UNIVERSITA' DELLA CALABRIA

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

Dottorato di Ricerca in

Information and Communications Technologies

CICLO XXXII

Heterogeneous FPGA-based Embedded Systems for Vision IoT Applications

Settore Scientifico Disciplinare ING-INF/01

Coordinatore: Ch.mo Prof. Felice Crupi

Firma oscurata in base alle linee guida del Garante della privacy

Supervisore: Ch.ma Prof.ssa Stefania Perri

Firma oscurata in base alle linee guida del Garante della privacy

Tutor: Ch.mo Prof. Pasquale Corsonello

Firma oscurata in base alle linee guida del Garante della privacy

Dottorando: Dott.ssa Fanny Spagnolo

Firma oscurata in base alle linee guida del Garante della privacy

Ad Antonio, ispirazione e sostegno nelle difficoltà.
A papà e Saverio, che mi hanno affiancata lungo tutto il cammino.
A Stefania e Pasquale che sono stati per me come una seconda famiglia.
Ma soprattutto alla mia dolce mamma, modello di vita.

ABSTRACT

Embedded sensor devices provided by processing capabilities are opening novel and exciting opportunities in the era of edge-computing Internet-of-Things (IoT). The workload decentralization leads to a plenty of benefits, including better reactivity and reliability and reduced data transfer costs. These advantages have a strong impact especially in the visual IoT field, for which the large bandwidth required by visual data is one of the most critical challenges. However, bringing vision technologies into smart nodes is not a trivial task, because of the stringent energy and performance requirements, in addition to the need of cost-effective and compact processing units. Heterogeneous architectures may represent the key to address these necessities. Among possible heterogeneous platforms, those based on reconfigurable devices such as Field Programmable Gate Arrays (FPGAs) show a high adaptability to a variety of workloads, which is an important goal for edge-computing. Therefore, their deployment in disparate IoT applications, ranging from video surveillance to autonomous driving, is emerging as a promising solution. This dissertation proposes a study on the suitability of modern heterogeneous FPGA System-on-Chips (SoCs) to implement embedded smart vision sensor nodes. To this purpose, several computer vision algorithms aimed to extract synthetic data from raw input frames have been analysed, and novel hardware-oriented solutions have been proposed to deploy them on heterogeneous SoCs. In all the presented cases, ranging from stereo vision to connected component analysis and deep learning, speed performances and/or energy efficiency are considerably improved with respect to state-of-the-art solutions. As an example, the proposed heterogeneous architecture for convolutional neural networks achieves a power efficiency up to 89.5% higher than competitive prior works, demonstrating its suitability in the scenario of energy-constrained and real-time IoT.

RIASSUNTO

I moderni sensori provvisti di capacità di elaborazione stanno aprendo nuove ed eccitanti opportunità nell'era dell'Internet of Things (IoT) basato su edge computing. Il decentramento del carico di lavoro dal centro cloud ai singoli nodi sensore comporta numerosi vantaggi, tra cui una migliore reattività e affidabilità e costi ridotti nel trasferimento dei dati. Questi vantaggi hanno un forte impatto soprattutto nel campo dell'IoT visivo, dove l'ampia larghezza di banda richiesta dai dati è una delle sfide più importanti. Ad ogni modo, dotare i nodi sensore di intelligenza propria non è un compito banale, a causa delle rigorose specifiche energetiche e prestazionali richieste nell'ambito dell'edge computing, oltre alla necessità di utilizzare unità di elaborazione economiche e compatte. Le architetture eterogenee possono rappresentare la chiave per rispondere a tali necessità. Tra le possibili piattaforme eterogenee, quelle basate su dispositivi riconfigurabili come FPGA (Field Programmable Gate Arrays) mostrano una elevata adattabilità a una varietà di carichi di lavoro, che è un obiettivo importante per l'edge computing. Pertanto, la loro implementazione in diverse applicazioni IoT, che vanno dalla videosorveglianza alla guida autonoma, sta emergendo come una soluzione promettente. Questa tesi di dottorato propone uno studio sulla capacità dei moderni sistemi eterogenei FPGA System-on-Chips (SoC) di implementare nodi sensore intelligenti per applicazioni di visione. A tal fine, numerosi algoritmi di visione artificiale volti a estrarre dati sintetici da frames di ingresso sono stati analizzati e nuove soluzioni orientate ad implementazioni hardware su SoC eterogenei sono state proposte. In tutti i casi presentati, che includono la stereo-visione, l'analisi delle componenti connesse e la visione artificiale, le prestazioni in termini di velocità e/o di efficienza energetica sono state notevolmente migliorate rispetto alle soluzioni già presenti in letteratura. Ad esempio, l'architettura proposta per l'implementazione di reti neurali convoluzionali raggiunge un'efficienza energetica fino all'89,5% superiore rispetto allo stato dell'arte, dimostrando la sua adattabilità nello scenario dell'IoT real-time e low-power.

LIST OF FIGURES

Figure 2.1 Architecture of the generic smart visual IoT node. 16
Figure 2.2 Lifetime vs average power consumptions for different types of battery [26]. 17
Figure 2.3 Design of a generic FPGA-based heterogeneous embedded system. 20
Figure 3.1 A stereo pair of images and their planes of projections. 25
Figure 3.2 Stereo pair (a) captured by unaligned cameras and (b) after the rectification process..26
Figure 3.3 An example on how locating the pixels within regions of interest................29
Figure 3.4 An example of transformation. 30
Figure 3.5 Top-level architecture of the embedded system realized for stereo vision on Zynq-7000 SoCs. 32
Figure 3.6 Structure of the Disparity computation module. 34
Figure 3.7 C-code used in the HLS tool to describe the top-level architecture of Disparity computing block. 35
Figure 3.8 Structure designed for all the buffers within the Disparity computing block. 36
Figure 3.9 (a) Computing WCVR and WCVC vectors. (b) Computing the element AV_{NW}. 37
Figure 3.10 HLS-code for the Window Selection block description. 37
Figure 3.11 Running example for the Windows Selection module when W=5 and Pl=2. 38
Figure 3.12 The HLS-code used to compute the dissimilarity. 39
Figure 3.13 HLS-code of the SelectMin function and its call in the main function.........39
Figure 3.14 Samples of disparity maps: a) ground truths; b) obtained with W_T=W=5; c) obtained with W_T=W=13. 40
Figure 3.15 Timing of the whole process performed by the proposed stereo vision embedded system...42
Figure 3.16 Example of IIM values calculation exploiting the recursive approach and row-parallel computation; grey squares represent locations for which only RS has been calculated, whereas orange-highlighted positions are related to the final IIM values updated in the current step...47
Figure 3.17 (a) Strips-based IIM computing approach adopted in [86]; (b) corresponding input pixels organization within the memory bank...48
Figure 3.18 The proposed IIM computing scheme working on a 4×4 input image; grey squares represent locations for which only RS has been calculated, whereas orange-highlighted positions are related to the final IIM values updated in the current step. 49
Figure 3.19 Schematization of a DSP operating as SIMD dual-mode accumulator. 50
Figure 3.20 Architecture designed to implement the novel IIM computing scheme. 51
Figure 3.21 Adapting the SIMD dual-mode DSP to the proposed IIM computing scheme. (a) Analysis of the data dependency; (b) packing of the inputs......................................52
Figure 3.22 Embedded system accommodating the proposed IIM computing accelerator. 52
Figure 3.23 Stride-mode strategy used to access the external memory for row-parallel IIM computing when m=4 and n=8. 53
Figure 4.1 An example of detecting multiple objects within a real scene. 57
Figure 4.2 Pixel connectivity for a four-connected (a) and an eight-connected (b) set....58
Figure 4.3 Example of labeling process for (a) a new label assignment; (b) an inherited label; and (c) a collision. 58
Figure 4.4 Example of critical path containing a chain of collisions with length 3. 61

