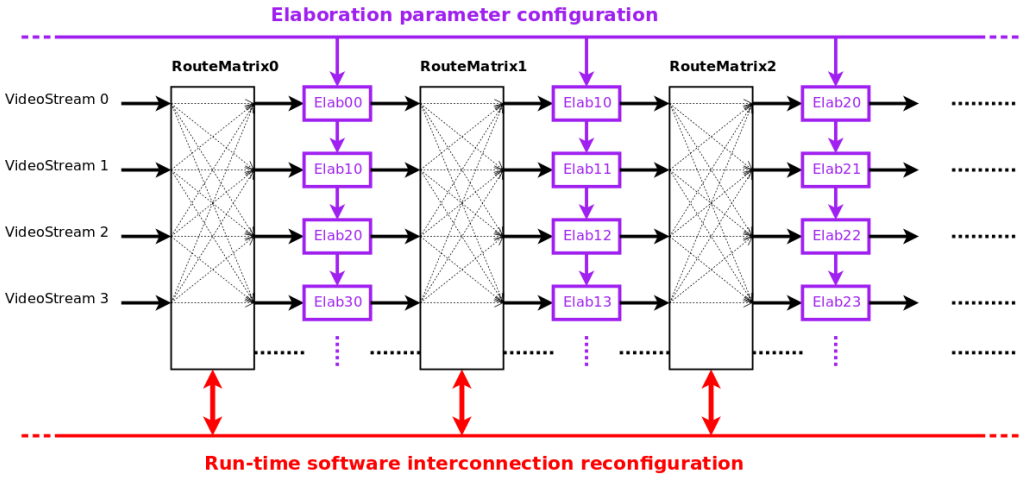


Figure 4.1: Camera_OneFrame architecture.

for multiple applications, a custom hardware design is thought to be optimised and dedicated to a specific operation.

To overcome the above presented drawbacks, CameraOneFrame architecture separates the elaboration chain from the interconnection controller. The former has strict timing constraints and is eligible for a custom hardware design, while the latter keeps a simple structure allowing a software reconfiguration. The paradigm that permits to implement these features is called hardware software codesign, and represents the state-of-the-art of the FPGA programming. This method merges the flexibility of software programming with the parallel computation allowed by hardware modules.

4.2 CameraOneFrame

The proposed FPGA architecture called *Camera_OneFrame*, is shown in Figure 4.1. It permits to create a full reconfigurable pipeline in a SCN node. The whole architecture is designed focusing on interconnection modules, called *RouteMatrix*, and functional blocks, called *Elab*. The former realises the connections between the functional blocks, while the latter implement basic computer vision algorithms that can be part of a specific elaboration pipeline tunable at run-time.

In the left side of the figure four video input ports, labeled as *VideoStream* are shown. This architecture allows to have a multi-camera video inputs that can be parallel processed as a continuous pixel flow. The data flow, composed by one or more video streams, is captured and then processed by successive steps, represented by Elab blocks.

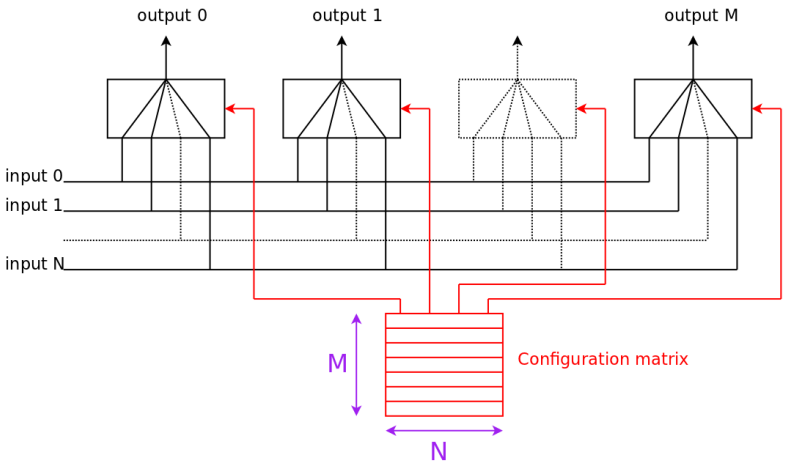


Figure 4.2: RouteMatrix internal architecture.

RouteMatrix module

RouteMatrix is the core of the CameraOneFrame architecture: it is the connection point between hardware configuration and software programming. The logic behind our approach can be seen as a 3D-multiplexer, having a $N \in \mathbb{N}$ inputs and $M \in \mathbb{N}$ outputs (Figure 4.2), and configured by a $N \times M$ matrix through a dedicated bus (the red lines in Figure 4.2).

In this way, the RouteMatrix internal logic guarantees that each output is connected to a selected m-th input vector (with $m \in [1, M]$), to avoid data collision. On the other side every input vector can be connected to several outputs, in order to generate two or more twin sub-pipelines from the same source.
More in details, the RouteMatrix module permits to:

- define the routing path of a data stream (Figure 4.5a);
- handle the execution of parallel pipelines with different data sources (Figure 4.5b);
- split a data stream in two or more pipelines (Figure 4.5c).

Elab module

In this work we define an hardware block as the implementation of a certain computer vision algorithm in any HDL languages (e.g., Verilog, VHDL, CAPH [27] [28]). Every block requires at least one input and one output with the related data-valid signals, notificationing valid output data towards following blocks.

The data-valid signals are necessary for enabling the streaming paradigm: in this direction the proposed architecture does not require to specify any data latency

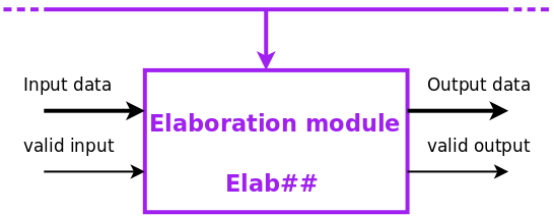


Figure 4.3: Elab block external architecture.

and processing time.

Therefore, the overall latency can be computed as a cumulative sum of the delay cycles inserted by the activated elaboration blocks. By controlling the path at runtime, we can predict the data latency of a selected application, taking into account the overhead introduced by the RouteMatrix modules.

Despite a software oriented solution, which does not allow a precise delay evaluation, due to the cache miss parameter, memory accesses and interrupts that may stop the exectution, this solution provide a deterministic delay latency extimation.

In Chaper 5 a set of elaboration modules are shown as a part of a Hardware Library tool.

4.2.1 Streaming processing

As pointed out by the previous section, in embedded devices we have to cope with limited hardware resources, such as RAM and processing capabilities and low clock speed. The problem arises when a high speed dataflow has to be processed with real-time constraints, i.e. object recognition and tracking. Embedded devices usually lack of enough memory to store a single frame, thus rapid processing tasks are needed.

In this context, a hardware architecture can provide an optimised solution and process the datas directly as they appear to the input. This is the concept above the so-called *Streaming processing*, which is typically deployed into FPGA modules [29]. CameraOneFrame implements an extreme streaming approach, where every stages and every hardware modules are built to process the informations as a continuous flow.

In order to explain the internal behaviour, in Fig. 4.4 an example is shown. Every elaboration block represents a separated sequential logic clock domain and behave as a registered pipeline stage. Thus the architecture provides the maximum allowable clock speed regarding to the selected silicon technology, reducing as

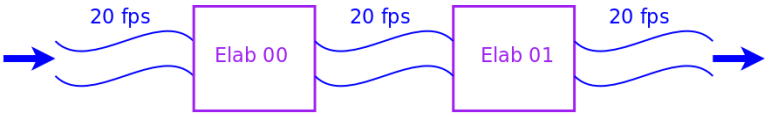
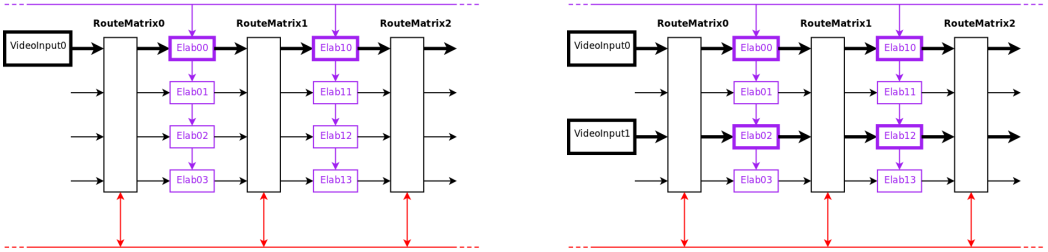
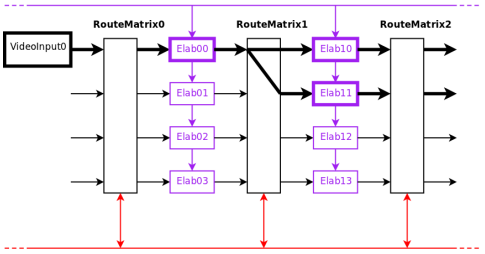


Figure 4.4: Streaming elaboration example.



(a) Single pipeline.

(b) Parallel pipelines.



(c) Pipeline splitting.

Figure 4.5: Data path reconfigurations.

possible the critical path between close sequential logic module.

4.2.2 Reconfigurable architecture

Reconfigurable solutions are usually addressed by software oriented approaches. In such a vision it is straightforward to modify both configuration parameters and applications at run-time, at the cost of avoiding possible low-level optimizations.

Instead, the use of a pure based hardware approach results in the realization of static and monolithic hardware pipelines optimized only for a single application. To overcome the above depicted limitations, while keeping the possibility of dynamic configuration, in this work we present a mixed solution, which takes advantages from hardware optimisation and still considers a software-based configuration. Through RouteMatrix configuration the data path can be created at run-time. As run-time we mean during the operating phase of the system, i.e. after an event or during a communication from another node.

immagine da slide prima presentazione (o da articolo icdsc) esempio di riconfigurazione, considerazioni su redirezione dei dati, funzionamento del data valid e

prospettive di inserimento dei segnali di end_frame, start_frame

4.2.3 Modular architecture

Having a modular architecture allows design engineers to reuse previously developed and tested IP blocks, without any further modification. Thus, an optimised hardware module can be used on a wide range of applications and could be easily shared for community development. Moreover, the hardware abstraction performed by the architecture allows the instantiation of some Library modules without any electronic and design skills. This is important to define a general architecture, to be used as a pervasive, where also a not expert personnel have access to the system design.

Despite the fact that lower knowledge are requested, an optimised solution is guaranteed by the accurate design of the hardware IP blocks. In other words, how is a module instantiated does not interfere with the optimisation grade of the overall elaboration pipeline but depends only on the involved HDL design.

The described internal FPGA structure can be expanded as a function of a set of possible applications. Indeed, the number of input and output ports, even the number of elaboration steps can be abstracted as parameters and then configured during the hardware compilation. This allows to modify the number of the parallel data flow, the amount of pipeline steps, or the elaboration blocks, without modifying the HDL instances, but easily inserting them as a new Elab instance using for example a graphic interface tool.

Each functional block performs an associated algorithm, implemented by a hardware module available into a Hardware Library tool (see Chapter 5). This library contains a certain amount of functional blocks, defined using HDL, and then collected into a package made available as high level resource. After the instantiation phase, the system is ready to be compiled and programmed in the FPGA as a bitstream.

Afterwards, though the FPGA bitstream is statically programmed, the system still keeps the flexibility through the software configuration of blocks and connections. This aspect assures and guarantees a dynamic and adaptive system while realising an optimised solution eventually after software re-configuration.

The proposed internal FPGA architecture introduces two degrees of freedom: (i) it is possible to grab the requested functional operation from a library, without any knowledge of HDL languages as it was in a model-based unit, and (ii) it is possible to configure the system at run-time to perform new pipelines with the

already installed hardware modules configured and connected through the software interface.

4.2.4 Configuration semantic

The captured flow has not an associated semantic, so that an user interacting with a configuration manager can select and compose the appropriate modules suited to the desired application.

The software controls the datastream redirection and the parameter update without any specific instruction. As a consequence, every RouteMatrix instances can be addressed using the build-in instruction (load and store assempbly instruction and the corresponding C wrapper).

Thus, given a generic architecture, composed by a various number of elaboration blocks and RouteMatrix modules, we will be able to develop a hybrid system, able to configure itself during start-up phase.

4.3 Architecture Implementation

All the developed hardware library functions are compliant with the Qsys tool provided by Altera as a part of the Quartus II software edition [30]. Qsys abstracts every HDL module as a functional entity into a graphical interface, thus helping in instantiating the blocks inside the proposed architecture. In Figure 4.6 the design flow for a generic computer vision application is shown using the Qsys tool: (i) the application is chosen (ii) and divided into functional elements; then (iii) the concrete HDL blocks are instantiated using the above mentioned tool, and finally (iv) the code is compiled into a bitstream.

To complain with the above mentioned behaviour, a Softcore CPU has been programmed in the FPGA. The CPU has the role of datapath configurator and manages the communication to the network layer. Adopting a softcore CPU brings to us some advantages rather than deploying an external microcontroller, for intance:

Single chip solution A Softcore CPU is well suited for cost effective, integrated embedded system because it reaches the highest integration possible. Even though Softcores are not optimised for processing as ASIC solutions, here we use the CPU only for controlling purposes, with limited requirements on

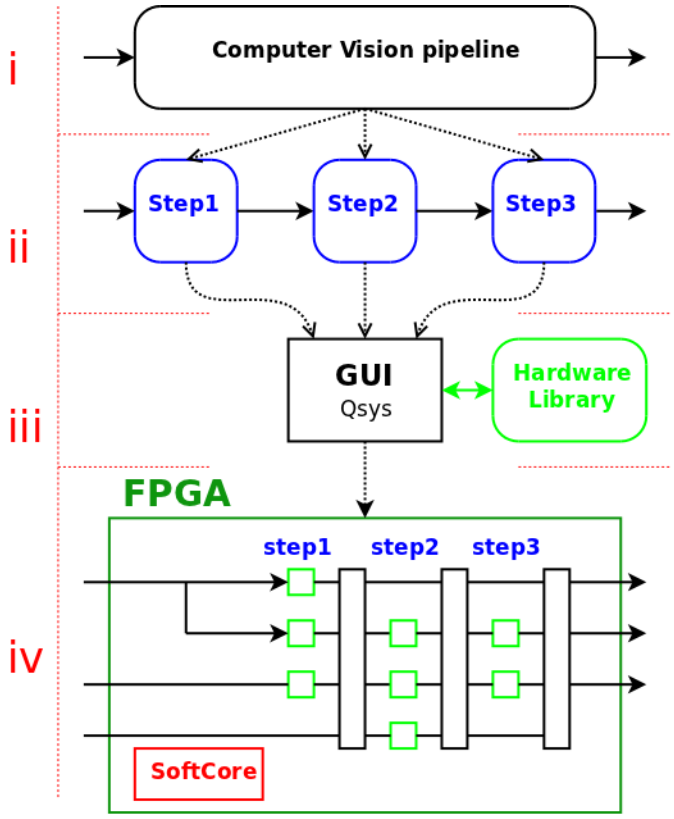


Figure 4.6: CameraOneFrame design flow.

the computational resources. Indeed, we can deploy even a small CPU, able to perform simple operation and reduce the area as possible.