Figure 4.5 Example of a stale label. 61
Figure 4.6 Heterogeneous embedded system including a conventional CCA module.....63
Figure 4.7 Heterogeneous embedded system including the proposed CCA module. 64
Figure 4.8 Design of the Provisional Labeling circuit proposed in this work. 65
Figure 4.9 Architecture of the 2D binary table with updating capability. 67
Figure 4.10 Circuit designed for extracting coordinates of bounding boxes for each connected component within the input image. 68
Figure 4.11 Samples of benchmarks images from the GSTRB [115] database. 71
Figure 4.12 Pseudo-code of the proposed one-scan CCA algorithm [21]. 73
Figure 4.13 (a) The input binary image labeled by the novel algorithm. Status evolution of the Translator LUT and Feature Table (b–f). 75
Figure 4.14 Sample images used to extract: (a) area features, (b) bounding boxes, and centroids. 75
Figure 4.15 Top-level architecture of the proposed CCA accelerator [21]. 76
Figure 4.16 The Decision circuit designed for the novel CCA. 77
Figure 4.17 The Extract Feature circuit designed for the novel CCA. 78
Figure 4.18 (a) An example of benchmark image; (b) Histogram of the number of provisional labels assigned by the proposed algorithm to the examined image dataset....79
Figure 4.19 Resources efficiency for different FPGA-based one-scan CCA accelerators. 81
Figure 5.1 Computations performed by (a) a CONV layer and (b) a FC layer................85
Figure 5.2 Top-architecture of the proposed CNN accelerator [10]. 88
Figure 5.3 Data packing strategy to store data into the external DDR memory. 88
Figure 5.4 Data packing strategy adopted for storing the 8-bit convolution kernels (case T_M=8). 89
Figure 5.5 The proposed reconfigurable SIMD Buffer: (a) the architecture; (b) the strategy used to recouple input data; (c) data provided by the registers depending on nr. 91
Figure 5.6 SIMD multiplication with a common operand C and one guard-bit. 92
Figure 5.7 Circuit proposed for the SIMD CE. 93
Figure 5.8 The Accumulate circuit. 94
Figure 5.9 The ReLU & Quantization circuit. 95
Figure 5.10 The Pooling circuit. 96
Figure 5.11 Pseudo-code used to accelerate FC layers and the Softmax function. 97
Figure 5.12 The top-level architecture of the proposed embedded system on Zynq-7000 devices. 99
Figure 5.13 Throughput and bandwidth analysis for different kernel dimensions.........103
Figure 5.14 Schematic diagram of the proposed reconfigurable buffer for non-uniform kernel sizes across CNN layers. 104
Figure 5.15 Design of the novel reconfigurable CE. (a) the top-architecture; (b) details of the ConvMAC block. 105
Figure 5.16 The heterogeneous embedded system accommodating the proposed reconfigurable buffer and CE. 107
Figure 5.17 State-of-the-art comparison for performances, power and resources occupancy. 109

LIST OF TABLES

Table 2.1 Requirements of visual IoT sensors and their impact on hardware specifications. 16
Table 2.2 Comparison of different computing platforms for smart visual IoT nodes. 19
Table 2.3 Smart Camera Systems for edge computing targeting hardware devices.........21
Table 2.4 Some hardware platforms for CNN inference on-the-edge. 22
Table 3.1 Error rates obtained by the novel stereo vison algorithm and other competitors for several test images. 41
Table 3.2 Implementation results for the proposed stereo vision embedded system, and state-of-the-art comparison........44
Table 3.3 Data width required for integral images computation of different sized input frames. 50
Table 3.4 Implementation results of the novel IIM circuit and state-of-the-art comparison. 54
Table 3.5 Speed performances evaluation for a complete embedded system on the XC7Z020 SoC. 55
Table 4.1 Implementation results of the proposed embedded system for one-scan CCA with complete labeling. 70
Table 4.2 Number of clock cycles required by the update step for several GSTRB [118] images. 71
Table 4.3 Characterization of the proposed CCA accelerator [21] and other FPGA-based one-scan CCA designs. 80
Table 4.4 Memory requirement of several CCA architectures processing n×m input images. 81
Table 5.1 Acronyms and parameters used to within the proposed SIMD CNN accelerator [10]. 87
Table 5.2 Characterization of the proposed CNN embedded system compared to most relevant prior works. 101
Table 5.3 Underutilization analysis for different design strategies of the input buffer for *W*=224. 105
Table 5.4 Characterization of the proposed heterogeneous embedded system with reconfigurable computational patterns and comparison with prior works. 108
Table A.1 Parameters and acronyms used to explain the novel stereo vision algorithm. 114

CONTENTS

Abstract 4

Riassunto 5

List of Figures 6

List of Tables 8

Contents 9

1 Introduction 11

1.1 Background and Motivations 11

1.2 Thesis Contributions 12

1.3 Outline of the Thesis 14

2 Smart Visual IoT Nodes 15

2.1 Overview 15

2.1.1 Requirements and Challenges 16

2.1.2 Physical integration of FPGA SoCs in visual IoT nodes 19

2.2 Heterogeneous FPGA Architectures 19

2.3 Hardware Reconfigurable Processing Units for Visual IoT 20

2.4 Summary 23

3 Stereo-Vision on FPGA SoCs 24

3.1 Background 25

3.2 The Novel Algorithm 26

3.2.1 Rectification step 27

3.2.2 Matching cost computation 28

3.2.3 Disparity map refinement 30

3.3 The Embedded System Architecture 31

3.3.1 HLS-based design of the *Disparity computation* block 33

3.4 Results 40

3.4.1 Accuracy measurements 40

3.4.2 Implementation 42

3.5 The novel Integral Image Computing Approach 45

3.5.1 Background 45

3.5.2 The proposed idea and its hardware implementation 48

3.5.3 Experimental results 52

3.6 Summary 55

4 Efficient CCA Approaches for FPGA-based Embedded Systems 57

4.1 Background 57
4.2 Related Works 59
4.2.1 Two-scan CCA methods 59
4.2.2 One-scan CCA methods 60
4.3 The Proposed Embedded System for one-scan CCA with complete Labeling 63
4.3.1 System architecture 64
4.3.2 Custom modules 65
4.4 Results 69
4.4.1 Implementation 69
4.4.2 Experiments for Traffic Sign Recognition applications 70
4.5 The novel Low-cost CCA Implementation 72
4.5.1 Algorithmic-level optimizations 72
4.5.2 Hardware circuit 76
4.5.3 Experimental results for aerospace applications 79
4.6 Summary 81

5 Inference of Deep CNNs on Heterogeneous SoCs 83
5.1 Background and Related Works 84
5.2 The proposed SIMD CNN Accelerator 86
5.2.1 Architecture of the reconfigurable SIMD buffer 89
5.2.2 Architecture of the SIMD Convolutional Engine 92
5.2.3 Architecture of the ReLU and Pooling blocks 94
5.2.4 Software acceleration of FC layers and Softmax function 96
5.3 Results 98
5.3.1 Implementation 98
5.3.2 Experiments on the VGG-16 model 100
5.4 Supporting non-uniform Kernel Sizes 101
5.4.1 The proposed reconfigurable design 103
5.4.2 Results 108
5.5 Summary 109

6 Conclusions 111
6.1 Future Perspectives 112

Appendix A 114

Publications 115

References 117

1 INTRODUCTION

1.1 BACKGROUND AND MOTIVATIONS

The growing Internet of Things (IoT) paradigm is strongly influencing several aspects of everyday-life and behavior of potential users. Nowadays, effects of the IoT are visible in both working and domestic fields, where domotics, assisted living, e-health, enhanced learning are only a few examples of possible application scenarios [1]. Similarly, from the perspective of business users, there exist important consequences in fields like automation and industrial manufacturing, logistics, intelligent transportation of people and goods.

The IoT describes several technologies and research disciplines that enable the Internet to reach out into the real world of physical objects [2]. The basic idea of IoT is the pervasive presence around us of a variety of objects, such as Radio-Frequency IDentification (RFID), sensors, actuators, mobile phones, etc., which are able to interact with each other and cooperate with their neighbors to reach common objectives. It has been estimated that by 2020 the IoT will have an economic impact up to a trillion of dollars, with more than 50 billion low-power devices producing petabytes of data [3]. Even considering the fast-growing size of the server infrastructures, the cloud is expected to fail to manage such an amount of data. Consequently, many research works focus their attention on the decentralization of the workload, enabling that is referred to as edge computing. In the edge computing paradigm, IoT nodes not only sense and collect data, but they also are made able to locally perform some elaborations, avoiding accesses to the cloud that considerably impact on power consumptions and communication times.

However, turning IoT nodes from simple sensors into more powerful and smart embedded systems still represents a challenge from a hardware perspective. Existing solutions are simply a miniature version of cloud servers, based on a Central Processing Unit (CPU) with tightly coupled co-processors (e.g. Graphic Processing Units, GPU) [4]. Unfortunately, CPUs and GPUs are power hungry and have limited energy efficiency [5]. Conversely, an IoT node is expected to be autonomous in terms of functionality and energy, using batteries or energy harvesters as source power. Therefore, targeting a power-optimized design at the device level becomes essential in this scenario [6].

Among existing sensor technologies in the IoT field, cameras are attractive since they are widely used in multiple contexts, such as smart cities, environmental monitoring and security [7]. Despite the great interest, the majority of the existing smart camera embedded systems consist of power-hungry components, including high-resolution imagers [8] and/or

high-end platforms as processing units [9]. Conversely, to meet the energy requirement discussed above, resource-constrained devices have to be taken into account during the design process of IoT nodes [10]. However, low-end processing units typically have limited computational power, thus representing a bottleneck for computationally demanding video processing implementations, especially when real-time is a desirable goal.

Field Programmable Gate Arrays (FPGAs) have been recently recognized as one of the most attractive candidate to realize customizable hardware processing units in smart visual IoT nodes [10]-[12], [30]-[31]. Indeed, FPGAs offer higher flexibility and shorter development times compared to the pure Application Specific Integrated Circuits (ASICs) counterpart, providing acceptable performances and power consumptions [13]. However, designing with FPGAs while taking into account concurrently power, performances, size and cost is not a trivial task, especially for the wide variety of applications, and corresponding requirements, involved in the visual IoT field.

1.2 THESIS CONTRIBUTIONS

This work investigates the suitability of modern heterogeneous FPGAs to implement low-power and high-performance accelerators to be accommodated within resource-constrained smart visual IoT nodes. To this purpose, we firstly introduce the concepts of **HW-SW co-design and embedded systems for video processing** [14]. The systems referenced along this work are based on heterogeneous platforms embedding a general-purpose processor and a FPGA fabric section. The synergy between the aforementioned processing units allows merging the flexibility offered by Operating Systems and by software routines, used to control peripheral interfaces and interconnection resources, with the capability of structuring highly parallel specialized architectures. The latter can be successfully used to implement computationally critical subsystems by reducing latency and energy dissipation.

Low-power and real-time accelerators for stereo vision [15][16] **and connected component analysis** [19][20][21] have been properly designed to be embedded within resource-constrained IoT nodes based on heterogeneous FPGAs. Both stereo vision and connected component analysis are crucial tasks in applications such as advanced autonomous driver-assistance and video surveillance, where objects identification and tracking are required.

Regarding the Connected Component Analysis (CCA) problem, we introduced both architectural [19][20] and algorithmic [21] optimizations to minimize power consumptions and maximize performances with respect to prior works. Results obtained by the Zynq-

7000 XC7Z020 chip demonstrate effective advantages of the novel CCA strategy [21], which allows an efficiency up to six times higher than the most competitive state-of-the-art implementations.

Concerning the stereo-vision task, we designed a complete embedded system making conjunctly use of high-level synthesis (HLS) and hardware description language (HDL) approaches. As usual with applications oriented to energy efficiency and high performances, several optimizations have been introduced at an algorithmic level. Results demonstrate that, in the most of the cases, the novel stereo vision algorithm achieves an accuracy higher than prior works, performing faster than several state-of-the-art designs. Specifically, the fastest prototype, realized on a Zynq-7000 XC7Z045 chip, exhibits a frame rate up to 101 fps, making use of more than 182000 LUTs and 143000 FFs. Conversely, the cheapest implementation, characterized for low-power and resource-constrained environments on a Zynq-7000 XC7Z020 chip, shows an area occupancy up to ×3 lower than the other prototypes, maintaining a good frame rate of 81 fps. As additional contribution, **this work presents a novel strategy for parallel computing integral images on FPGAs** [17]. Integral image is a very powerful way to speed-up algorithms based on specific features extraction, such as the face detection [85], as well as the stereo matching process [18], [94]-[96]. Therefore, the design of hardware architectures able to accelerate the integral image computation receives a great deal of attention. When the proposed computing scheme is implemented within a Zynq-7000 XC7Z020 SoC, interesting results are obtained for both power and speed, which suggest that the novel accelerator is suitable to be integrated within a more composite system like that for stereo vision applications.

Nowadays, convolutional neural networks are widespread used in the most disparate areas of the IoT [22], because of the higher robustness and accuracy compared to traditional computer vision algorithms. To further enhance the smart capability of modern visual embedded systems, **this work also proposes novel hardware architectures for accelerating deep Convolutional Neural Networks (CNNs)** [10][22]. Despite the rapid development of CNNs, the gap between software and hardware implementations is already considerable [24]. In order to make state-of-the-art networks useful at IoT end-nodes, more attention must be paid to limit their power consumption and memory demands. Based on such considerations, in this research, reduced precision CNNs trained models are taken into account, in order to significantly reduce the bandwidth overhead and enable novel Single-Instruction-Multiple-Data (SIMD) computing units to achieve real-time performances even for very deep CNN models. When the proposed SIMD accelerator is implemented within a complete heterogeneous embedded system realized on the Zynq-7000 XC7Z045 device,

the measured power efficiency, defined as the number of operations at second per Watt, is about 135; this corresponds to an up to 89.5% improvement of the state-of-the-art.

1.3 OUTLINE OF THE THESIS

This work is organized as follows.

- Chapter 2 clarifies requirements and challenges in the design of smart visual IoT nodes, providing perspectives for energy, performances and area occupancy. Several hardware-based state-of-the-art implementations are reviewed, with particular focus on heterogeneous System-on-Chips (SoCs) and HW-SW co-design.
- Chapter 3 focuses on the complete embedded system realized for the novel stereo-vision algorithm proposed in [15][16], reporting some details related to the design, as well as accuracy and hardware results obtained testing the architecture by known benchmarks. The novel integral image computing approach [17] is also described in this chapter, and experimental results are discussed and compared with state-of-the-art parallel implementations.
- Chapter 4 discusses the CCA problem, presenting the solutions implemented in [19][20][21] to make this task suitable for smart IoT sensor nodes. Experimental results are presented for two reference applications: traffic sign recognition and aerospace navigation.
- Chapter 5 introduces deep CNNs, with a special focus on the main issues related to hardware implementations. An efficient SIMD approach [10][22] is presented to double the number of operations performed by the same computing unit, providing an actual advantage in terms of power consumption and speed performances compared to traditional designs. A novel runtime reconfigurable circuit is also proposed to accelerate the inference of CNN models having a non-uniform kernel size across layers.
- Chapter 6 summarizes the findings of the thesis work and provides some perspective directions in the field of energy-efficient smart visual embedded systems.

2 SMART VISUAL IOT NODES

2.1 OVERVIEW

Today, Visual IoT nodes are embedded within a wide range of systems to enable several applications, including surveillance, environmental monitoring and advanced driver-assistance. Compared to other mono-dimensional data sensors, cameras produce a richer information content, thus leading to the proliferation of a large amount of data and possible network congestions. In addition, other issues arise when considering privacy aspects of exchanging visual raw data and the expensive computing resources on the remote servers. Moreover, within the node device, the transmission subsystem is commonly the energy bottleneck of the system: in case of continuous wireless streaming of raw data, the battery discharging may occur in a very short time [25]. An efficient solution to address the aforementioned issues consists of bringing intelligence close to the sensor. In accordance to the edge computing paradigm, sensed data is locally analyzed in-the-node by means of the embedded computing capabilities. Locally filtering data reduces its dimensionality and decreases the transmission bandwidth with respect to streaming out all the raw data. This also leads to a noticeable reduction of the transmission energy cost, because the communication subsystems would be in use only for the time required to transmit the high-level information extracted from the visual signal.

A smart visual IoT node combines in a compact embedded platform the image sensing and the local processing unit. Optionally, an external memory can be included at the board-level to deal with the memory requirement of sensing and processing tasks. A block diagram of the typical hardware architecture is depicted in Figure 2.1. The image sensor digitizes the visual signal before dispatching data to the processing unit. When dealing with continuous monitoring, the image source is kept always-on, therefore the power cost for producing and digitally converting the data needs to be extremely contained. Moreover, the sensor-to-processor data transfer, whose bandwidth linearly increases with the amount of produced data, determines an additional energy consumption, impacting the total budget. Any compression scheme aiming to reduce the bandwidth can lead to potential energy savings. The digital processor is the core of the systems and it is responsible for running computer vision algorithms on the visual data. Typically, some low-level features are extracted from the visual signal to be then aggregated into a high-level information through classification or regression models. To this aim, a processing platform consists of a heterogeneous set of digital engines, which may include general-purpose processors

coupled with specialized HW accelerators, a memory subsystem and several peripherals that interface the processing unit with external sensors.

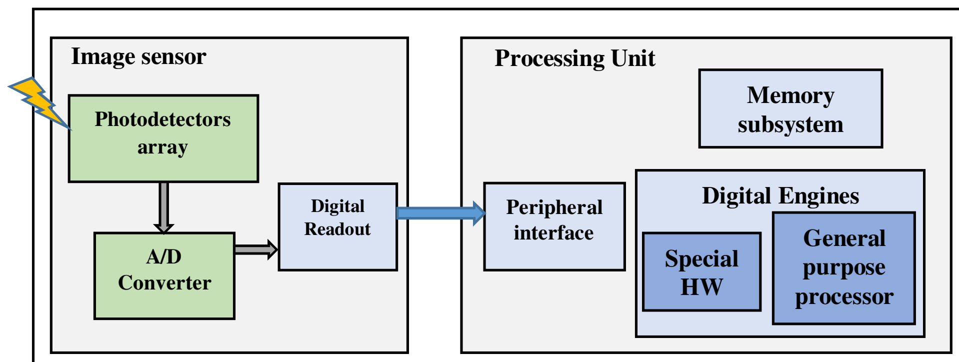


Figure 2.1 Architecture of the generic smart visual IoT node.

2.1.1 Requirements and Challenges

The distinctive features of IoT nodes strictly depend on the requirements of a specific application in terms of physical constraints, type of interaction with the external world, required capabilities and user specifications [26]. Table 2.1 summarizes application requirements of visual IoT nodes and their impact on hardware components and features of a typical smart camera sensor [27].

Table 2.1 Requirements of visual IoT sensors and their impact on hardware specifications.