Direct addressing As pointed out previously, custom hardware peripherals can be addressed by the CPU data bus, assuring a complete control of the system. Importance has to be paid to the hardware design, in order to exploit the requested functionality with a focus on "what can be configured at runtime?", " how can i design a hardware IP in a more general way?".

Reduced latency time The internal bus works on the maximum clock speed allowable by the FPGA technology, without delays due to the I/O blocks. As a result, most of the communication are executed on a one-cycle clock period.

Extreme system integration The complete system, composed by the CameraOneFrame architecture and the CPU, represents a System on a Programmable Chip (SoPC) which does not require any other external components, at least for a minimal working solution.

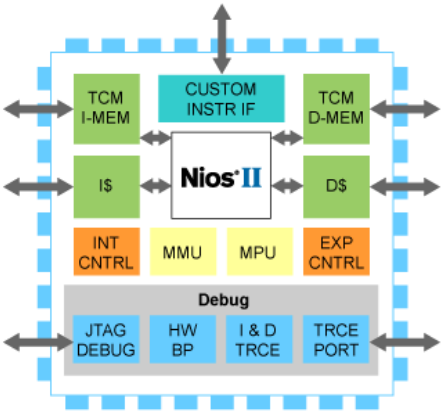


Figure 4.7: NiosII RISC CPU

4.3.1 NiosII RISC CPU

With this respect, in the development phase it has been deployed the Altera Nios II Softcore CPU, which is a part of the Altera Embedded Design Suite (EDS).

Nios II is a 32-bit RISC embedded-processor architecture designed specifically for the Altera family of FPGAs. Nios II is suitable for a wider range of embedded computing applications, from DSP to system-control. Nios II is implemented entirely in the programmable logic and memory blocks of Altera FPGAs. The soft-core nature of the Nios II processor lets the system designer specify and generate a custom Nios II core, tailored for his or her specific application requirements. System designers can extend the Nios II's basic functionality by adding a predefined memory management unit, or defining custom instructions and custom peripherals.

For performance-critical systems that spend most CPU cycles executing a specific section of code, a user-defined peripheral can potentially offload part or all of the execution of a software-algorithm to user-defined hardware logic, improving power-efficiency or application throughput.

Nios II is offered in 3 different configurations: Nios II/f (fast), Nios II/s (standard), and Nios II/e (economy).

economy **600 LE** (3%), 0,15 DMIPS/MHz, **royalty-free**

standard 1300 LE, 5-stage pipeline, 0,74 DMIPS/MHz

fast 1800 LE, 6-stage pipeline, 1,16 DMIPS/MHz, MMU

The envisioned softcore application does not require high computational power, whereas a limited resource occupation is well-regarded. Indeed, as CameraOneFrame controller we deploy the smaller version, the NiosII/economy. The Nios

II/e core is designed for smallest possible logic utilization of FPGAs. It is capable of 0,15 DMIPS/MHz and does not require any royalties for a commercial version (it also be incorporated in the flash bootloader which is launched at power up).

4.3.2 Avalon Memory Mapped bus

The NIOS II Softcore implement an Altera royalty-free data bus, called Avalon Memory Mapped [31] (or Avalon-MM) which is capable to address a custom FPGA hardware IP as a memory location. This abstraction is usually deployed in the microcontroller system, where the external interfaces (such as UART, SPI, I2C, etc.) are addressable through the standard load *ldw* and store *stw* assempler instructions. In the programmable logic, this concept extends the hardware blocks integration and provides a reliable and flexible way to a parametric configuration. Indeed, every elaboration block can be mapped on the Avalon-MM bus to be addressed from the NIOSII as a standard memory location.

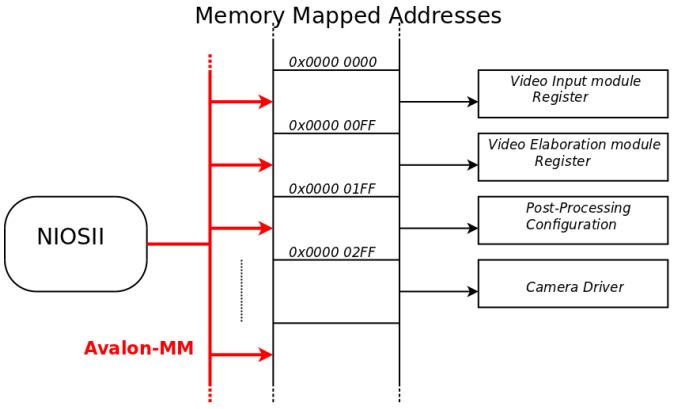


Figure 4.8: Altera Avalon Memory Mapped bus.

All in all, this main feature allow us to: (i) set up the block interconnection at run-time and (ii) control the elab- oration parameters for every block inserted into the elaboration pipeline.

In Fig. 4.8 a schematic view of a possible addressing schema is shown, along with some custom peripheral. The addressing index has been fixed during the instantiation phase and then fired into the FPGA bitstream during programming.

Every hardware module which has an Avalon-MM port could be configured at runtime as requested by the application. Indeed, during the operating time of the smart camera, could be necessary a pipeline change. With CameraOneFrame we can figure different updating methods:

Authonomous setup This is the first scenario, when the smart camera decides

alone to request a hardware reconfiguration to adapt itself to a enviromental change or after a certain event (i.e. wheather and lights conditions or a detected event). The software application defines which are the thresholds or the behaviour behind this kind of setup.

Coordinator request In this situation, the device is connected to a star-topology SCN, where a central node controls the overall configuration. Through a defined communication protocol (an exanple is shown in Sec. 4.2.4), it would be possible to deploy a centralised SCN. Moreover a reconfiguration request could be requested by the user, for example as video surveillance system.

Distributed network request Distributed processing needs also a distributed configuration. In this case, the recognition of certain features could trigger some involved nodes to request a recalibration through the network. In this case we do not have a central node but a distributed decision layer, made up by a set of cameras.

Chapter 5

Hardware Library

The CameraOneFrame architecture has been designed to extend the flexibility of an optimised hardware design. Given a set of image processing hardware blocks, namely hardware IPs, CameraOneFrame creates a functional computer vision pipeline by configurable interconnections to fit the target application.

As introduced in Chapter 4, CameraOneFrame architecture offers interconnection and configuration facilities to the processing IPs, but does not provides them. Indeed, the available hardware IPs are provided in a Hardware Library tool released with CameraOneFrame architecture.

The aim of the Hardware Library is to collect every hardware modules designed for CameraOneFrame and make them available for future reuse. Hardware IPs reuse is a key factor of CameraOneFrame, because:

Reduced development time deploying already tested hardware blocks drastically reduce the development time.

Easy to use Even a complex hardware IP can be instantiated easily in the pipeline. Digital and hardware design skills are requested during the circuit design only.

Simple interface the designer should follow the few design guidelines shown in Sec. 4.2.

Bug-free IP The more will be the users, the sooner bugs will be discovered and corrected.

These Hardware Library intances can be developed with any available Hardware Description Languages (HDLs), e.i. Verilog or VHDL for low level design or other model-based languages such as CAPH or CAL. High level approaches are

still supported, e.g. Matlab and Simulink, but they are disincouraged due to the reduced optimisation degrees that an automated HDL generation provides. Once the module interfaces is fully defined, the internal structure is hidden to the architecture level. As a consequence, even different programming languages can be deployed in the same application, ensuring the maximum flexibility.

In the rest of this Chapter, some developped hardware modules are shown. They are specifically designed to reach great time perfromance compared to software solutions. Each IP block will be presented as a result of a optimisation work for a specific task and then evaluated towards a software reference.

Figure 5.1: Hardware Library tool.

5.1 VideoSampler

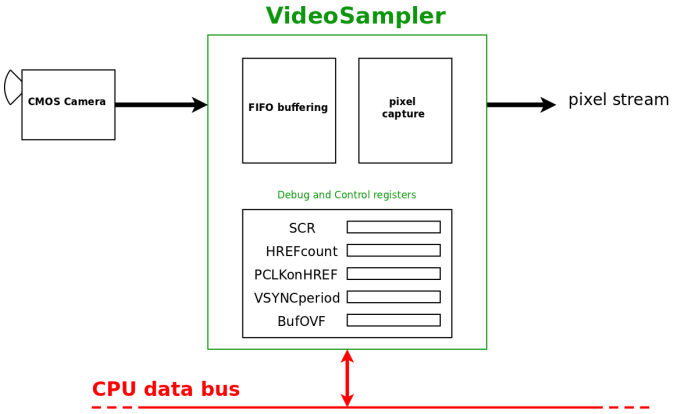


Figure 5.2: VideoSampler module.

The VideoSampler module is an interface towards an external CMOS camera and manages the video stream synchronization. VideoSampler takes as input the synchronization signals, namely HREF (Horizontal Reference), VSYNC (frame synchronization), PCLK (Pixel CloCK) and generates one or more stream data flow for the CameraOneFrame architecture.

In Fig.5.3 the internal structure is shown. In the upper side the sampling module performs the clock domain conversion between PCLK and system clock. This operation is performed by deploying a dual clock FIFO (made by 512 locations) which acts as a buffer between different data speed. Then in the right three signal are connected to output: *pixel_data*, *data_valid_intensity* and *data_valid_intensity*. Pixel_data keeps the actual value of the pixel, referred to the system clock, while the data intensity signals are meant to separate color and intensity values if a YUV-YCbCr format is used. For instance, a interlaced YUV422 format can be trated as a double data stream: one contains the color informations while the other luminance levels. This de-interlacing behaviour can be activated or configured by software application. Moreover VideoSampler provides hardware RGB to YUV conversion, for both RGB444 and RGB565 format.

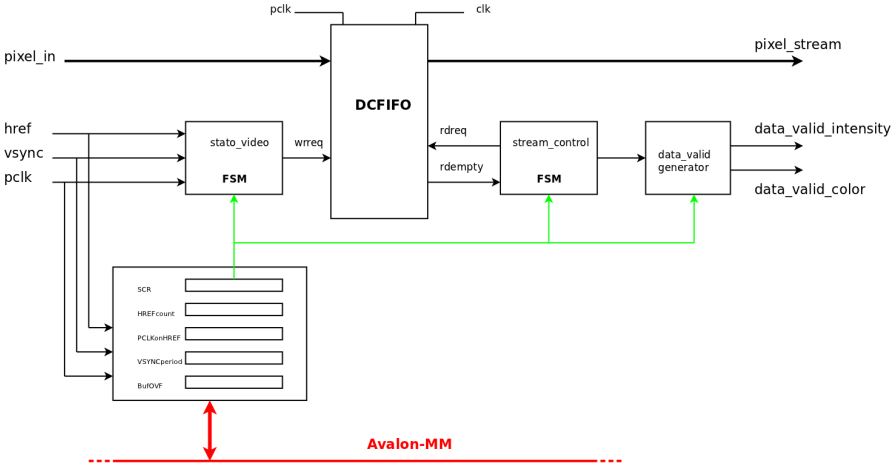


Figure 5.3: VideoSampler module.

The lower side of Fig.5.3 represents the Avalon Memory Mapped interface. By accessing the configuration register with a SoftCore CPU is possible to:

- Control the YUV-RGB conversion.
- Handle the deinterlacing machine.
- Trigger the acquisition to the next VSYNC pulse.
- Detect which is the captured frame resolution and perform an autoconfiguration.
- Control the strean flow by checking the number of captured pixels, the amount of line in a frame and the input frame rate.

The maximum pixel clock speed (PCLK) allowable depends on the FPGA technology deployed. Using an Altera Cyclone IV EP4CE22 FPGA the maximum pixel clock frequency is up to 100MHz. Hence about 50fps of a VGA frame (640x480 pixels) in the YUV422 format can be captured by VideoSampler without any external buffering system.

5.1.1 Data Register Settings

Tab. 5.1 shows the register map for the VideoSampler core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 5.1: VideoSampler Register map.

Offset	Name	R/W	Descriptions of Bits
0x00	Status Control	RW	RGBYUV_EN[3] COUT[2], YOUT[1] , ON[0]
0x01	HREF count	RW	HREF pulse counter [31:0]
0x02	PCLKonHREF	RW	Pixels in a line [31:0]
0x03	VSYNC period	RW	Frame rate [31:0]
0x04	BufOvf	RW	Buffer overflow [31:0]

By controlling the BufOvf bit the system can assure a correct data acquisition, excluding overwriting problem inside the above presented dual clock FIFO. The other registers may be used to detect the image resolution automatically after a single test frame has been captured.

5.2 RemoteImg

The RemoteImg module performs the image data acquisition through a standard UART link. It can be used as a video stream source in parallel with VideoSampler. Therefore it is very useful during the development and during dataset training, where the image datas are sent by the user. The host system can be a PC, which runs a simple sender application or another device (also another smart camera). In this latest case, another external peripheral acts as a data source, sending raw image data or even precomputed complex features enabling a data fusion through different smart camera instance.

An internal view of the RemoteImg module is shown in Fig. 5.4.

Internally this module performs the host clock sincronization and samples the input data according to the selected baudrate, which is configurable through a dedicated register.

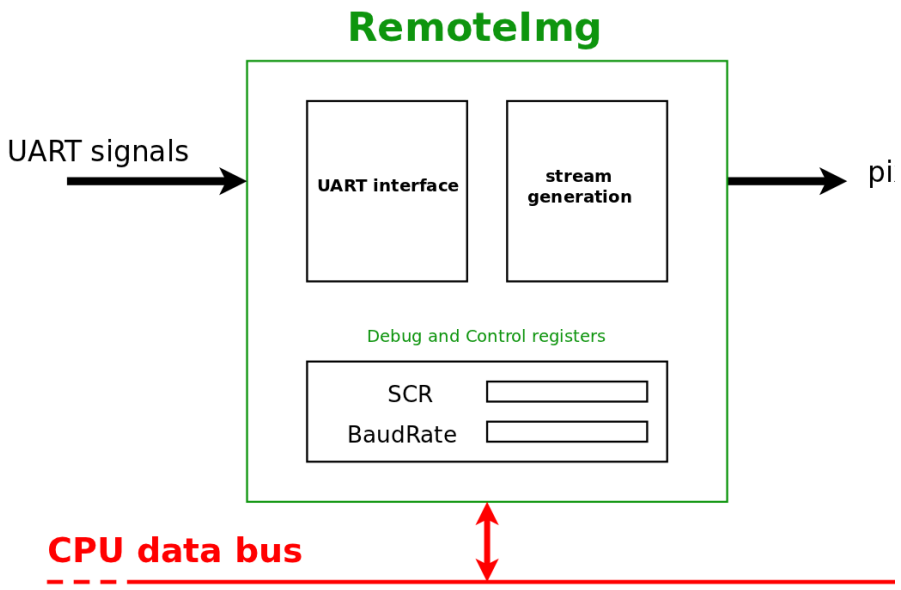


Figure 5.4: RemoteImg module.