	Camera	Processing unit	Communication	Power	Embedded encoders
Execution time	✓	✓	✓	-	✓
Energy	✓	✓	✓	✓	✓
Data saving	-	-	✓	✓	-
Data quality	✓	-	✓	-	✓
Size	✓	✓	-	✓	-
Robustness	-	✓	✓	-	✓
Programmability	-	✓	-	-	-
Interfaces	✓	✓	✓	✓	-
Cost	✓	✓	-	-	-

Physical constraints of IoT are dictated by size considerations and the necessity to avoid as much as possible any maintenance intervention, e.g. battery replacement. Regarding the form factor, IoT nodes need to be sufficiently small to make the deployment of IoT nodes non-intrusive, with a typical volume ranging from cubic millimeters to hundreds of mm^3. The weight plays an important role and it strictly depends on the application. It should be < 2kg for a sensor device integrated within an intelligent vehicle, but more severe constraints are required, for example, for a smart watch (in such a case the

weight must be minor than 40g). Moreover, IoT nodes need to be energy autonomous and rely on a battery and/or an energy harvester source. In purely battery-powered nodes, power consumed on average by the IoT node, i.e. P_{avg}, should be small enough to achieve the desired lifetime $t_{lifetime} = E_{battery}/P_{avg}$, for a given battery energy capacity $E_{battery}$. Typical energy capacities vary from a few Watts per hour (Wh), required by ground moving devices such as smart- watches and phones, to a hundred Wh needed by low-altitude moving devices like the unmanned aerial vehicles (UAVs). The interaction of IoT nodes with the external world and the environment they are embedded in also influences the lifetime, as battery replacement is not an option due to the large number or the inaccessibility of nodes. Indeed, when IoT nodes are deployed in buildings or other living environments and infrastructures, a lifetime of several decades can be achieved, whereas industrial applications, transportation and shipping might require a shorter lifetime. Figure 2.2 plots the battery lifetime as a function of the average power consumption, assuming optimistically that the battery self-leakage and ageing are negligible. From this figure, smartphone or button cell batteries assure a reasonably long lifetime of a decade (or more) for P_{avg} in the order of μW. Moreover, the plot suggests that a larger battery may lead to increased lifetimes, but at the cost of larger dimensions and costs.

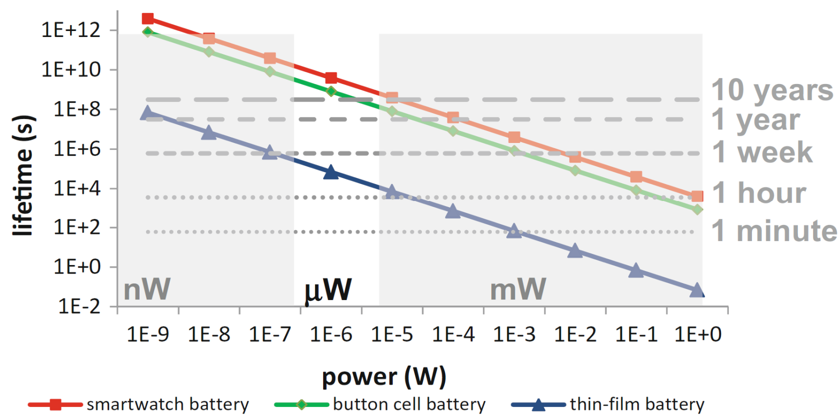


Figure 2.2 Lifetime vs average power consumptions for different types of battery [26].

For many applications, such as video-surveillance, sensed data has to be locally stored to be processed, which introduces the necessity of efficient and cheap local storage units. Furthermore, resolution, brightness, colour space and contrast of the acquired visual data should depend on the specific application requirements. As an example, in the context of automated lane line tracking, converting the input frame to the Hue Lightness Saturation (HLS) colour space ensures higher detection rates than the traditional RGB. For this reason,

recent sensor nodes also feature embedded algorithms to provide encoded/compressed visual data to the processing unit.

As edge computing introduces in-node processing, a smart IoT device targeting computer vision applications shall also guarantee a high computational power, for running data processing tasks in real-time. Moreover, it should be programmable to favour system flexibility and to reduce the development time of a given product. To enable local data processing, multiple kinds of digital processing units can be included within a smart visual system. Among the software-programmable platforms, Graphic Processing Units (GPUs) provide a huge computational power by exploiting a highly parallel architecture. However, this comes at the cost of an increased size and a high power consumption, within a range of tens of Watts, which is too high in the perspective of being powered by batteries. A largely diffused solution for smart camera systems leverages mid-to-high end Central Processing Unit (CPU) architectures, e.g. 32-bit and 64-bit RISC ARM Cortex-A processors [28]. The software-programmability and the high-clock frequencies (hundreds of MHz) achievable with these platforms enable the implementation of many computer vision algorithms running with low latencies. But still, the power consumption does not satisfy the IoT requirements. MicroController Units (MCUs) have become largely widespread as low-power embedded processors [29]. Despite the low cost, MCUs provide limited computational power (the clock frequency is typically up to few tens of MHz) and memory resources (typically up to 128kB), which may not be sufficient to sustain the requirement of some video processing algorithms. Therefore, the design and implementation of computer vision tasks on these devices results still challenging and need to be highly optimized.

Certainly, application-specific integrated circuits show an energy-efficiency much higher than software-programmable digital processors but lack flexibility that, instead, can be essential to run different types of applications on a given platform [32]. Furthermore, other important requirements of IoT nodes come from the user, and are mainly related to the cost. Consumer applications dictate a target of approximately 1 $/node [26]. Such issues, combined to complex and long design flows, hinder the adoption of pure ASIC technologies [32].

The above challenges need to be addressed through platform-based design and moderate reconfigurability to reduce the design costs and expand the range of targetable applications, especially in consumer electronics [26]. In this perspective, heterogeneous SoCs based on FPGA [33]-[34] and standard-cell ASIC [35]-[36] technologies well trade off power, performance, size and cost discussed so far [37].

2.1.2 Physical integration of FPGA SoCs in visual IoT nodes

Table 2.2 summarizes advantages and limitations exhibited by different computing platforms when they are integrated within a visual IoT node. Actually, while ASIC-based SoCs are still preferred in applications that focus on the size of the device (e.g. body wireless nodes for health monitoring), heterogeneous solutions integrating both an FPGA and a hard-core CPU within the same chip are becoming the most popular in other applications, such as autonomous driving [38], industrial IoT [39] and video surveillance [40]. Moreover, it is worth noting that the footprint of modern FPGA SoCs is small enough to permit an easy integration within most of the smart camera sensor based systems typically used in IoT. As an example, the smallest device-packages provided by the Xilinx and Intel FPGA SoCs vendors are, respectively, 13×13×0.8 mm (Zynq XC7Z010) and 8×8×0.5 mm (Cyclone 10), which result to be lower with respect to the hundreds of mm^3 mentioned in the previous subparagraph.

Table 2.2 Comparison of different computing platforms for smart visual IoT nodes.

	Performance	Flexibility	Power	Size	Cost	Time-to-market
MCU	Low	Low	Low	Medium	Low	Fast
CPU	Medium	High	Medium	Medium	Medium	Fast
GPU	High	High	High	Large	High	Fast
FPGA	High	High	Medium	Medium	Medium	Fast
ASIC SoCs	Very High	High	Low	Small	High	Slow
FPGA SoCs	Very High	High	Low	Medium	Medium	Fast

2.2 HETEROGENEOUS FPGA ARCHITECTURES

Technological advances in the fields of heterogeneous SoC has revolutionized the way electronic systems are designed. The FPGA has evolved from a simple prototyping device to an essential solution for the development of systems that require high processing capacities, real-time operations, interoperability, flexibility, safety, and high availability [39]. Modern FPGAs, integrate also ARM processors implemented in silicon. Thanks to these novel manufacturing techniques, the sensors are getting smaller and can perform more processing, allowing the execution of more complex applications, such as machine learning algorithms and data analysis.

As illustrated in Figure 2.3, the typical FPGA SoC consists of two main portions: **i)** the Processing System (PS), which provides the functionality of a single- or a multi-processor core to run the software design and, if needed, the operating system; **ii)** the Programmable Logic (PL), in which the specialized hardware design can be accommodated. The PS and the PL, in addition to exchanging data with each other, can access external memory and auxiliary peripheral resources through appropriate interfaces.

In such embedded systems, the advanced extensible interface (AXI), which is a part of the advanced microcontroller bus architecture (AMBA), is typically adopted since it achieves high performance and can connect hundreds of masters and slaves within SoCs. The fourth generation of this protocol [80] provides three types of interfaces:

- the AXI4 that is used to control traditional memory-mapped transactions and supports high throughput bursts of up to 256 data transfer cycles with just a single address phase;
- the AXI4-Lite that supports single transaction memory-mapped communications;
- the AXI4-Stream that does not require the address phase, allows unlimited data burst size and therefore it is suitable for high-speed streaming data.