5.2.1 Data Register Settings

Tab. 5.2 shows the register map for the RemoteImg core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 5.2: RemoteImg Register map.

Offset	Name	R/W	Descriptions of Bits
0x00	Status Control	RW	RX_ERR[1] , ON[0]
0x01	BaudRate	RW	Baudrate configuration [31:0]

5.3 StreamStore

The StreamStore module is the stream sink after the capturing and elaboration stages. Here the datastream (that, we would remind, could be also a generic data input, such as audio or generic data) is converted and then transferred to a memory addressable locations. Once transferred into memory locations, the data can be accessed by the SoftCore CPU and used for other computer vision application and high level data aggregation. The sink operation uses a FIFO buffer to handle the different memory timing access and burst mode configuration.

The StreamStore module works like a Direct Memory Access (DMA) controller. By giving the initial storage address and the number of the location that will be used, the StreamStore capture the data stream and convoys it to the memory.

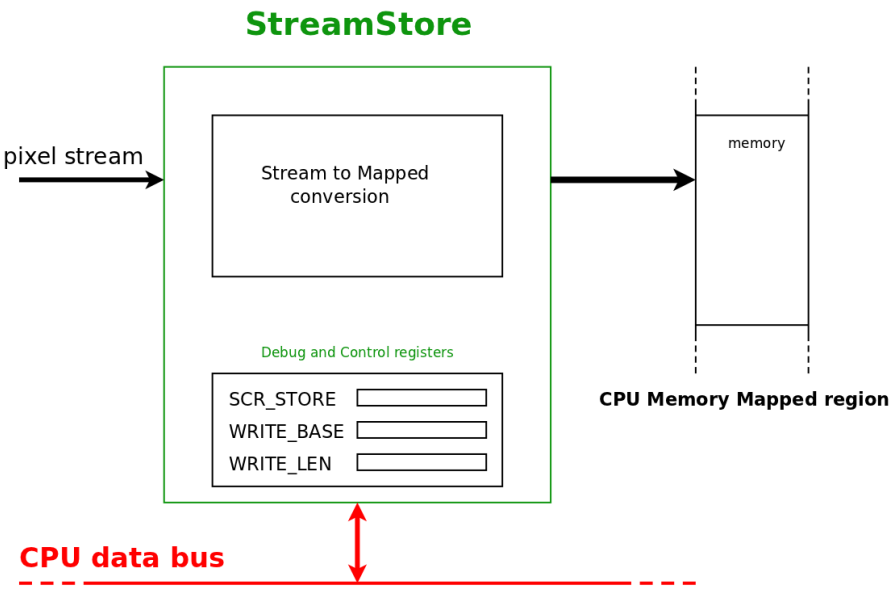


Figure 5.5: StreamStore module.

The destination locations will be then overwritten and at the end, the core assign an "operation completed" flag. In future releases this bit will be mapped as interrupt trigger for a dedicated Interrupt Service Routine (ISR). By controlling the *FIXLOCATION* bit, the user can interface the StreamStore directly to another Avalon Memory Mapped peripheral, for instance a LCD controller or a data link cable, which are able to handle the data flow directly on a single addess.

It is important to underline that a generic elaboration pipeline usually exploit image processing operation by reducing the amount of data from the source to the sink. This is due to the context-based informations that are extracted each stage and provided as input to the following blocks. By image understanding we can reduce the amount of data from the high bandwidth requested at input to the memory interface at the output. As a result, even a multicycles Synchronous Dynamic RAM (SDRAM) can be deployed as a storage sink, with better performance on intergration and cost rather than a complete static memory.

5.3.1 Data Register Settings

Tab. 5.3 shows the register map for the StreamStore core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 5.3: StreamStore Register map.

Offset	Name	R/W	Descriptions of Bits
0x00	Status Control	RW	FIXLOCATION[1] , ON[0]
0x01	WRITEBASE	RW	Initial base address [31:0]
0x02	WRITELENGHT	RW	Number of locations [31:0]

5.4 GradientHW

Edge extraction is a typical task performed in hardware due to the simple operation repeated continuously every pixel. In the spatial domain, an edge become a high frequency components inside a defined pixel area. Thus an edge can be revealed with an thresholding of the local spatial gradient magnitude, taking into account the pixel luminance values. This is a very commmon operation in image processing because represents the first elaboration step of a complex computer vision algorithm. Therefore is important to realise such low-level operation as an efficient solution, in order to speed up the entire pipeline.

As a mathematical point of view, the gradient extraction can be explained considering the luminance spatial variation in over a crown of close pixels. The gradient is a vector defined by:

$$\nabla I = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right] \tag{5.1}$$

where I represents the luminance computed as a derivative over $x - axis$ and $y - axis$.

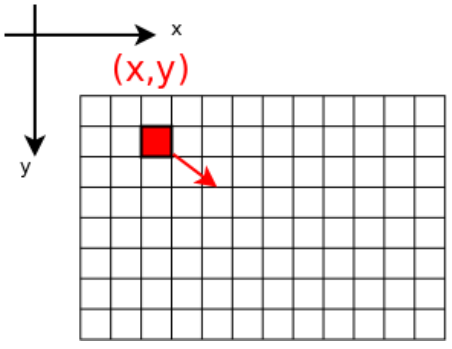


Figure 5.6: Pixel space.

In the pixel space (see Fig. 5.6), each derivative is computed as a Newton's difference quotient. Every pixel has an associated gradient vector, with rectangular component I_x and I_y, each one calculated with a convolutional mask over three

oriented pixel.

$$\frac{\partial I}{\partial x} \equiv I\big(x+1, y\big) - I\big(x-1, y\big) \tag{5.2}$$

$$\frac{\partial I}{\partial y} \equiv I\big(x, y+1\big) - I\big(x, y-1\big) \tag{5.3}$$

where I means luminance at (x, y). Then magnitude m and direction Θ of the computed gradients are computed by

$$m = \sqrt{I_x^2 + I_y^2} \tag{5.4}$$

and

$$\Theta = arctan\Big(\frac{I_y}{I_x}\Big) \tag{5.5}$$

respectively.

In Fig. 5.7 the gradient computation dataflow is shown. Magnitude and angle are computed by a *square root* and *arctangent* operation which are computed with iterative methods and floating point arithmetic.

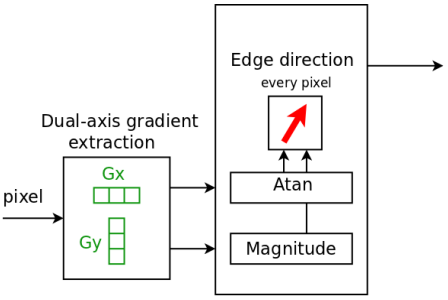


Figure 5.7: Polar space transformation.

5.4.1 Are Floating Point operations suitable for FPGAs?

These operations are not an issue on modern CPUs, which more often have a floating point coprocessor, but still represent a critical problem in embedded devices. As things stand today, floating point arithmetic is avoided in microcontroller because fixed point arithmetic provides better performances, although a lower accuracy.

Even though a floating point arithmetic can be deployed (with a significative decrease in time performances) in microcontroller, hardware design does not

consider at all this possibility. Indeed, in hardware we have access directly to the natural numbers with two's complement maths and adding some tricks to the fractional representation while the IEEE754 arithmetic needs a dedicated hardware IP and specific FSM to manage the exception routines.

As reported by Altera in [32], a single Floating Point (FP) core in an FPGA can achieve great performance even in a rather small FPGA. Of course, this is not representative of any real-world application, where many such cores are used in parallel. Routing multiple 64-bit data paths while completely filling an FPGA with FP cores is a challenge, usually resulting in unused logic in the device and a decrease in clock speed for some of the FP functions.

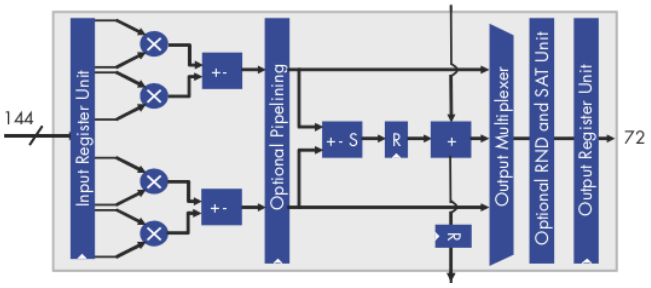


Figure 5.8: Altera Stratix II DSP block.

Since typical math operations are composed by several concurrent tasks, all operands are require to complete at the same time, these parallelized operation will occour at the slowest clock speed of all the FP functions on the FPGA.

The following consideration are usually accepted as typical "constrained performance" prediction in a real-case FP core usage:

- Estimated 15 percent logic unusable (due to datapath routing, routing constraints, etc.)
- Estimated 33 percent decrease in FP function clock speed
- Extra 24,000 ALUTs for local SRAM memory controller and processor interface

In the following Tab. 5.4 the FPGA resource occupancy are shown. While FP multiplication uses rather small resources in terms of ALUT, the adder drawn about 1700 ALUTs, which are about 10 % of a small FPGA. Those results considers only the simpliest FP operation, while division and square root are not considered.

With those assumptions, only one or two FP core will fit in the considered low-cost platform and the clock speed will drop from 100MHz to a 47MHz.

a	Synthesis	Multipliers	ALUTs	Latency
Multiplier	Logic + Multiplier	9	900	13
Adder	Logic	0	1721	17

Table 5.4: FP Core Resource Utilization.

Another challenge in using all the FP cores that fit in the FPGA is feeding them all with data on every clock cycle. When dealing with double-precision 64-bit data, and parallelizing many FP arithmetic cores, wide internal memory interfaces are needed. Again we need an internal machine that controls the data stream inside the FPGA [33].

For example, [34] presents an implementation of a IEEE754-compliant FP co-processor using a high-end fpga chip, the Virtex II distributed by Xilinx. The module provides a data valid every clock cycle, with a latency of 4 cycle and implements most of the operations required by the floating point maths (add, sub, mul, compare, divide, round) but is not suitable for a streaming processing application due to the FSM that controls the core behaviour.

5.4.2 Integer arithmetics

The previous section shows that a FP core is not suitable for the parallelization requested from the gradient extraction purpose. Moreover, the above outlined method of angle calculation is only suitable for processor implementation, but not for FPGA implementation because calculating the tan^{-1} function and division (in 5.5) are very expensive to implement on FPGA.

Even though there are many hardware friendly approximation algorithms available [35], they are generally iterative algorithms, which slows down the overall systems speed.

For these reasons, gradient computation is usually implemented using fixed point arithmetic on FPGA. Integer hardware gradient extraction is well-known in literature, as described in [35], [36], [37], [38], [39] and [40]. Those solutions present a fully custom and dedicated hardware for spatial gradient extraction.

To the best of our knowledge three methods are avaiable for image gradient computation with integer arithmetic: (i) memory-based, (ii) pre-calculated table and (iii) custom logic design.

A common approach that has been employed is using look up tables (LUTs). This methodology uses directly a pre-loaded LUT which contains all the possible

values, taking into account the input data width. However, using LUTs will require large amount of memory space, which eventually will increase systems cost and will introduce several data latency cycles.

As reported in [35], [36] and [38], a pre-calculated table provides a cheaper (in terms of hardware usage level) alternative. They directly quantize the pixels angular value from its corresponding gradients.

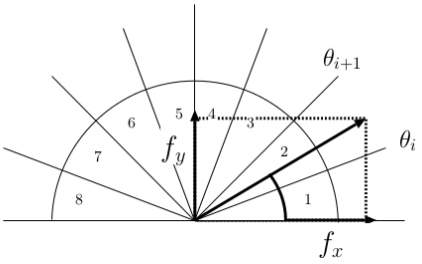


Figure 5.9: Gradient approximation [36].

For example, a pixel at location $P(x, y)$ belongs to a quantized direction (see Fig. 5.9) when its angle is in the range (from Eq. 5.5):

$$tan\Theta_i < \frac{I_y}{I_x} < tan\Theta_{i+1} \tag{5.6}$$

Division operation is prohibitive in term of hardware cost, it can be eliminated by multiplying both side to I_x:

$$I_x \cdot tan\Theta_i < I_y < I_x \cdot tan\Theta_{i+1} \tag{5.7}$$

By this equation, Θ_i and Θ_{i+1} corresponding to two gradient direction shown in Fig. 5.9.

For example, a pixel at location $P(x, y)$ belongs to direction 2 (see Fig. 5.9 when its angle is in the range: $22.5° < \Theta < 67.5°$ or $0.4142 < tan(\Theta) < 2.4142$. This is equivalent to the condition $0.4142 < \frac{I_y}{I_x} < 2.412$.

Finally, the square root operation is implemented by using a look-up-table. This kind of simplification is often adopted for hardware design of embedded systems. To avoid fractional computation, Eq. 5.7 can be approximated by multiplying both side by a scaling factor, arbitrarily chosen to be 1024 in [36].

This methods provides a reliable solution to the gradient extraction routine but in our opinion, lacks of flexibility and requires a high memory footprint to store the square root values, expecially if high accuracy has been demanded.

In the next Section, will be exposed our algorithm proposal to compute the image gradient as well as a comparison versus the floating point implementation.

5.4.3 Our proposal

We propose a different gradient extraction technique, which provides a flexible solution with respect to the already existent works. Rather than the previously mentioned solutions, GradientHW can:

- Implement a general image preprocessing step, which can be configured to be a gradient extraction or a filtering operation or both toghether.
- Provide a configurable accuracy.
- Work at the FPGA operating frequency due to a fully pipelined architecture.

As proposed by [38], the corner domain is sampled in N bins, called Θ_k. The angle resolution is then $\frac{2\pi}{N}$. Each slice identify a versor, called $\hat{i_k}$ (shown in Fig. 5.10, in red).

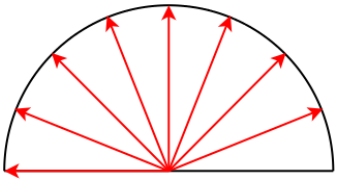


Figure 5.10: Corner domain sampling ($N = 8$).

Each versor $\hat{i_k}$ can be defined as:

$$\hat{i_k} = \hat{x}cos\Theta_k + \hat{y}sin\Theta_k \tag{5.8}$$

The next step considers the Eq. 5.1, where we can imagine the ∇I operator decomposed by its components over the vector $\hat{i_k}$.

$$\frac{\partial I}{\partial \hat{i_k}} = \nabla I \cdot \hat{i_k} \tag{5.9}$$

Eq. 5.9 reveals the gradient component extraction by using a *dot product* repeated N times.

Then substituting at Eq. 5.9 the previously shown Eq. 5.1, 5.8 we obtain:

$$\frac{\partial I}{\partial \hat{i_k}} = \begin{bmatrix} \frac{\partial I}{\partial x} & \frac{\partial I}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} cos\Theta_k \\ sin\Theta_k \end{bmatrix} \tag{5.10}$$

And then

$$\frac{\partial I}{\partial \hat{i_k}} = \frac{\partial I}{\partial x} cos\Theta_k + \frac{\partial I}{\partial y} sin\Theta_k \tag{5.11}$$

Finally, using the Eq. 5.2 and Eq. 5.3, we obtain

$$\frac{\partial I}{\partial \hat{i_k}} = cos\theta\Big[I(x+1,y) - I(x-1,y)\Big] + sin\theta\Big[I(x,y+1) - I(x,y-1)\Big] \tag{5.12}$$

In Eq. 5.12 we ended up with an algebric expression which represents the gradient component over the $\hat{i_k}$ vector. This step has to be repeated for each one of the N versors which divided the angle space.

In Fig. 5.11 the magnitude comparation over the set of versor is shown. The comparation is based on a pipeline bisection procedure and then the result is given, with magnitude and prevalent angle direction.

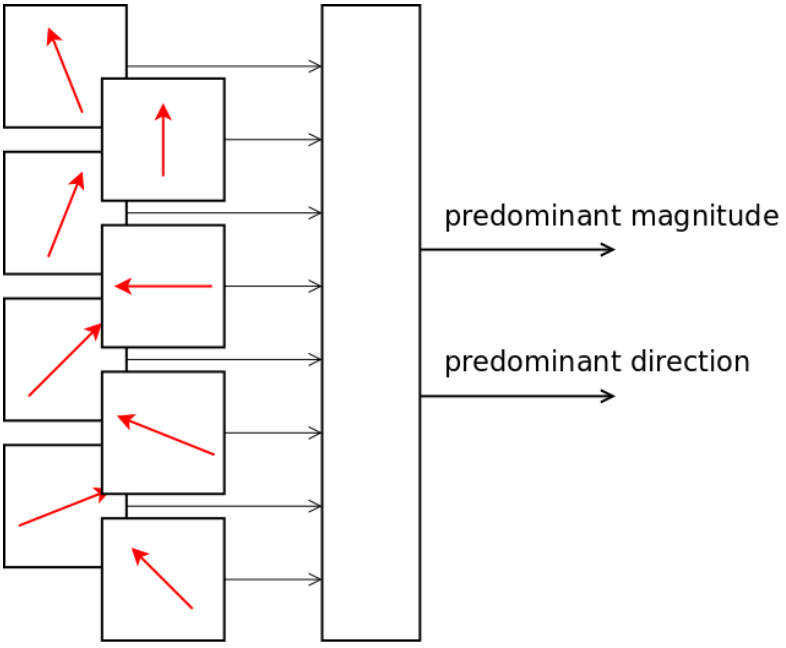


Figure 5.11: Prevalent component searching ($N = 8$).

The outputs contain the magnitude and angle of the pixel spatial gradient, with a resolution of $\frac{2\pi}{N}$.

The above mentioned methodology exploit a flexible and configurable solution to the image gradient extraction and more in general to the image pre-processing. Indeed the number of versors N and the number of comparation stages are defined before syntesis, in order to satisfy accuracy and latency specifications.

In the next section the hardware implementation will be exposed, focusing to the HDL porting of the Eq. 5.12 presened before.

0	$sin\Theta_k$	0
$-cos\Theta_k$	0	$cos\Theta_k$
0	$-sin\Theta_k$	0

Table 5.5: Gradient convolutional kernel.

5.4.4 GradientHW module

As a part of the Hardware Library, **GradientHW** is fully CameraOneFrame architecture compliant.

In Fig. 5.12 the external features of the **GradientHW** module are shown. The module has an input port for video stream and two output ports, for prevalent magnitude and angle respectively. It is made up by two different section: a processing area and a control machine. The former processes the data stream while the latter takes charge of controlling the pipeline, the parameters and the interface to the Avalon Memory Mapped bus.

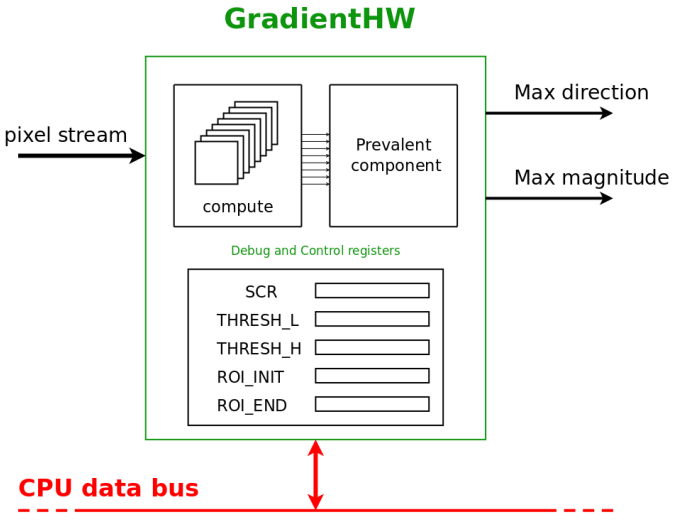


Figure 5.12: GradientHW module.

GradientHW implements the above presented algorithm and is designed to provide output data valid every clock cycle.

It is important to underline that this algorithm is suitable only for hardware implementation, because of the massive parallelism usage. Even though it can be simulated in software, that solution has not any sense in a real-application.

Indeed the module is made up by several hardware blocks which implement the Eq. 5.12 as a parallel operation. Thus creating as many blocks as N value, the components extraction can be completed in one clock period.

The operation is performed by using 4-DSP 9x9bits modules each versor. These

DSP blocks are available in the Altera Cyclone/Arria/Stratix series FPGA and they are composed by hardware multipliers which provide a one cycle latency to output. Nowdays tens of DSP modules are embedded into the FPGA silicon thus a resolution above $N = 8$ is allowable even in a rather small FPGA.

In the Tab. 5.5 the convolutional matrix coefficients are shown. To avoid fractional arithmetic, each cosine function has been pre-multiplied by an arbitrary factor equal to 256. The results are then represented by two's complement over 9 bits. These coefficients will be configurable through dedicated registers, in order to extend the module functionality.

In Fig. 5.13 the parallel matrix computation is shown.

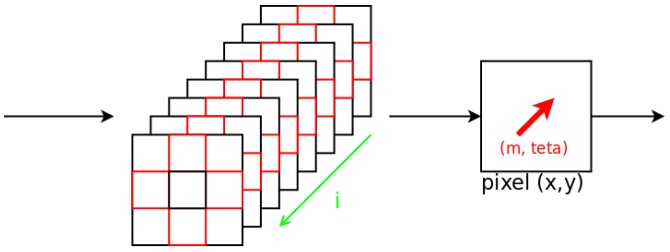


Figure 5.13: Compoment extraction.

Once the gradient components are computed, the next step is the comparation between them. An inner view of the comparation module is show in Fig. 5.14.

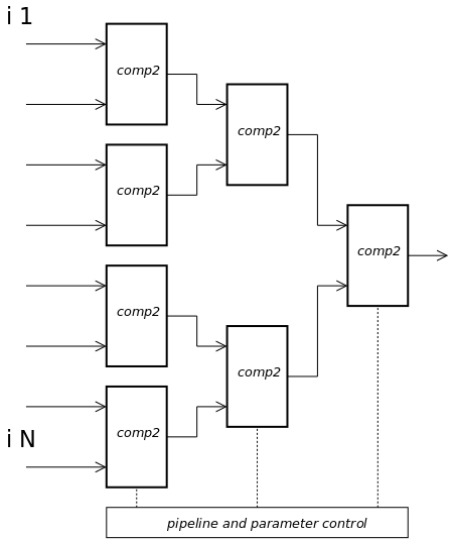
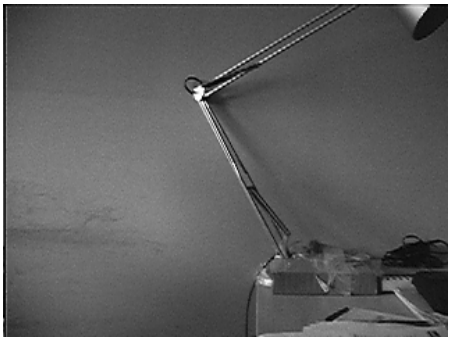
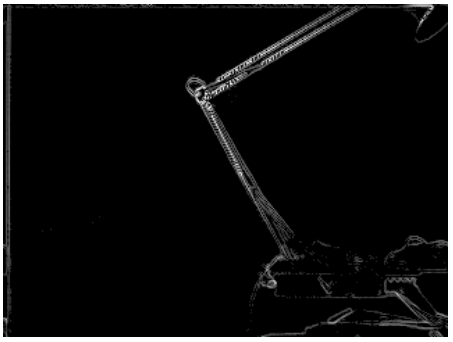


Figure 5.14: Internal compare unit ($N = 8$).

Each *comp2* stage performs an arithmetic comparison of two magnitude inputs (by deploying a subtraction operation) in one clock cycle. The entire pipeline is composed by a variable number of *comp2* stages, depending on the N value.

Finally we obtain the gradient magnitude and angle over a single pixel. This value is compared to a configurable threshold that acts as a low magnitude filter and then sent to the output.





In addition to the previously exposed features, in the last version of GradientHW a multiscale region thresholding has been inserted. By defining an area of pixels, called Region of Interest above the whole frame, we can insert a double thresholds, one for the gradient computed inside and one for the outside. Once a focal area has been selected, the GradientHW module can reduce the edge false positive by image segmentation. This tecnique has been inserted expecially for high frame video streams, where close frames are often correlated to each other.

5.4.5 Data Register Settings

Programmers using the IOWR, IORD instrcutions can access the GradientHW core directly via its registers. In future a HAL device driver will handle this controls. In general, the register map is useful to programmers writing a device driver for the core. It is worth to underline that every modification of these registers take place in the next clock cycle after the operation, thus the reconfiguration takes two clock cycles to be handled.

Tab. 5.9 shows the register map for the GradientHW core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 5.6: GradientHW Register map.

Offset	**Name**	**R/W**	**Descriptions of Bits**
0x00	Status Control	RW	ROIEN[2], BIN[1] , ON[0]
0x01	ThresholdIN	RW	Threshold inside ROI [31:0]
0x02	ThresholdOUT	RW	Threshold outside ROI [31:0]
0x03	ROI_START	RW	RoiSTART[31:0]
0x04	ROI_END	RW	RoiEND[31:0]

Status Control Register

The Status Control Register (SCR) consists of individual bits that indicate conditions inside the GradientHW controller. The SCR can be read at any time. Reading the SCR does not change its value. The SCR bits are shown in Tab. 5.7.

Table 5.7: Status Control Register Bit.

Bit	Name	R/W	Description
0	ON	RW	If the ON bit is 1, the GradientHW core captures new input datas. otherwise, the core performs an internal reset and does not generate any output. Default 0.
1	BIN	RW	If the BIN bit is 1, after the comparison with the threshold the magnitude output data will be binarized, 255 if is bigger, 0 otherwise. If the BIN bit is 0, compare stage does not modify the magnitude. Default is 0.
2	ROIEN	RW	If the ROIEN bit is 1, multithreshold gradient is enabled. Otherwise only ThresholdIN will be considered. Default is 0.

ThesholdIN Register

The ThesholdIN Register (TIR) specifies the threshold considered inside the ROI. If ROIEN bit is disabled, TIR is considered only. Default value is 35.

ThesholdOUT Register

The ThesholdIN Register (TOR) specifies the threshold considered outside the ROI. If ROIEN bit is disabled, TOR is not considered. Default value is 35.

ROI_START

The ROI_START Register specifies the start pixel of the ROI. It is computed following the raster scan method. First pixel is located in the top left corner. Default value is 0.

ROI_END

The ROI_END Register specifies the last pixel of the ROI. It represents the bottom right corner of the ROI. Default value is 0.

5.5 HistogramHW

Real-time histogram computation is used in a wide variety of applications, such as for image enhancement in photography and imaging, in speech recognition and audio systems, in tracking and surveillance and for biomedical applications such as DNA sequencing.

The histogram computation is a rather simple operation but induces a strong data dependence. In general, a simple histogram computation can be represented as depicted in Fig. 5.15.

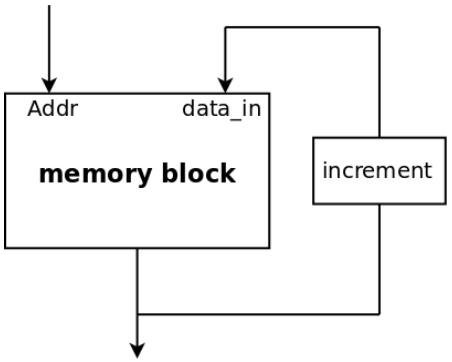


Figure 5.15: Histogram computation overview.

Most often, histogram analysis is performed serially in software written for embedded processors using a memory array. For parallel computation, access to the memory array creates read and write conflicts when two or more data items access to the same bin.

In general, the main challenge for a parallel histogram using a memory array is to handle updates to a particular bin count when at least two data items map to the same bin. Memory accesses collisions represent a upper bound limit in the real-time performance of parallel histogram computation. Since a bin is being updated the actual value may not be stable for at least 2 clock cycles. If the next data belongs to the same bin, there we will have a conflict (see Fig. 5.16).

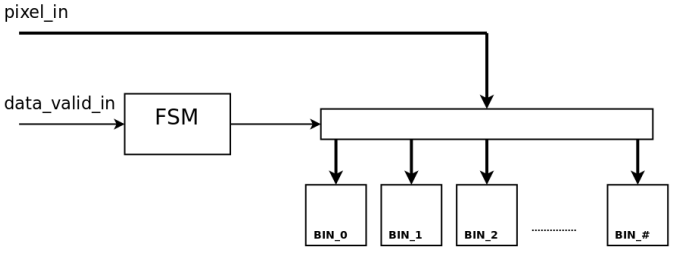


Figure 5.16: Histogram bins management.

To avoid data conflic a memory management system is needed even though it

will penalize the elaboration performance. Therefore a great effort has been put on redesigning algorithm for parallel computation.

In the next Section the state-of-the-art solutions on histogram parallel computation are shown. Then we will propose, implement and evaluate our parallel solution against the previously existent.

5.5.1 Existent hardware implementations

Real-time histogram computation has been extensively applied on signal processing and analysis in digital devices, for instance digital camera, video surveillance and mobile processing.

At the first glance the histogram calculation module is regarded as composed of memory blocks and incremental logic as it is shown on Fig. 5.15. In this picture a schematic view illustrates the main issues of a histogram computation module: (i) addressing management through input data, (ii) incremental step and (iii) histogram updating. Unfortunately, as it is the memory block limits the calculation speed as the memory output data is one clock cycle delayed with respect to the address bus (as a synchronous memory read operation). Therefore the evaluation of a single input pixel involves two clock cycles.

In order to speed up the above presented circuit to an evaluation per clock cycle, a dual port RAM solution has been presented in [42] and [43]. Dual port memory arrays are the common solution due to functional histogram behaviour. For example, the atomic operation to update a bin count is split into a read step and a modify-write step.