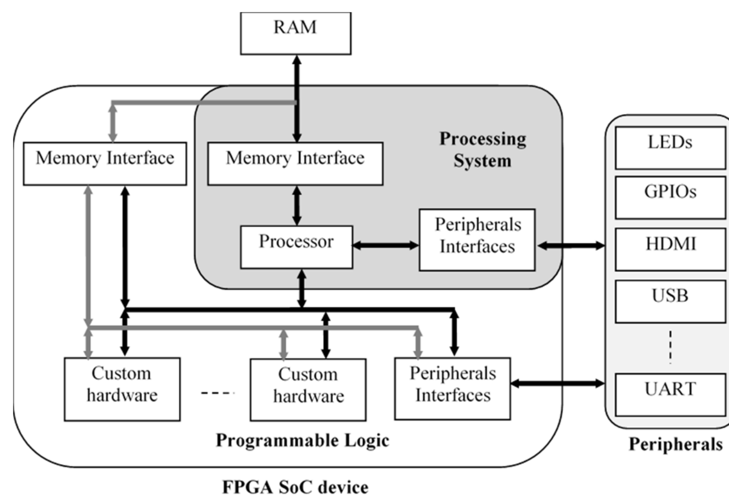


Figure 2.3 Design of a generic FPGA-based heterogeneous embedded system.

2.3 HARDWARE RECONFIGURABLE PROCESSING UNITS FOR VISUAL IOT

In the last few years, significant effort has been put into the development of efficient smart camera systems targeting FPGAs [15], [40]-[46]. A survey is reported in Table 2.3. Typically, local processing is performed in order to reduce the acquired image to a few synthetic data useful for the target application, by feature extraction algorithms, thus avoiding communication bottlenecks [40]-[42]. As an example, the micro power smart camera system presented in [41] exploits a dynamic background subtraction method to detect unusual event, as commonly occurs in video surveillance applications. The system adaptively updates its sensitivity and thresholds in order to efficiently filter out noise and light changes. Then, the output of this process is segmented and connected components [19], corresponding to the cluster of pixels identified as abnormal in the scene, are analysed to extract the features of interest. It is worth noting that such a dataflow enables some

portions of the processing unit to operate in sleep mode for most of the time, leading to a considerable reduction of the power consumptions.

As an alternative solution to avoid sending raw-images captured by the camera sensor, compression techniques can be adopted [46]. In this case an image coder based on a transformation stage, such as the discrete wavelet transform (DWT), is necessary. Unfortunately, implementing conventional 2-D DWT on hardware would require the whole raw-image to be available in the memory subsystem. This would make the implementation on resource-constrained devices unfeasible for high resolution images. Therefore, an optimized technique is applied in [46] to drastically reduce the memory requirements, obtaining power consumptions of a few ten of milliWatts for DWT process applied on 512×512 frame resolutions.

Table 2.3 Smart Camera Systems for edge computing targeting hardware devices.

	Imager	Processor	Data processing	Power consumption
[40]	SCAMP (128×128)	IGLOO FPGA + NXP LPC1769 MCU	Object detection and counting - Smart visual Trigger	5.5 mW
[41]	Custom sensor [56] (104×104)	Xilinx Spartan 6 FPGA	Background Subtraction, segmentation and CCA	80 μW*
[42]	KOVA1 [58] (96×96)	Focal plane processor ASIC + Xilinx FPGA Spartan 6	Segmentation and Hough transform	<1 W**
[43]	Custom sensor (32×32)	Xilinx Kintex-7 FPGA	Binary Neural Networks	0.52 W
[45]	OmniVi,OV7670 [57] (640×480)	Xilinx XC7Z020 FPGA SoC	Visual saliency detection	0.36 W
[46]	Not included	Xilinx Spartan 3 FPGA	Fractional Wavelet filtering	0.041W

*Referred just to the camera sensor. **Estimated.

More frequently, smart cameras are the building blocks of larger systems, developed for high-performance and energy-constrained applications, and targeted for compact and low cost heterogeneous CPU-FPGA SoCs [45]. The ability of this kind of embedded platforms has been already demonstrated also in the context of advanced intelligent transportation systems [44]. Results that are even more impressive have been recently obtained in the deep learning field [47]. There, FPGA allows to implement irregular parallelism, customized data type and application-specific hardware architecture, offering great flexibility to accommodate the recent deep learning models that are featured with increased sparsity and compact network structures [48], [50]. In addition, the heterogeneity of CPU-FPGA SoCs promotes the implementation of complex cognitive tasks, like those involved in CNN algorithms. Indeed, in such cases, traditional FPGAs may be inefficient, and a well-organized co-design has to be adopted to partition operations between hardware

and software computing, even sharing the available space address between CPU and FPGA [49]. This allows running selected portions of an algorithm on FPGA, thus maximizing the efficiency of the specific target application, e.g. in terms of energy, leaving the CPU to perform the remaining non-critical operations.

Unfortunately, running CNN models requires high computing and memory resources, thus making deep learning on edge quite challenging [50]. Energy and performance efficiencies are the main metrics when designing accelerators for edge computing. Boosting such efficiencies depends on both algorithmic-level optimizations and hardware improvements. Table 2.4 reports the main features of some state-of-the-art hardware platforms for in-node CNN inference. It is worth noting that, although FPGA-based SoCs represents the most common solution to realize such a task, offering short development times and high flexibility, there exist HW/SW approaches that exploit ASIC technologies [35]-[36]. In such a case, a custom hardware accelerator integrated in silicon can communicate with an external CPU through proper interfaces [35], or it can be coupled within the same chip with one or more simile-CPU cores that include a reduced instruction set computer [36].

Table 2.4 Some hardware platforms for CNN inference on-the-edge.

	[35]	[36]	[51]	[52]	[53]	[54]
Year	2012	2018	2014	2018	2018	2017
Technology	45 nm*	65 nm*	28 nm	28 nm	28 nm	28 nm
Operating frequency (MHz)	400	400	142	100	125	150
GOPs	294	30	23.1	84	48.5	36.8
GOPs/Watt	490	83.5	25	24.1	27.7	79.7

*ASIC technology.

Neural Network-Next (nn-X) [51] is a SoC platform, based on the Xilinx Zynq-7000 XC7Z045 device, for deep learning targeting mobile devices. The top-architecture includes a processing element module, implemented on FPGA, which is responsible for running all arithmetic operations required by CNNs, like multiply-and-accumulations (MACs). The CPU is used to control data transfers, runtime configure the programmable logic and compile the co-processor for different CNN models. At a running frequency of 666MHz and 142MHz for CPU and FPGA respectively, the platform dissipates 4 Watts, leading to a performance-energy efficiency (GOPs/Watt) of more than 25, which resulted enough high for mobile processors.

Angel-Eye [52] is a design flow to map CNNs onto low and medium density FPGAs. The hardware architecture allows runtime configuration to run different CNN models

through a controller that is properly instructed by the processor. When implemented within the Xilinx Zynq-7000 XC7Z020 SoC, Angel-Eye achieves a peak performance of 84 GOPs at a 100MHz running frequency, delivering an efficiency of 24.1 GOPs/W.

The *fpgaConvNet* framework presented in [53] generates an architecture to run different CNNs starting from a high-level description of the model. Specifically, this framework partitions a graph representation of the network and produces several bitstreams, referred to each part of the graph, thus allowing a dynamically reconfiguration of the FPGA. *fpgaConvNet* has been deployed on a Zynq-7000 XC7Z020 chip, achieving a throughput up to 48.5 GOPs, corresponding to an efficiency of 27.7 GOPs/W.

A CNN-specific Instruction Set Architecture (ISA), which embeds the parallel computation and data reuse parameters in the instructions, is presented in [54]. An instruction generator deploys the instruction parameters according to the feature of CNNs and hardware computation and storage resources. In this way, power consumption is significant improved, but at the expense of a non-negligible deterioration in performances, which is not tolerable in several real-time applications [20][41][44].

2.4 SUMMARY

This chapter introduced the main challenges in the design of smart visual IoT nodes. Based on various requirements, which are application-dependent, designers must choice the most adequate platform to perform close-to-node processing, thus exploiting the several advantages of edge computing until now discussed. Although software programmable processors lead to lower development times, reduced costs and more flexibility with respect to hardware-based counterparts, several researches have already demonstrated their unsuitability in low-power applications. On the other hand, application specific integrated circuits show high design complexity and costs, as well as long time-to-market and low flexibility. For these reasons, FPGA devices, and in particular modern heterogeneous SoCs, are widely recognized as the most appropriate hardware realization platforms to trade-off performances, power, design times and cost, as required in the context of smart visual IoT.

3 STEREO-VISION ON FPGA SOCS

In the last decade, stereo vision has become crucial for almost all modern computer vision applications, such as autonomous vehicles, robot's navigation, gesture recognition, three-dimensional reconstruction, smart cameras, augmented reality, and many others [59]-[61]. Essentially, stereo vision provides the ability of estimating depth information of an observed scene from 2D images captured by two cameras having different viewpoints. Objects visible to both cameras are projected into the captured images at different positions. These differences, called disparities, allow 3D information about the surrounding world to be inferred through the stereo matching, which is used to search and locate within the acquired images corresponding projections of the same 3D points [62]. Nowadays, Machine Vision (MV) is considered as one of the key technologies for the Internet of Things (IoT) and Industry 4.0 or Industrial IoT. The fusion between stereoscopic techniques and MV systems is highly desired for several industrial and consumer IoT applications. For this reason, designing efficient hardware architectures for stereo vision algorithms is attracting many researchers.