The data items are read into one port at one speed into a portion of a memory array while the modify-write works on a second port at **twice** the speed into a second portion of the same memory array. However, such systems can only achieve a speedup factor of up to two [44]. Similar approaches have been proposed for low speed data throughput, by [43]. This solution rely on the different clock speed between data input and FPGA system clock. While the single input period is substantially bigger than clock period (double or more), all the histogram operations have sufficient time to be executed. Therefore memory conflicts are avoided. Besides this architecture is based on the low input throughput compared to the system clock. Even if could work with low frame rate, is not suitable expecially for high resolution data streams. The condition between clock speed set an upper bound on the input data throughput.

Recently, new parallel histogram architectures have been proposed [45], [46]

and [47]. These works develop circuits for the computation of histograms using generic parallel architectures that process up to four data items at a time without involving memory arrays. The architectures are arranged as a linear array of cells where computation proceeds in a pipelined fashion. This has the advantage that it will keep more than one cell active per unit of time; having only one cell active at a time is the main drawback of the traditional histogram using an array of counters. Once all the data items are consumed and the histogram is computed, the final bin counts are read one bin at a time from one end of the array also in a pipelined fashion [45]. In this circuit two or four input data items per clock cycle are processed in parallel by the array, resulting in speedup factors of two or four respectively when compared to processing one input data item per clock cycle.

This new solution overcome the previously mentioned approaches. It has been implemented in modest hardware and can achieve real-time histogram computation when streaming high resolution images at 30 frames per second. However still represents a dedicated solution and is designed to process already available datas.

5.5.2 Our proposal

Our proposed implementation is based on the streaming fashion. Since storing a whole frame rate is not feasible for mid-range FPGA, the operation has to be done directly on the input pixel. Streaming histrogram computation has been exploited by a parallel processing and great effort has been put to accomplish a seamless integration in the CameraOneFrame architecture.

As pointed out by [41], parallel histogram computation requires a complete algorithm redesign. In this context, our work removes the memory access conflicts from the main histogram computation leading to more flexible and efficient ways to exploit parallelism. This opens up the possibility for better performances with respect to the state-of-the-art solutions and a configurable behaviour for a wide range of applications.

In Fig. 5.17 the external features of the **HistogramHW** module are shown. The module has an input port for data stream and an outport port for histogram storing. As the previously presented GradientHW, HistogramHW presents two different section, the top one is dedicated to histogram processing, while the bottom one represents the interface towards the Avalon Memory Mapped bus.

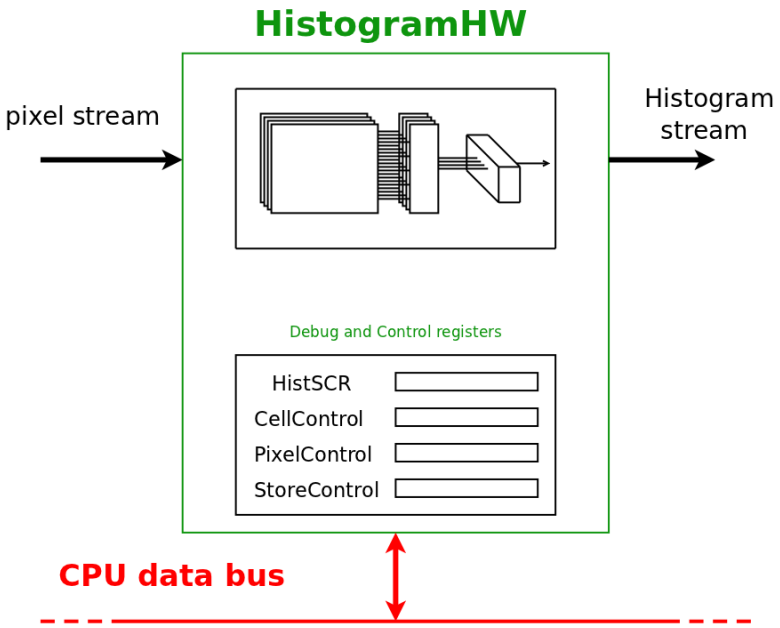


Figure 5.17: HistogramHW overview.

5.5.3 Exploiting parallelism

The proposed mechanism can be used to perform a generic histogram while reading image data directly from an external source for real-time application.

This module is designed to:

- calculate window-based histogram for real-time
- allow a parameter configuration
- process data directly when it is captured
- be fully compliant with CameraOneFrame architecture
- minimize the output data latency

Based on memory array design fashion [43], is built around the *RAMsubcell*, which is a dual port memory and represents the core of the circuit. The *RAMsubcell* provides the functionality exposed on Fig. 5.15. More in detail, **RAMsubcell** uses the data input as memory address (called *rdaddress_input* in the picture below), while considers another input as incremental value, that could be 1 or an arbitrary value (the reason will be explained in Chapter 6).

The memory array could be configured based on the requested bin resolution, bin widths and window dimension. When a pixel comes to the input port, a data valid signal will be asserted and the internal FSM begin a new elabration cycle. An

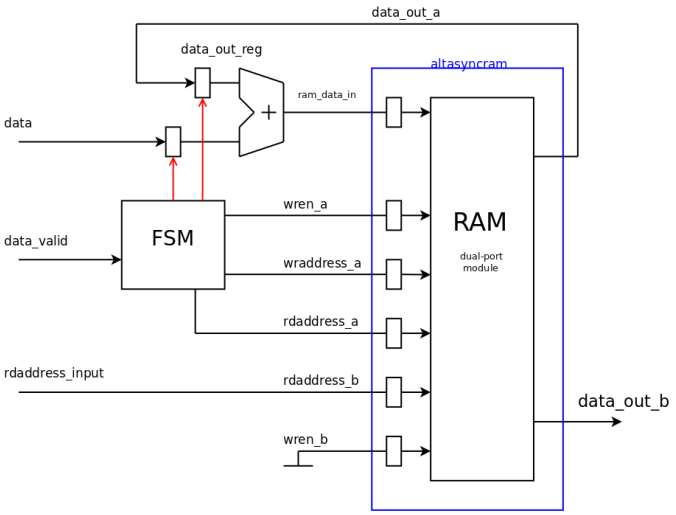


Figure 5.18: RAMsubcell module.

elaboration phase is composed by four states, each one last a clock cycle: *READ*, *LOAD*, *ADD*, *WRITE*. These states control the read-modify-write operation and guarantee a complete registered design, in order to speed up as much as possible the clock speed.

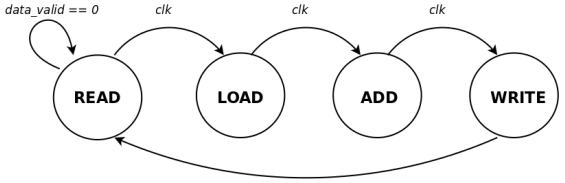


Figure 5.19: RAMsubcell FSM.

This circuit offers a reliable solution to the read-modify-write operation but does not resolve the memory access conflict problems and moreover takes four clock cycles to process a single pixel. As it is this circuit does not provides the requested atomic read-modify-write operation. Since a single histogram computation takes that time a real-time elaboration is not feasible in FPGA.

In Fig. 5.20 the pixel flow is shown. The RAMsubcell that is processing the first pixel does not capture a new data until the previous cycle is completed thus the represented behaviour is not feasible with a single RAMsubcell module.

The picture suggest a different approach: since the RAMsubcell is locked for four cycles, we can imagine an architecture composed by four instances of RAMsubcell, where each one takes care of one pixel at time. This structure has been called *Histogram cell*. The results is the follow: RAMsubcell first instance takes *pixel 1*, second instance *pixel 2*, and so on. Then after the WRITE phase the first RAMsubcell becomes available again and takes pixel 5. This sequential operation

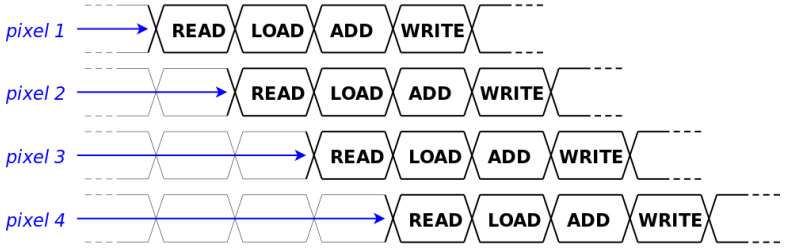


Figure 5.20: Streaming pixel flow.

is repeated over the whole frame.

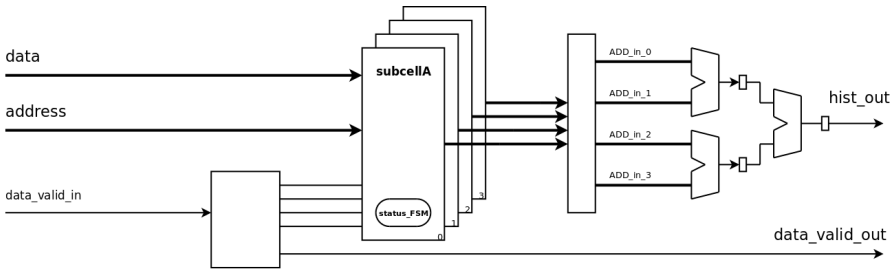


Figure 5.21: Histogram cell overview.

As described in the previous paragraph, this simple architecture can handle new input data every clock period and then processes directly a video stream while reading image data directly from an external source. When the frame has been completely acquired, the histogram bins are sent to the output. Every RAMsubcell bin contains a quarter part of the histogram, thus a data aggregation operation is performed before the storing phase.

As a result, RAMcell module provides a full-scale image histogram by processing a pixel per clock cycle. By controlling the memory locations of RAMsubcell, the system can handle different images resolution and different histogram bin resolution.

For instance, a 320x240 image resolution, with 256 bins (8-bit grayscale pixel) and 32bit width bins takes 256x16bitx4 = 2048 Bytes and about 100 LE on a Altera Cyclone IV EP4CE22 FPGA. Integrated in the CameraOneFrame architecture, this module can handle a real-time frame rate while introduces a latency of only ten clock cycles due to the module design.

5.5.4 Window-based histogram

A more complex behaviour of HistogramHW module is represented by the window-based operations. As introduced before, histograms are deployed as analysis

tool between consecutive frame to detect luminance or colors variations. Histograms are being deployed also for change detection algorithms expecially for video-surveillance applications [48].

Histogram based change detection algorithms have been used for years even with limited resources devices. They can analyse two or more consecutive frame and predict whereas a scene change has happened. This behaviour becomes extremely useful in video-surveillance scenarios, where a significant scene variation could trigger a video acquisition rather than a pre-allarm notification.

In more recent years, histogram analysis have been deployed also for comparing images based on their content or features [49] [50]. There a global histogram computation is useless, while it has been preferred a region histogram extraction. A region, often called window, is detailed by its shape and the amount of pixel contained.

Region based methods are better than local methods at capturing pixel interrelations. Region based histograms provide a rich estimate of a regions intensity distribution and then advantageous for features detection [51] or local background change [50]. Moreover can be deployed to provide an optimal color equalization for High Dynamic Range images, where a region based analysis gives better results [41].

Figure 5.22: Cell-based image.

Computing a cell-based histogram is similar to the previous shown method but require a considerably amount of data memory. This is due to the fine-grain histogram computed over a single cell. For instance with a cell of 8x8 pixels over a QVGA images we get:

1200 windows $\cdot$ 256 bins $\cdot$ 32 $\frac{bit}{bin}$ $\cdot$ 4 $\approx$ **4,9 MByte**

That is far to big even for high-end FPGA!

Our window-based approach

We propose an extended feature of the previously presented circuit which is capable to handle configurable histogram cell with limited memory requirements and minimum output latency. In our work, a cell has a configurable size. As a result 4x4, 5x5, 6x6 and 8x8 pixels cells are available for histogram calculation. Every cell histogram has an indipendent memory location and it is processed indipendently.

The structure of a cell histogram circuit is quite the same explained in the Section before. It uses a *RAMsubcell* module, along with a more complex FSM that controls the memory addressing method. Indeed histograms of close cells are stored as a sequential locations in the memory array.

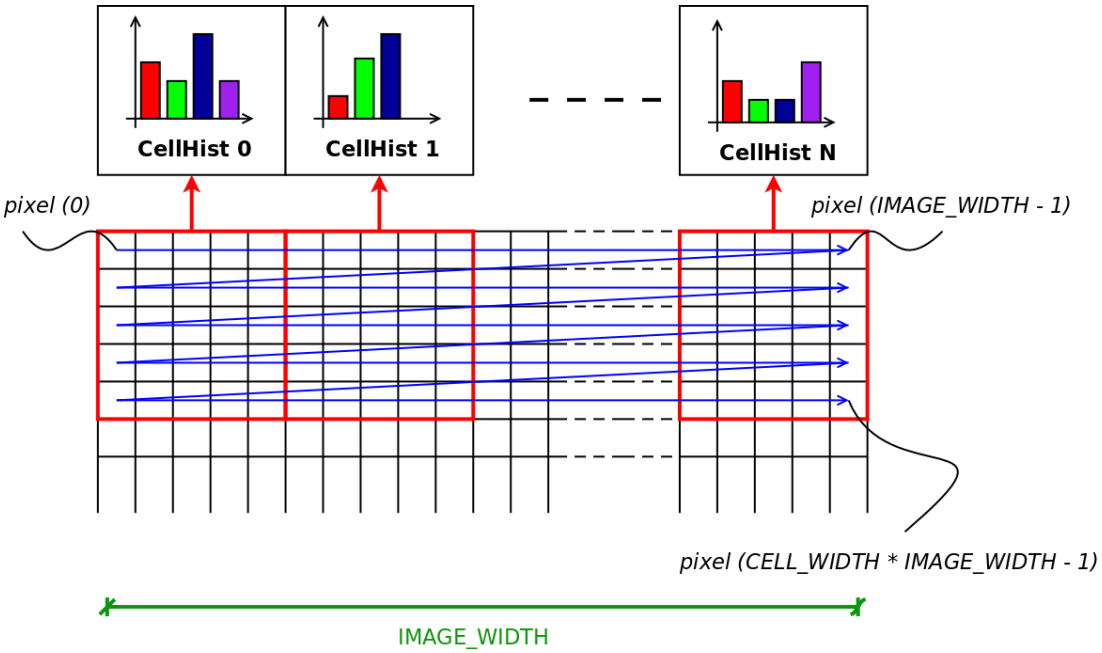


Figure 5.23: Cell-based Histogram computation.

They are computed during the acquisition phase an kept until the operation on the current cell is completed. In Fig. 5.23 the cell histogram computation is shown. The circuit keeps on memory only the active row of cell and process the incoming pixel addressing the correct memory location.

Since all of the CellHist block are instances of *RAMsubcell* a continuous data stream is handled. Memory conflicts are managed by the internal architecture thus, once again, the circuit works on a single clock period per pixel.