Although conceptually simple, performing the stereo matching easily and accurately is not trivial. Several algorithms have been proposed in the past with the aim of either increasing the accuracy, or reducing the computational complexity, or both [63]-[72]. The comprehensive review of stereo vision algorithms known in literature and provided in [73]-[74], shows that stereo matching algorithms can be mainly classified into local algorithms [64]-[72], also named area-based approaches, and global algorithms, such as those presented in [75]-[76]. The former compute disparities by processing the intensity pixel values within windows around points of interest, whereas the latter recognize matching pixels by minimizing a global cost function (e.g. the disparity at a given pixel is computed considering the disparities at all other pixels). Global algorithms can be much more accurate than local approaches, but, as a drawback, they consume much more time and require much more resources or special platforms to be implemented, thus making low-cost real-time standalone hardware implementations unapproachable. Starting from this observation, in the following the attention is focused only on local area-based approaches. Among them, algorithms oriented to hardware designs [63]-[72] are desirable when the main objective is including stereo vision in portable consumer electronic and multimedia systems, which demand high computational speed, good accuracies, low costs, low memory requirements and low power consumption, without renouncing to a good flexibility.

3.1 BACKGROUND

The stereo-vision process is based on using two cameras that are placed on the same plane at a known distance to capture the same scene from different point of views. By comparing these two images, the relative depth information can be obtained. In order to explain the technique behind the stereo-vision, let us consider Figure 3.1. In such an example, a generic point *P*, referred to an object in the scene, produces two perspective projections that are located at pixels *Pl* and *Pr* on the image planes of left and right cameras, respectively. The pixel *Pl* is shift from the centre, while *Pr* is at the centre of its projection plane. This shift between the corresponding pixels on the left and the right camera images should be computed to estimate the *matching* pixels and to get the depth information of the object. The shifted amount is called *disparity*. The higher is the disparity of an object pixel the closer is this object to the cameras.

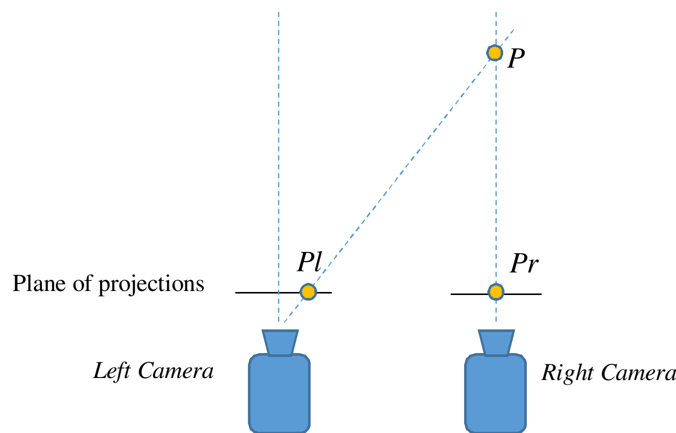


Figure 3.1 A stereo pair of images and their planes of projections.

It is then clear that both *Pl* and *Pr* pixels are identified by their respective (*xl*, *yl*) and (*xr*, *yr*) coordinates in each projections plane. Therefore, for each pixel of the image chosen as *reference*, the matching process needs of a 2D search within the *candidate* image. This is mainly due to the optical distortion of the camera lens. However, to reduce the search complexity, the rectification step can be used as pre-processing to obtain a pair of images where matching pixels are located along the same row. By such an elaboration, the two cameras are virtually made parallel and perfectly aligned, as illustrated in the sample of Figure 3.2.

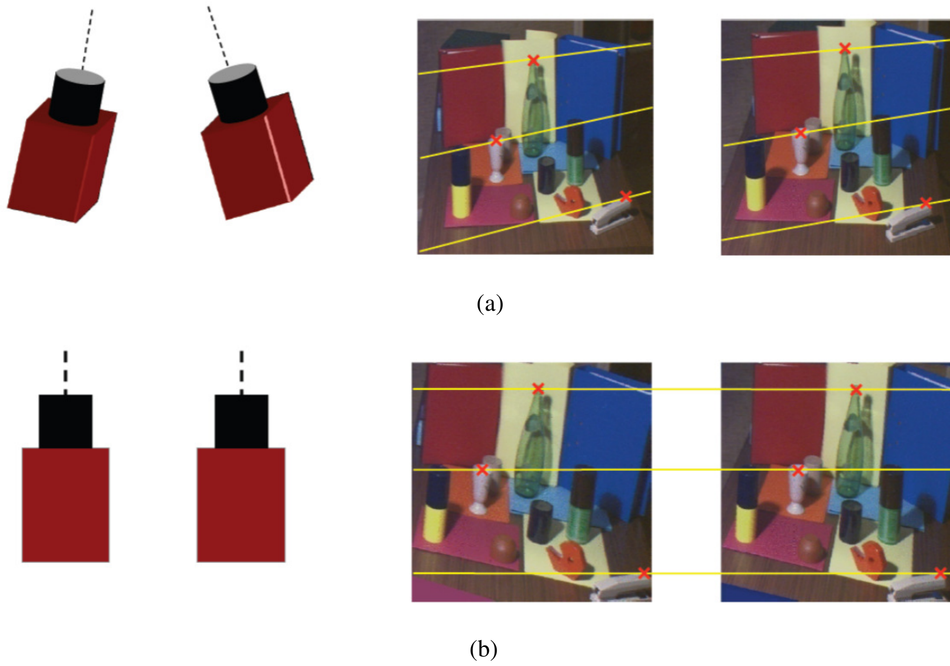


Figure 3.2 Stereo pair (a) captured by unaligned cameras and (b) after the rectification process.

3.2 THE NOVEL ALGORITHM

To facilitate discussions and descriptions of the novel stereo vision algorithm [16], principal parameters and acronyms used in the following are summarized in Table A.1 provided in Appendix A.

Computing disparity maps by the proposed algorithm requires five steps: **i)** the input stereo images are rectified and transformed through a modified version of the non-parametric Adaptive Census Transform (ACT) [64]-[65]; **ii)** one of the transformed images is set as the reference and the other one as the candidate; then, for each pixel $R_{(x,y)}$ in the reference image and for each pixel $C_{(x,y+d)}$ in the candidate image, the matching cost $MC_{(x,y,d)}$ is computed, with d being the disparity that varies between the minimum d_{min} and the maximum d_{max} values; **iii)** the dissimilarity between the reference pixel $R_{(x,y)}$ and each candidate pixel $C_{(x,y+d)}$ is computed by aggregating the matching costs $MC_{(x+r,y+r',d)}$ within an aggregation window of radius s, so that r and r' vary between $-s$ and s; **iv)** the disparity d leading to the minimum dissimilarity value is recognized and associated to the value $\delta_{(x,y)}$ in the disparity map; **v)** finally, validity check and post-processing operations are performed on the disparity map to improve its accuracy.

3.2.1 Rectification step

The rectification is a transformation required to compensate imperfect alignment between the cameras and their geometric distortions. In order to rectify the raw images, the preliminary camera calibration must be performed to obtain intrinsic and extrinsic camera parameters. The former defines the internal geometry of each camera, whereas the latter define the relation between the two cameras. Calibration parameters summarized in Table A.1 were obtained off-line using the Matlab calibration toolbox [76].

Among several rectification techniques, [78] has been chosen since it removes both tangential and radial distortions. To understand how the rectification process works, let consider the generic $n \times m$ raw left image iL. The objective is computing each pixel $R_{(xrect,yrect)}$, with $x_{rect} = 0, \ldots, n-1$ and $y_{rect} = 0, \ldots, m-1$, of the rectified left image R (used as the reference image in the subsequent disparity computation) by identifying its corresponding raw pixel $iL_{(xraw,yraw)}$, with $x_{raw} = 0, \ldots, n\text{-}1$ and $y_{raw} = 0, \ldots, m\text{-}1$. In order to do this, the non-integer raw coordinates (x_{raw}, y_{raw}) are computed by Eq. (1).

$$\begin{bmatrix} xx \\ yy \\ zz \end{bmatrix} = ML \times \begin{bmatrix} x_{rect} \\ y_{rect} \\ 1 \end{bmatrix} \tag{1a}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} xx/zz \\ yy/zz \end{bmatrix} \tag{1b}$$

$$r_2 = x^2 + y^2; r_4 = r_2^2; \tag{1c}$$

$$a_1 = 2 \cdot x \cdot y; a_2 = r_2 + 2 \cdot x^2; a_3 = r_2 + 2 \cdot y^2 \tag{1d}$$

$$\begin{bmatrix} xd \\ yd \end{bmatrix} = \left(1 + kcL_{(1)} \cdot r_2 + kcL_{(2)} \cdot r_4\right) \times \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} kcL_{(3)} \cdot a_1 + kcL_{(4)} \cdot a_2 \\ kcL_{(3)} \cdot a_3 + kcL_{(4)} \cdot a_1 \end{bmatrix} \tag{1e}$$

$$\begin{bmatrix} x_{raw} \\ y_{raw} \\ z \end{bmatrix} = KKL \times \begin{bmatrix} xd \\ yd \\ 1 \end{bmatrix} \tag{1f}$$

The raw coordinates are finally used to compute the rectified pixel value by interpolation. More exactly, *xri* and *xrf* being the integer and the fractional part of x_{raw}, and

yri and yrf being the integer and the fractional part of y_{raw}, four adjacent raw pixels are interpolated as given in (2), with aa_1, aa_2, aa_3 and aa_4 being defined as given in (3).

$$R_{(xrect,yrect)} = aa_1 \cdot iL_{(xri,yri)} + aa_2 \cdot iL_{(xri,yri+1)} + aa_3 \cdot iL_{(xri+1,yri)} + aa_4 \cdot iL_{(xri+1,yri+1)} \quad (2)$$

$$aa_1 = (1 - xrf) \cdot (1 - yrf) \quad (3a)$$

$$aa_2 = (1 - xrf) \cdot yrf \quad (3b)$$

$$aa_3 = xrf \cdot (1 - yrf) \quad (3c)$$

$$aa_4 = xrf \cdot yrf \quad (3d)$$

Equations (1)-(3) are used also to compute the generic rectified pixel of the right image C, which will be used as the candidate image in the disparity computation. Obviously, in this case, the calibration parameters MR, kcR and KKR must be used in place of ML, kcL and KKL. It is worth noting that, in both hardware and software implementations exploited in this paper for the image rectification, equation (1b) was eliminated since the alignment matrices ML and MR of the used stereo camera system led to $zz \approx 1$ for any coordinate (x_{rect}, y_{rect}), thus making $x \approx xx$ and $y \approx yy$. In this way, non-integer division operations are avoided with a significant benefit in terms of computational complexity.

3.2.2 Matching cost computation

A non-parametric matching cost MC is exploited to guarantee less sensitivity to the intensity differences that the acquisition of images from two distinct cameras can cause on actually corresponding pixels. The input images are preliminary transformed: for each reference pixel $R_{(x,y)}$, a $W_T \times W_T$ support window centred at $R_{(x,y)}$ is processed, with $W_T = 2s_T + 1$ and s_T being the radius of the window. As the result, the W_T^2-element weighted census vector $WCVR_{(x,y)}$ is obtained. It has one element for each neighbouring pixel $P_{(x+r,y+r')}$, with r and r' ranging between $-s_T$ and s_T. The pixel intensity difference $\Delta c = |P_{(x+r,y+r')} - R_{(x,y)}|$ is used to compute the k-th element of $WCVR_{(x,y)}$ as given in (4a), with $k = W_T \cdot (s_T - r) + (s_T - r')$ and $sign$ being obtained as shown in (4b). It is worth noting that, in order to exploit the scaled approximation of the exponential support weights originally demonstrated in [64][65], Δc is divided by 4.

$$WCVR_{(x,y)}(k) = 4 \times sign \times \begin{cases} 16 & | \quad \lfloor \Delta c/4 \rfloor = 0 \\ 12 & | \quad \lfloor \Delta c/4 \rfloor = 1 \\ 8 & | \quad 2 \leq \lfloor \Delta c/4 \rfloor < 4 \\ 4 & | \quad 4 \leq \lfloor \Delta c/4 \rfloor < 8 \\ 2 & | \quad 8 \leq \lfloor \Delta c/4 \rfloor < 12 \\ 1 & | \quad 12 \leq \lfloor \Delta c/4 \rfloor < 16 \\ 0 & | \quad \lfloor \Delta c/4 \rfloor \geq 16 \end{cases} \tag{4a}$$

$$sign = \begin{cases} 1 & if\ P_{(x+r,y+r')} - R_{(x,y)} \geq 0 \\ -1 & otherwise \end{cases} \tag{4b}$$

In order to further reduce the sensitivity of the *MC* to the intensity differences and to characterize the relationships between the pixels within the $W_T \times W_T$ window as better as possible, the transformation step is performed in an innovative way. The support window is divided into six regions of interest around the central pixel, namely the northwester (NW), northeaster (NE), southwester (SW), southeaster (SE), central column (CC) and central row (CR) regions. Within each region, the number of pixels having an intensity value greater than the central one is counted and accommodated within the 6-element additional vector *AVR*. Recognizing the pixels of each region through their row and column indices is simple. Figure 3.3 shows an example on how locating the pixels of each region of interest in the case of 9×9 support window. It is then clear that, after the transformation, for each reference pixel $R_{(x,y)}$, a total number of W_T^2+6 elements are computed, which are more than the W_T^2 elements computed by applying the simple ACT as demonstrated in [64], but much less than the $(4 \cdot W_T - 3) \cdot W_T^2$ elements calculated exploiting the approach proposed in [65].

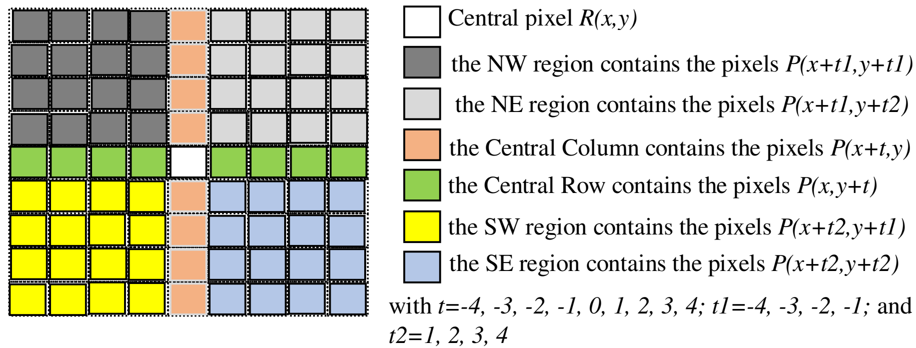


Figure 3.3 An example on how locating the pixels within regions of interest.

The same computations are performed also for each pixel $C_{(x,y+d)}$ within the candidate image to compute the W_T^2-element vector $WCVC_{(x,y+d)}$ and the 6-element additional vector $AVC_{(x,y+d)}$. To better explain how the transformation works, let consider the example

reported in Figure 3.4. The 5×5 support window *WinR* given as input is centred at the pixel $R_{(x,y)}=19$. The absolute values of the pixel intensity differences, Δc, are computed by subtracting 19 from each neighbour pixel. The sign of each subtraction is also obtained. As the next step, equation (4) is applied to each value of $\left\lfloor \frac{\Delta c}{4} \right\rfloor$ to compute the 25-element vector $WCVR_{(x,y)}$ also shown in Figure 3.4. By counting the number of pixels greater than 19 within the regions NW, NE, CC, RC, SW and SE, the 6-element vector $AVR_{(x,y)}$={2, 2, 3, 0, 2, 2} is obtained. The matching cost of the reference and candidate pixels $R_{(x,y)}$ and $C_{(x,y+d)}$ is finally computed as shown in equation (5). Several matching cost metrics and functions, such as the sum of absolute differences (SAD), the sum of squared differences (SSD), the normalized cross correlations (NCC), and many others, can be exploited to this aim. In this research the SAD was used, since it is the simplest matching cost function. Obviously, more complex approaches can also be exploited. However, any potential benefit would be paid in terms of computational complexity.

$$MC_{(x,y,d)} = \sum_{k=0}^{W_T^2-1} \left| WCVR_{(x,y)}(k) - WCVC_{(x,y+d)}(k) \right| + \sum_{v=0}^{5} \left| AVR_{(x,y)}(v) - AVC_{(x,y+d)}(v) \right| \quad (5)$$

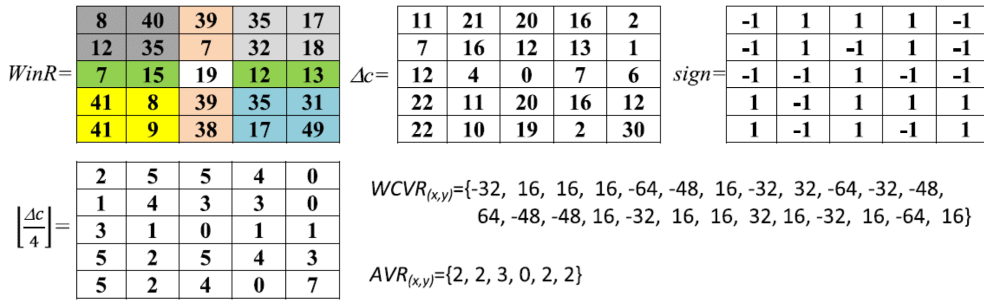


Figure 3.4 An example of transformation.

3.2.3 Disparity map refinement

The disparity map computed as above explained is typically noisy, and performing appropriate post-processing computations can improve the overall accuracy. The validation check is typically used to recognize erroneous disparity values [78][79]. Among several existing approaches, the asymmetric uniqueness validity check demonstrated in [78] is adopted in the proposed algorithm since it furnishes a good compromise between accuracy and complexity. This method simply discards the disparity values obtained for reference pixels that match the same candidate pixel. Successively, to remove the holes introduced by the validation step and to smooth the resulting disparity map, the filling and the 5×5

median filtering are performed. It is just worth noting that the generic hole at the position *(xr,yr)* within the checked disparity map is filled with the maximum disparity between the previous and next valid values in the same row *xr* that have the column indices closest to *yr*.