Fig. 5.23 introduces some configuration parameters:

IMAGE_WIDTH the horizontal width per units of pixels. In case of QVGA images, IMAGE_WIDTH is 320.

CELL_WIDTH the pixel size of the cell. At the moment, only square shapes are considered. In Fig. 5.23 CELL_WIDTH is 5.

The considered row cell is composed by the cells which are being updated by new pixel values. The row cell width is correlated to the CELL_WIDTH and IMAGE_WIDTH parameters. For instance, with CELL_WIDTH and IMAGE_WIDTH respectively equal to 5 and 320, the ROW_WIDTH becomes 320·5 = 1600 pixels. These parameters are important to setting up the HistogramHW module and to understand how the hardware configuration works.

Once the last pixel on the active cell row is processed (namely the 1599-th pixel considering the previous example) the memory allocated histograms are ready to be serially sent to output.

This operation called "store phase" is managed by the internal HistogramHW FSM. Unfortunately incoming pixels are still captured whereas during "store phase" the memory is unavailable. This drawback has been resolved using a shadow memory structure, which takes new data while the other one is sending out the results (see Fig. 5.24).

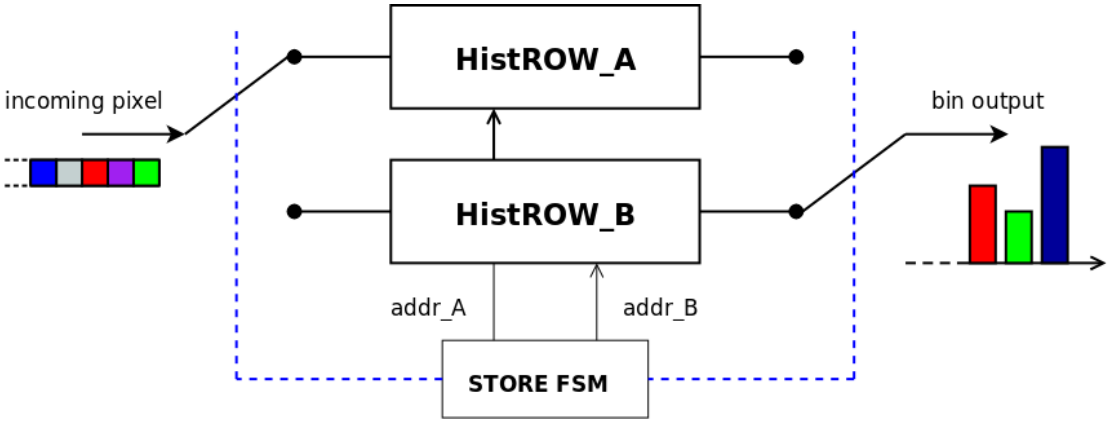


Figure 5.24: Shadow memory handling.

While HistROW_A is capturing the input stream, the other set called HistROW_B, is sending out the bin values. A brief explanation of the STORE FSM operations follows:

- controls the stream redirection to the histogram modules
- generates the memory address for read operations
- clears the previously read location (for a new acquitistion cycle)
- handles the RAMsubcell pipeline

Once the store phase has been completed, the HistRow involved is paused waiting for a new active status.

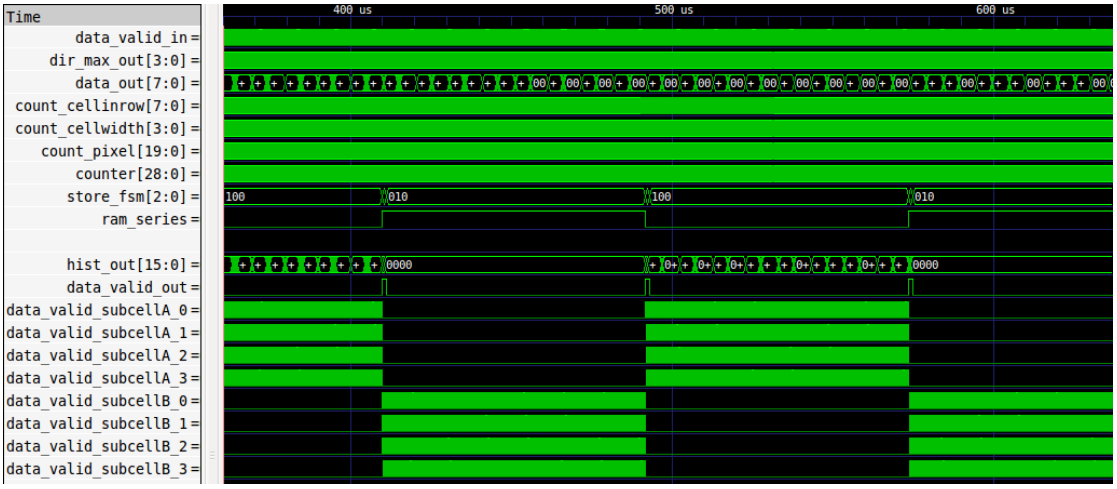


Figure 5.25: HistogramHW functional simulation.

in Fig. 5.25 the communication burst during the store phase are shown. The *data_valid_out* signal is asserted when a histogram row is being sent to the output.

This hardware architecture has been designed to hide the internal circuits to the output in order to be easlily implemented as a single entity in the CameraOneFrame architecture. Every parameters are available to the CPU as configurable resources, changeable at run time.

5.5.5 Evaluation

Since the architecture has been designed for wide range application, global and window based histograms are exploited by the same hardware module, through software parameters setup. Indeed the global based histogram can be seen as a full scale image window.

The HistogramHW module has been tested on a cycle accurate Verilog simulator and then implemented on a low cost FPGA. The test has been performed with a QVGA video stream, at 24fps.

Before reporting any results, some clarifications are needed:

- HistogramHW is designed for low-cost FPGA devices where limited amount of RAM is implemented. As a result, many intances can be implemented even on a rather small FPGA

- all of the time constraints are respected up to 100MHz clock frequency

	Available bins	Delay latecy	FPGA resource occupancy
4x4	8, 16	1280 T clk	∼ 16kB + 850 LE
5x5	8, 16	1600 T clk	∼ 16kB + 850 LE
6x6	8, 16	1920 T clk	∼ 16kB + 850 LE
8x8	8, 16, 32, 64 [1]	2560 T clk	∼ 16kB + 850 LE
full frame	16, 32, ..., 256 [1]	76800 T clk	∼ 16kB + 850 LE

Table 5.8: Performance results on QVGA resolution.

- resource occupancy and delay latency performances are measured with the Altera Design Tools and verified for Altera FPGA only.
- delay latecy cycles are computed assuming that a countinuos video stream is captured.

In Tab. 5.8 the preliminary performance results are shown. In particular in the second column are presented the available bin resolutions for a certain cell width. Since *store phase* is performed while the other shadow memory is capturing new input data, it is straightforward to notice that both phases should last the same time. Since active phase can not be bounded (it depends on the input stream), we must assure that the other one completes before that.

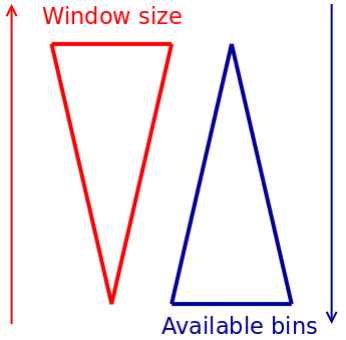


Figure 5.26: Bins and window size relation.

This condition becomes a restriction in the number of bins avalilable with respect to a defined window size. As a result, we obtain an inverse proportional relationship between the window size and the amount of bins. A simple numeric example explain this behaviour: using a 5x5 cells we get 25 incoming pixels and 16 available bins. Since the number of bins is lower than the amount of pixels considered, the *store phase* last less than the active phase. Differently, using a

[1] not tested yet

32 bins resolution, the circuit is not able to handle the stream and data will be corrupted.

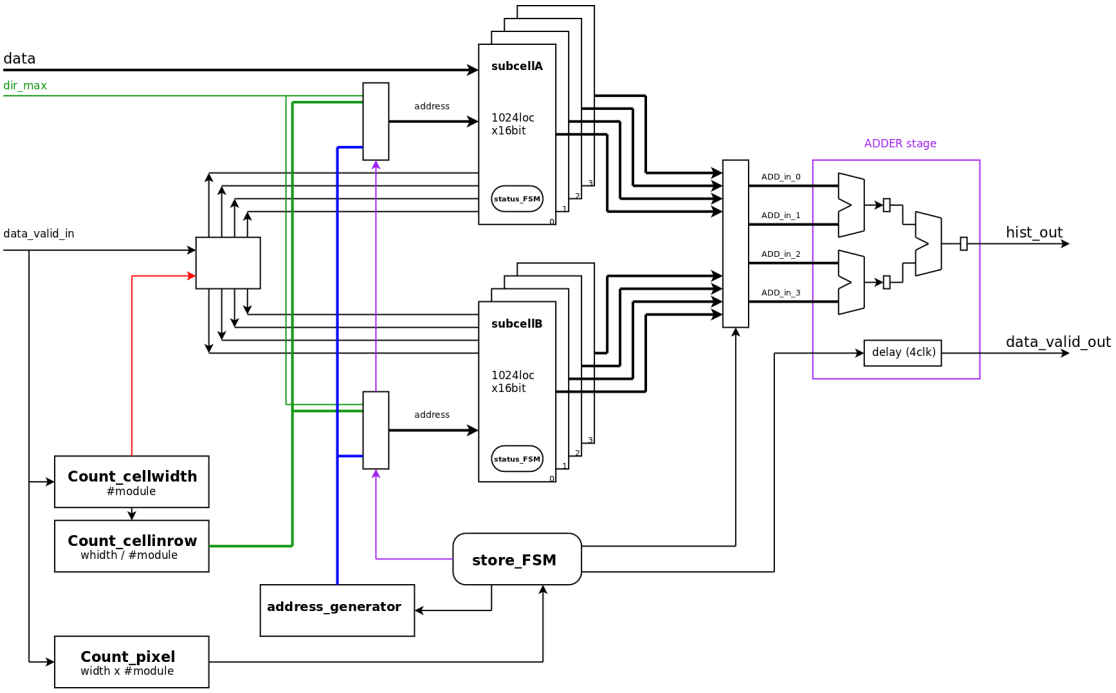


Figure 5.27: HistogramHW overview.

5.5.6 Data Register Settings

Programmers using the IOWR, IORD instrcutions can access the HistogramHW core directly via its registers. In future a HAL device driver will handle this controls. In general, the register map is useful to programmers writing a device driver for the core. It is worth to underline that every modification of these registers take place in the next clock cycle after the operation, thus the reconfiguration takes two clock cycles to be handled.

Tab. 5.9 shows the register map for the HistogramHW core. Device drivers control and communicate with the core through the memory-mapped registers.

Table 5.9: HistogramHW Register map.

Offset	Name	R/W	Descriptions of Bits
0x00	Status Control	RW	CountPixel[31:16], AddrMgt[1] , ON[0]
0x01	CellControl	RW	CellInRow[23:16] , CellWidth[15:0]
0x02	PixelControl	RW	PixelCellRow[31:0]
0x03	StoreControl	RW	AddressGenerator[31:0]

Status Control Register

The Status Control Register (SCR) consists of individual bits and group bits that indicate conditions inside the HistogramHW controller. The SCR can be read at any time. Reading the SCR does not change its value. The SCR bits are shown in Tab. 5.10.

Table 5.10: Status Control Register Bit.

Bit	Name	R/W	Description
0	ON	RW	If the ON bit is 1, the HistogramHW core captures new input datas. otherwise, the core performs an internal reset and does not generate any output. Default 0.
1	AddrMgt	RW	If the AddrMgt bit is 1, the input datas are treated as address and the histogram incremental step becomes 1. Otherwise data are treated as intremental step. Default 0.
31:16	CountPixel	RW	CountPixel keeps the count of the captured pixel in the last active operation (until ON is active).

Cell Control Register

The Cell Control Register (see Tab. 5.11) specifies the number cells and their dimensions. The CellWidth register is specified in pixels while CellInRow per unit of cell. With AddrMgt bit manages the histogram incremental values.

Table 5.11: Cell Control Register Bit.

Bit	Name	R/W	Description
15:0	CellWidth	RW	In CellWidth the cell size (per unit of pixel) are kept. Note that this configucration can be modified only when ON bit is low. Default 8.
23:16	AddrMgt	RW	If the AddrMgt bit is 1, the input datas are treated as address and the histogram incremental step becomes 1. Otherwise data are treated as intremental step. Default 0.
31:16	CellInRow	RW	Through CellInRow the user can configure the cell contained in a frame row (e.g. QVGA resolution, 8x8 cell gives CellInRow equal to 40). Defult 40.

Pixel Control Register

The Pixel Control Register (PCR) specifies the amount of pixels expected by the HistogramHW module inside a row of cells, minus 1. This value is useful to control the machine status and it is used only during the active phase. The PCR is wide enough to manage high resolution images. The default value is 2559 (QVGA frame, with 8x8 pixels cell).

Store Control Register

The STore Control Register (STCR) specifies the whole amount of bins to be tranferred during the "store phase", minus 1. This value is the combination of: number of bins per cell and the number of cells in a single row. The default value is 639 (QVGA frame, 8x8 pixels cell).

Chapter 6

Experimentation

In this chapter a working prototype for the CameraOneFrame architecture is presented. The implementation follows the design principles described in the Chapter 4. In order to evaluate the performances of CameraOneFrame architecture and validate the proposed methodology, a functional simulation and a prototype platform will be presented.

In Sec. 6.1 the System on a Programmable Chip (SoPC) subsystem is exposed while Sec. 6.2 will expose the development boards used for evaluation and testing. Finally, in Sec. 6.3, some different algorithm implementation are shown along with timing and resource occupation performances.

6.1 Image processing SoPC

As described in Chapter 3, the CameraOneFrame architecture permits to have a greater flexibility with respect to a pure hardware based solution. This is a consequence of using an embedded softcore CPU, which handles the connection and the reconfiguration abstraction. In this first design step, we would create the connection between the peripherals and the CPU core.

Once an Elab block has been designed, a new Qsys component has to be created. A Qsys component is created directly from the HDL code and becomes a model for multiple instances of the same hardware IP. By using the Qsys builder tool, the designer works on a hardware abstraction level, where a specific instance is addressable as a memory mapped location.

In Fig. 6.1 the implemented SoPC circuit is shown. In red the Avalon-MM bus is shown, while the arrows in black represents a unique address location, available on the NiosII Data Bus. In the image, special consideration is given to

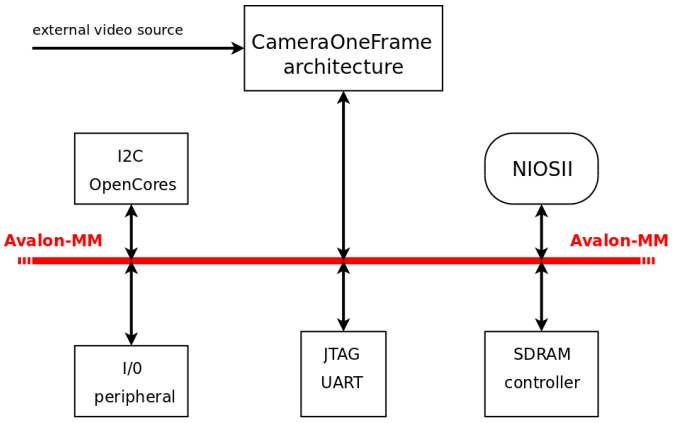


Figure 6.1: SoPC overview.

the NiosII environment, showing some hardware instances offered by Altera with the QuartusII release. Basically an OpenCores I2C module is deployed to manage the camera configuration bus, a memory controller for SDRAM management and a JtagUART for debugging purposes.

The CameraOneFrame instances internally is composed as described in Chapter 3. Here a certain number of RouteMatrix and Elab block instances realise the desired configurable computer vision pipeline. In Tab. 6.1 the resource occupancy of a single Hardware Library instancy is shown.

Considering the limited data bandwidth requested to the CPU, the smaller NiosII instance has been deployed. Indeed, the economy version provides the necessary computational power while occupies only 600 Logic Elements and is provided by Altera with a royalty-free license.

6.2 Development board

The considered development board are principally two: the low-cost Terasic DE0-nano board and the DreamCAM designed by the University of Clermont Ferrand.

Table 6.1: Hardware library occupancy.

	Logic elements	RAM footprint (Bytes)	DSP (9x9 bit)
VideoSampler	200 (0,9%)	512	0
RemoteImage	200 (0,9%)	1	0
GradientHW	1200 (5,4%)	640	32
HistogramHW	850 (4,0%)	16384	0
RouteMatrix (x1)	400 (1,8%)	40	0

This two solutions represents different target application: the former is a extremely low cost solution with a limited hardware resource, ideal for simple and pervasive scenarios. The latter represents a high end product, where resources and memory are comparable with the newest state of the art devices.

The Terasic DE0-nano board embeds an Altera Cyclone IV FPGA, with 22000 Logic Elements (LE), 600 Kbits of on-board memory and 144 9x9 bits DSP modules. In such board we deploy a 1.3MP Omnivision CMOS Camera (OV9650) connected through the header connections.

In Fig. 6.2 the above presented solution is shown. The communication with PC is managed by the USB cable, which is used also for FPGA programming and NiosII software download.

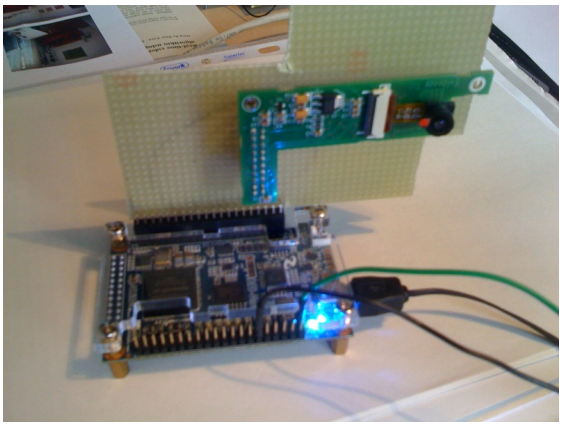


Figure 6.2: DE0-nano with OV9650 board.

The DreamCam platform is equipped by a 1.3 Mega pixels active-pixel digital image sensor from E2V, supporting sub-sampling/binning and multi Region of Interests (ROIs). The FPGA is a Cyclone-III EP3C120 FPGA manufactured by Altera. This FPGA is connected to 6x1MBytes of SRAM memory blocks wherein each 1MB memory block has a private data and address buses. The board also embeds a CMOS sensor and a communication controller layer for USB

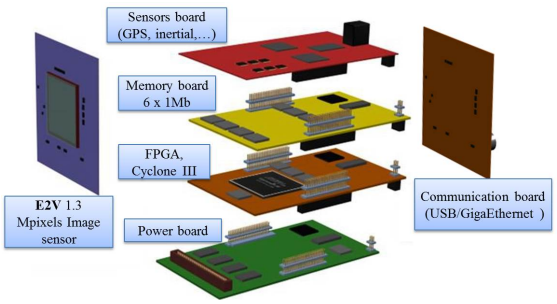


Figure 6.3: $DreamCAM^3$ layers.

or Giga-Ethernet interface. To assure future board extension, the designer places an expansion slot on top of each PCB, able to accomodate a potential new design.

Figure 6.4: The $DreamCAM^3$ board.

6.3 Architecture evaluation

In this section we propose a suite of test cases aimed at validating all the claims described in the previous sections by using the modules presented in Chapter 5.

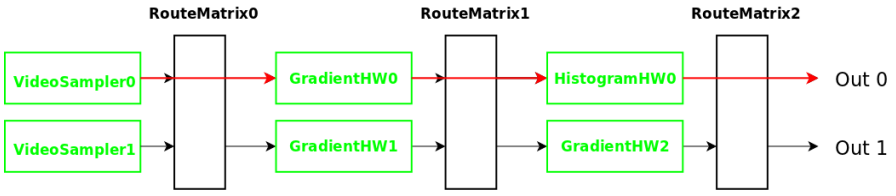
6.3.1 Test cases

More in detail, in the test cases (shown in Figure 6.5) the following modules have been instantiated: two VideoSampler, three GradientHW, a HistogramHW, three RouteMatrix instances and finally a NiosII SoftCore. In this scenario the proposed architecture permits to generate several combinations of the considered blocks, thus performing a set of applications fitting inside a rather small FPGA size.

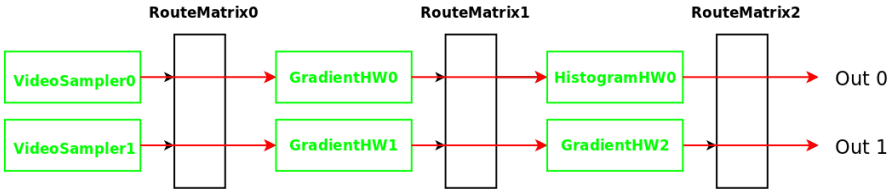
In fact, considering the occupancy data reported in Tab. 6.1, and considering the additional resource overhead due to the NIOSII SoftCore and the Avalon-MM bus, the whole bitstream occupies only 31% of the LE, the 38% of on-chip memory and the 51% of the 9x9 DSP modules.

The all possible cases of the test suite are reported in Figure 6.5, where it is shown that the proposed architecture can: (i) handle a single pipeline (Figure 6.5a), (ii) handle two parallel pipelines using two cameras (Figure 6.5b) and (iii) split a data stream to follow two pipelines (Fig. 6.5c).

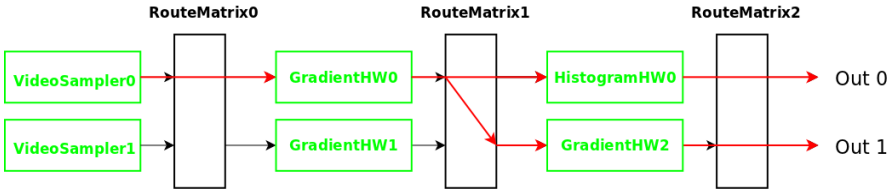
In order to evaluate the output latency of each pipeline, the time latency of each architecture block is shown in Table 6.2 in terms of FPGA clock cycles.



(a) Test 1: handling a single pipeline.



(b) Test 2: handling two pipelines.



(c) Test 3: split a data stream to follow two pipelines.

Figure 6.5: Test cases.

These values measure the amount of clock cycles needed by data to flow inside each block, considering that they are implemented according to the streaming paradigm, without buffering a whole image.

Table 6.2: Hardware module processing latency.

	Latency (clock cycles)
VideoSampler	0
RemoteImage	0
GradientHW	2
HistogramHW	2560
RouteMatrix	1

As reported in the table above, every module has a constant data delay, as a consequence of the hardware realisation based on the HDL description. More in detail, the VideoSampler and RemoteImage modules do not introduce any latency time, because they do not perform any elaboration, but only manage clock speed conversions. The GradientHW, instead, introduces a latency time 2 clock cycles. The most effective latency values are related to the HistogramHW module, that introduces a latency dependent on image size and cell size: in the contingent case,

using 8x8 pixels cells and Q-VGA images, it is of 2560 clock cycles. Regarding the last implemented block, RouteMatrix, it introduces a latency of 1 clock cycle only.

The overall system performance in time delay can be evaluated as a sum of the latency of the module inserted into a specific pipeline plus the one introduced by the architecture structure elements (namely the RouteMatrix), as shown in the following equation:

$$L_{total} = \sum_{i=0}^{N-1} L_i^B + M \cdot L^{RM} \tag{6.1}$$

where L_i^B is the latency of the i-th elaboration block, L^{RM} the latency of RouteMatrix, M the number of RouteMatrix iterations, and N the depth of the considered pipeline.

The latency time value in number of clock cycles can be easily converted in time delay by defining the oscillation frequency of the FPGA master clock. In our design, all the system runs at a frequency of 50MHz, thus the values of Table 6.2 can be expressed as delay time by multiplying them for the inverse of the clock frequency (in this case $40ns$). Table 6.3 shows the delays for all the test cases depicted in Figure 6.5. For all the performed experiments the latency in terms of clock cycles and delay time is evaluated following the rule described in Equation 6.1.

6.3.2 Histogram of Oriented Gradients

The above presented test case has been deployed to exploit the Histogram of Oriented Gradients algorithm (HOG) on FPGA. This algorithms, proposed by Dalal in [52] is particulary suited for pedestrian and vehicle detection. In [52] the authors propose a new feature extraction methods able to reduce the spatial complexity to a set of aggregated data. These results are sent to a classification machine (Support Vector Machine or a Neural Network) able to detect specific set

Table 6.3: Tests case results.

	Latency (clock cycles)		Time (μs)	
	Out_0	Out_1	Out_0	Out_1
Test 1	2565	-	102,600	-
Test 2	2565	9	102,600	0,360
Test 3	2565	9	102,600	0,360

of features after a specific training phase.

The HOG algorithm pipeline is shown in Fig.6.6.

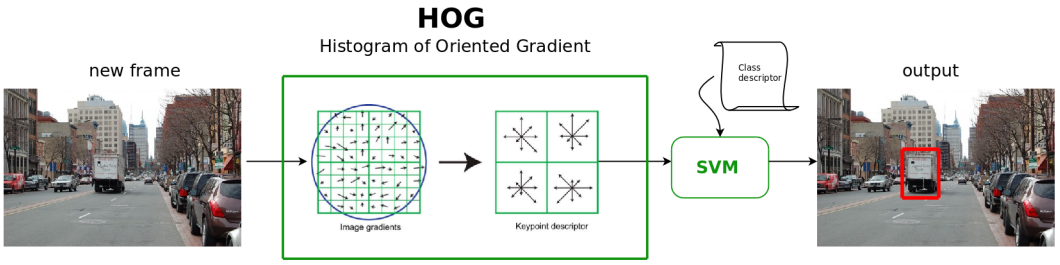


Figure 6.6: The HOG pipeline.

The HOG algorithm is similar to that of edge orientation histograms and scale-invariant feature transform (SIFT) descriptors, but differs in that it is computed on a dense grid of uniformly spaced cells and uses overlapping local contrast normalization for improved accuracy. The implementation of these descriptors can be achieved by dividing the image into small connected regions, called cells, and for each cell compiling a histogram of gradient directions for the pixels within the cell.

The combination of these histograms then represents the descriptor. For improved accuracy, the local histograms can be normalized by calculating a measure of the intensity across a larger region of the image, called a block, and then using this value to normalize all cells within the block.

This normalization results in better invariance to changes in illumination or shadowing [53].

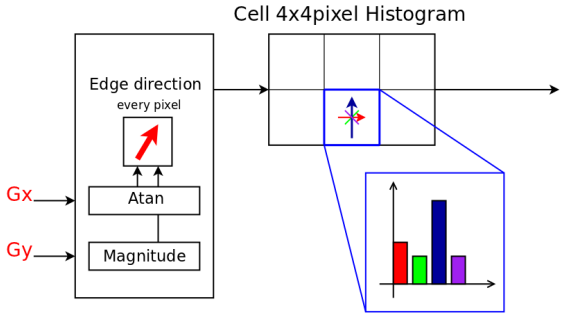


Figure 6.7: The HOG logic operation.

In the latest years, hardware implementations of HOG algorithm have been presented by [35], [38], [39]. In the above mentioned papers, the authors demonstates once more that FPGA is suitable for image vision processing. In particular, in [35] a real-time a simplified HOG extraction technique is deployed for road signal detection. In [38] and [39] an updated version has been presented, where a

custom hardware processing has been designed. Finally in [40] the latest hardware HOG implementation is shown. In this work, the authors presents optimised version of HOG extraction, based on a multistage pipeline elaboration.

Our HOG implementation represents a follow up of [39], where each elaboration step is performed by a single stage pipeline. By using the above mentioned elaboration blocks (as shown in Chapter 5), we implement a hardware-friendly version of the HOG algorithm. Indeed by using a VideoSampler, GradientHW, HistogramHW, StreamStore the histogram extraction has been exploited.

In Fig. 6.8 the HOG data flow modeling has been presented. Note that any RouteMatrix instances are used because the application could not require them. In this case, the pipeline becomes fixed once the bitstream is programmed but is still configurable by using the Avalon-MM interfaces of every hardware IP.

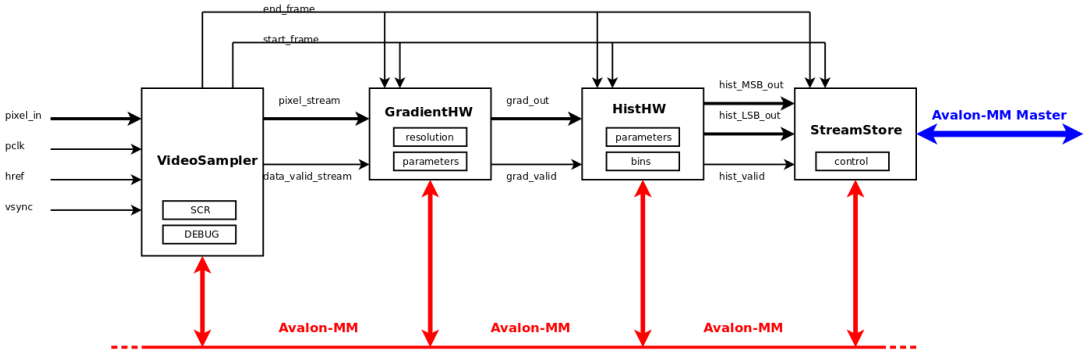


Figure 6.8: The CameraOneFrame HOG implementation.

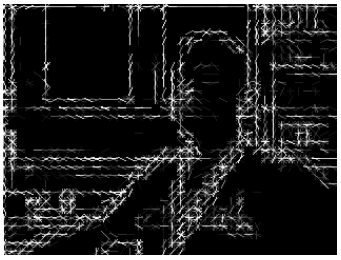
In Fig. 6.9 the results of this elaboration pipeline are reported. In particular, Fig. 6.9a is the original 320x240 (QVGA) frame captured from the OV9650 CMOS camera and Fig. 6.9b shows the results of a 8x8 pixels cell histogram.

By considering Tab. 6.1, the occupation results are contained and well-suited for the low-cost DE0-nano deployement. Considering the processing time performance, this hardware solution could handle up to 30fps at VGA resolution. In Fig. 6.10 a different implementation is presented. In this case we deploy two instances of RouteMatrix in order to extend the system configurability. Although the little resource overhead, using RouteMatrix the datapath can be redirected during the elaboration to provide intermediate results or parallel data flows.

In this latest case, a new source has been inserted. The RemoteImg can be connected to the HOG elaboration pipeline by a software configuration of the first RouteMatrix instance. This architectural solution can:

- acquire a frame from a CMOS camera or from a serial connection

(a) Original image.



(b) Histogram of Oriented Gradients results.

Figure 6.9: HOG output example.

- provide an original image as result
- provide a gradient image
- provide a histogram image
- provide a histogram of gradient image (by using the feedback loop)
- provide at the same time: original, gradient and histogram of gradient image. These results are given in parallel to the output.

6.4 FPGA-PC communication

The simple PC-NiosII interface used for development is based on the Altera JtagAtlantic driver released in the Quartus II software tool. The connection is managed as a serial communication throught the JtagUART peripheral. The Softcore CPU wait commands on the UART link and once received, configure the internal architecture and enable the acquisition.

The GUI interface has been coded in C with GTK graphic libraries. In Fig.6.11 the simple interface is shown. By a simple click on the button, it is possible to configure the internal architecture of the FPGA and obtain the requested result. The output results is then sent back from the FPGA to the PC. Considering the

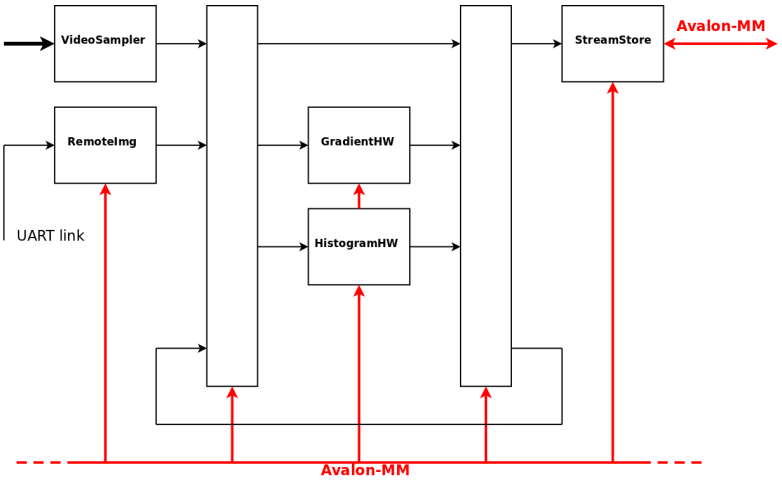


Figure 6.10: CameraOneFrame intermediate results.

Figure 6.11: Debug GUI.

limited bandwidth of the JTAG connection, a real-time frame updating is not feasible.

Chapter 7

Conclusion

In this master thesis a new architecture for supporting FPGA applications applied to the Smart Camera Network has been presented. Issues related to hardware and software processing have been discussed together with the benets of relying on hardware based image processing design.

In particular we present an innovative architecture concept for a smart camera network node. By leveraging the hardware software codesign, we propose a SCN node composed by a reconfigurable FPGA architecture. This architecture is targeted to a wide range of image processing applications, where real-time and low cost specifications require a dedicated system.

Concerning the FPGA architecture, the dynamic reconfiguration is managed by an internal Softcore CPU which acts as a data path controller. This architecture represents a flexible and reconfigurable solution into a static FPGA bitstream.

The flexibility and the reconfigurability is enabled to provide the possibility of composing computer vision pipelines at run-time, by routing data to certain elaboration blocks. Moreover, in the proposed design the computer vision algorithms are realized with modules of an HDL library, thus helping programmers in the development of IoT based applications leveraging computer vision capabilities.

In this work the proposed architecture is first presented by detailing the internal reconfigurable architecture. Then, a real implementation on FPGA devices is discussed in respect to a possible application. Finally, performance evaluation results are presented in terms of latency time and FPGA occupancy.

Results show that the high level abstraction realised by the architecture does not decrease the elaboration performance, indeed shows a constant output latency and an optimized solution, compared to a pure-software application.

7.1 Future work

The preliminary results obtained in this thesis are rather satisfying; although the few blocks available at the moment in the Hardware Library, some simple applications are already feasible.

In the near future effort will be directed to an extension of the Hardware Library, providing solution and hardware design to other image processing applications. Further, new updated version of the already available hardware module will be presented. In this respect the work will be directed to assure the maximum reconfiguration from a custom hardware IP.

With respect to the Smart Camera environment, in the future we envision a FPGA based Smart Camera where the Softcore CPU is able to manage both the architecture and a network stack. This scenario is the first step towards a real SCN deployment and leveraging on a distributed processing network able to exchange information or aggregated datas.

Bibliography

[1] B. Rinner, T. Winkler, W. Schriebl, M. Quaritsch, and W. Wolf. The evolution from single to pervasive smart cameras. In *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, pages 1–10, 2008.

[2] F. Qureshi and D. Terzopoulos. Smart camera networks in virtual reality. *Proceedings of the IEEE*, pages 1640–1656, 2008.

[3] Thomas Winkler and Bernhard Rinner. Pervasive smart camera networks exploiting heterogeneous wireless channels. In *Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, PERCOM '09, pages 1–4. IEEE Computer Society, 2009.

[4] iMinds. Little sister project. URL http://www.iminds.be/en/research/overview-projects/p/detail/littlesister.

[5] Stephan Hengstler and Hamid Aghajan. A smart camera mote architecture for distributed intelligent surveillance. In *In ACM SenSys Workshop on Distributed Smart Cameras (DSC*, 2006.

[6] Claudio Salvadori, Dimitrios Makris, Matteo Petracca, Jesus Martinez del Rincon, and Sergio Velastin. Gaussian mixture background modelling optimisation for micro-controllers. pages 241–251, 2012.

[7] Sek Chai Branislav Kisaanin, Shuvra S. Bhattacharya. *Embedded Computer Vision*. Springer, 2008.

[8] Martin Masek Michael Wirth, Matteo Fraschini and Michel Bruynooghe. Performance evaluation in image processing. *EURASIP on Applied Signal Processing*, pages 1–3, 2006.

[9] W.J. MacLean. An evaluation of the suitability of fpgas for embedded vision systems. In *Computer Vision and Pattern Recognition - Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*, 2005.

[10] M. Birem and F. Berry. Fpga-based real time extraction of visual features. In *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*, pages 3053–3056, 2012.

[11] Nasser Kehtarnavaz Kui Liu. Real-time robust vision-based hand gesture recognition using stereo images. *Journal of Real-Time Image Processing*, pages 1–9, 2013.

[12] Fabio Dias, Franois Berry, Jocelyn Srot, and Franois Marmoiton. A configurable window-based processing element for image processing on smart cameras. In *MVA*, pages 276–280, 2007.

[13] Texas Instruments Incorporated. TMS320C6414T, TMS320C6415T, TMS320C6416T Fixed-Point Digital Signal Processors. http://focus.ti.com/lit/ds/sprs226l/sprs226l.pdf, 2008.

[14] Nicolas Roudel. *SeePROC : Un modle de processeur chemin de donnes reconfigurable pour le traitement dimages embarqu*. PhD thesis, Universit Blaise Pascal - Clermont-Ferrand, 2008.

[15] Ben Cope. Implementation of 2d convolution on fpgas, gpus and a cpu. Technical report, Department of Electrical and Electronic Engineering, Imperial College London, 2005.

[16] C. Farabet, C. Poulet, and Y. LeCun. An fpga-based stream processor for embedded real-time vision with convolutional networks. In *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*, pages 878–885, 2009.

[17] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116, 2011.

[18] G. Van der Wal, M. Hansen, and M. Piacentino. The acadia vision processor. In *Computer Architectures for Machine Perception, 2000. Proceedings. Fifth IEEE International Workshop on*, pages 31–40, 2000.

[19] Gooitzen S. van der Wal. Technical overview of the sarnoff acadia ii vision processor, 2010.

[20] S. Kyo and S. Okazaki. In-vehicle vision processors for driver assistance systems. In *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pages 383–388, 2008.

[21] S. Kyo, S. Okazaki, T. Koga, and F. Hidano. A 100 gops in-vehicle vision processor for pre-crash safety systems based on a ring connected 128 4-way vliw processing elements. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 28–29, 2008.

[22] J. Clemons, A. Jones, R. Perricone, S. Savarese, and T. Austin. Effex: An embedded processor for computer vision based feature extraction. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 1020–1025, 2011.

[23] R.P. Kleihorst, A.A. Abbo, A. Van Der Avoird, M. J R Op De Beeck, L. Sevat, P. Wielage, R. Van Veen, and H. van Herten. Xetal: a low-power high-performance smart camera processor. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 5, pages 215–218 vol. 5, 2001.

[24] A. Abbo, R. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, and M. Heijligers. Xetal-ii: A 107 gops, 600mw massively-parallel processor for video scene analysis. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 270–602, 2007.

[25] Yu Pu, Yifan He, Zhenyu Ye, S.M. Londono, A.A. Abbo, R. Kleihorst, and H. Corporaal. From xetal-ii to xetal-pro: On the road toward an ultralow-energy and high-throughput simd processor. *Circuits and Systems for Video Technology, IEEE Transactions on*, pages 472–484, 2011.

[26] M. Boschetti, I. Silva, and S. Bampi. A run-time reconfigurable datapath architecture for image processing applications. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, pages 242–247, 2004.

[27] Jocelyn Serot, Francois Berry, and Sameer Ahmed. Implementing stream-processing applications on fpgas: A dsl-based approach. In *Proceedings of*

the 2011 21st International Conference on Field Programmable Logic and Applications, FPL '11, pages 130–137. IEEE Computer Society, 2011.

[28] Jocelyn Serot, Francois Berry, and Sameer Ahmed. Caph: A language for implementing stream-processing applications on fpgas. *Embedded Systems Design with FPGAs*, pages 201–224, 2013.

[29] Tam Phuong Cao, Darrell M. Elton, and Guang Deng. Fast buffering for fpga implementation of vision-based object recognition systems. *J. Real-Time Image Processing*, pages 173–183, 2012.

[30] Altera. Quartus II Software. http://www.altera.com/products/software/quartus-ii/about/qts-performance-productivity.html, 2012.

[31] *Avalon Interface Specifications*, 2013. URL http://www.altera.com/literature/manual/mnl_avalon_spec.pdf.

[32] Altera Corporation. Designing and using fpgas for double-precision floating-point math. URL http://www.altera.com/literature/wp/wp-01028.pdf.

[33] Chun Hok Ho, Chi Wai Yu, P. Leong, W. Luk, and S. J E Wilton. Floating-point fpga: Architecture and modeling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(12):1709–1718, 2009.

[34] Rudolf Usselmann. Floating point unit. URL http://opencores.org/project,fpu.

[35] Tam Phuong Cao and Guang Deng. Real-time vision-based stop sign detection system on fpga. In *Digital Image Computing: Techniques and Applications (DICTA), 2008*, pages 465–471, 2008.

[36] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware architecture for hog feature extraction. In *Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP '09. Fifth International Conference on*, pages 1330–1333, 2009.

[37] S. Schlotterbeck S. Bauer, U. Brunsmann. Fpga implementation of a hog-based pedestrian recognition system. In *MPC Workshop, IEEE Solid-State Circuit Society German Section*, 2009.

[38] K. Negi, K. Dohi, Y. Shibata, and K. Oguri. Deep pipelined one-chip fpga implementation of a real-time image-based human detection algorithm. In

Field-Programmable Technology (FPT), 2011 International Conference on, pages 1–8, 2011.

[39] Kosuke Mizuno, Yosuke Terachi, Kenta Takagi, Shintaro Izumi, Hiroshi Kawaguchi, and Masahiko Yoshimoto. Architectural study of hog feature extraction processor for real-time object detection. In *SiPS*, pages 197–202. IEEE, 2012.

[40] Seung Eun Lee, Kyungwon Min, and Taeweon Suh. Accelerating histograms of oriented gradients descriptor extraction for pedestrian recognition. *Computers and Electrical Engineering*, pages 1043 – 1048, 2013.

[41] B. Guthier, S. Kopf, M. Wichtlhuber, and W. Effelsberg. Parallel algorithms for histogram-based image registration. In *Systems, Signals and Image Processing (IWSSIP), 2012 19th International Conference on*, pages 172–175, 2012.

[42] Edgard Garcia. Implementing a histogram for image processing applications. URL http://www.hunteng.co.uk/pdfs/tech/xcell38_46.pdf.

[43] E. Jamro, M. Wielgosz, and K. Wiatr. Fpga implementaton of strongly parallel histogram equalization. In *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS '07. IEEE*, pages 1–6, 2007.

[44] A. Shahbahrami, J.Y. Hur, B.H.H. Juurlink, and S. Wong. Fpga implementation of parallel histogram computation. In *Proc. 2nd HiPEAC Workshop on Reconfigurable Computing*, pages 63–72, Gteborg, Sweden, 2008.

[45] J. Cadenas, R.S. Sherratt, and P. Huerta. Parallel pipelined histogram architectures. *Electronics Letters*, 47(20):1118–1120, 2011.

[46] G.M. Megson, J.O. Cadenas, R.S. Sherratt, P. Huerta, and W.C. Kao. A parallel quantum histogram architecture. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 60(7):437–441, 2013.

[47] Q. Gan, J.M.P. Langlois, and Y. Savaria. Parallel array histogram architecture for embedded implementations. *Electronics Letters*, 49(2):99–101, 2013.

[48] Haitao Jiang, Abdelsalam Helal, Ahmed K. Elmagarmid, and Anupam Joshi. Scene change detection techniques for video database systems. *Multimedia Syst.*, 6(3), 1998.

[49] Greg Pass and Ramin Zabih. Comparing images using joint histograms. *Multimedia Syst.*, 7(3):234–240, 1999.

[50] Haibin Ling and K. Okada. Diffusion distance for histogram comparison. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 246–253, 2006.

[51] Robert E. Broadhurst, Joshua Stough, Stephen M. Pizer, and Edward L. Chaney. Histogram statistics of local model-relative image regions. In *Proceedings of the First international conference on Deep Structure, Singularities, and Computer Vision*, DSSCV'05, pages 72–83, 2005.

[52] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893, 2005.

[53] T. Gandhi and M.M. Trivedi. Pedestrian protection systems: Issues, survey, and challenges. *Intelligent Transportation Systems, IEEE Transactions on*, pages 413–430, 2007.