3.3 THE EMBEDDED SYSTEM ARCHITECTURE

The top-level architecture of the proposed embedded system, designed for the Xilinx Zynq-7000 SoCs family [34], is depicted in Figure 3.5. Stereo images are captured by the stereo camera system consisting of two OV7670 Omnivision CMOS cameras [57] connected to the PL I/Os of the Zynq device through the PMODs connectors on-board available. The hardware portion implemented in the PL of the FPGA-based SOC includes the VGA_PLL, the *Camera Controller* and the *Capture* modules. The latter have been designed by VHDL coding. The VGA_PLL receives the external clock *clk* and generates the 50MHz camera clock signal *cclk*, in accordance to the selected 640×480 input video resolution. The *Camera Controller* appropriately configures the cameras to set the chosen YCbCr 4:2:2 data format [57]. The *Disparity Computation* module implements the proposed stereo vision algorithm; it processes 8-bit greyscale images that are easily obtained by extracting from the incoming colour pixels the luma component Y and discarding the chroma components Cb and Cr, taking into account that, in the YCbCr 4:2:2 format, the Cb and Cr channels are shared between two consecutive pixels. Such an action is performed by the *Capture* modules that produce data streams then inputted through AXI4-Stream interfaces to the AXI Video Direct Memory Access (VDMA) IP cores, which provide high bandwidth accesses to the external Double Data Rate (DDR3) memory on-board available. The VDMAs 0 and 1 firstly store the captured frames into the DDR memory; they directly access the memory through memory-mapped AXI4 interfaces supported by the High Performance slave ports S_HP_0 and S_HP_1 of the PS. To accomplish these tasks, the VDMAs receive configuration data, necessary to specify the address spaces, as well as the lengths of the input and output streams, from the PS through the master General Port M_GP_0, which supports the AXI4-Lite interface protocol.

Frames stored in the DDR memory are rectified by an appropriate C++ software routine running on the PS. Then, VDMAs 0 and 1 are configured to read the rectified images *R* and *C* from the DDR memory and to transfer them through AXI4-Stream interfaces to the *Disparity Computation* custom module. The resulting stream δ is then transferred to the DDR memory by means of the $VDMA_2$ through the S_HP_2 port of the PS. Finally, the validity checks and the filling steps are executed by the PS. Such refined

disparity map $r\delta$ is then read by the AXI VDMA$_2$ to proceed with its displaying. In order to do this, the High Definition Multimedia Interface (HDMI) and the HDMI Transmitter ADV7511 on-board available are used. The AXI Inter Integrated Circuit (I^2C) controller IP core receives configuration data from the PS through the M_GP$_0$ port and generates the protocol signals required to drive the ADV7511 transmitter device with a serial clock frequency of 100kHz. The auxiliary IP cores Video Timing Controller (VTC), AXI4S to Video Out, and HDMI Output are also used to properly accommodate video data transferred to the ADV7511 device, which finally sends the refined disparity map to the HDMI monitor through the HDMI Out connector on-board available.

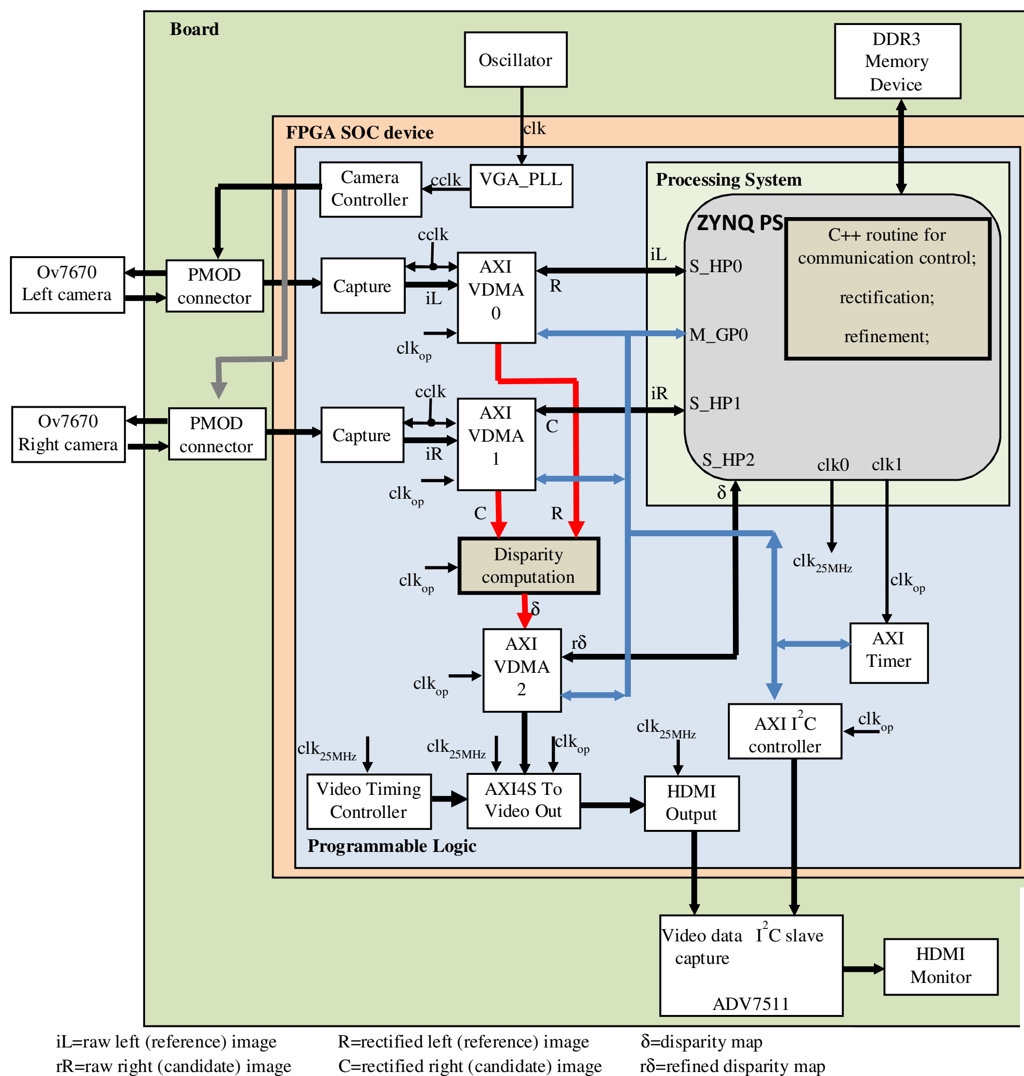


Figure 3.5 Top-level architecture of the embedded system realized for stereo vision on Zynq-7000 SoCs.

From Figure 3.5 it can be seen that the proposed embedded system uses different clock domains. They are associated to the following signals: the 666MHz processor clock; the 533MHz DDR clock; the 50MHz clock signal *cclk*, used to capture the input stereo images;

the 25MHZ clock signal clk_{25MHZ} generated by the PS and used to interface the PL and the HDMI peripheral; the clk_{op} clock signal depending on the speed achieved by the custom accelerator and used also to synchronize the AXI4-Lite, the AXI4 and the AXI4-Stream data transfers. In the aforementioned architecture, both rectification and refinement processes are software implemented. Certainly, such a design choice doesn't lead to the highest achievable speed performances, but it significantly reduces hardware complexity. However, in the next sections alternative HW-SW partitioning strategies will be discussed and characterized.

3.3.1 HLS-based design of the *Disparity computation* block

The design of the module responsible for computing disparity maps within the embedded system of Figure 3.5 has been made through the Xilinx High-Level-Synthesis tool of Vivado [81]. This allows to synthetize the custom circuit starting from its C language based description.

The top-level architecture of the novel *Disparity computation* module is illustrated in Figure 3.6. Reference and candidate pixels are parallel transferred to the Left and Right channel respectively in raster order. In both cases, such pixels are inputted to an *Image Buffer* block, which properly accommodate the $W_T \times W_T$ support windows that will be then inputted to the *Transformation* modules for the computation of the W_T^2-element vectors *WCVR* and *WCVC*, and the 6-element vectors *AVR* and *AVC* as described in Section 3.2.2. Vectors generated after the transformation are stored within the *MC buffers*, which prepare the $W \times W$ aggregation windows needed for the subsequent dissimilarities computations. It important underlying that, while the *MC buffer* referred to the left channel provides just one aggregation window, the *MC buffer* designed for the right channel has to make available d_r $W \times W$ aggregation windows at a time. The latter are evaluated to select the *Pl* candidate windows corresponding to the *Pl* disparity levels processed in the current iteration, where $Pl = d_r/h$ is the level of parallelism adopted to analyse the disparity levels within each partition. This allows computing the generic disparity value $\delta_{(x,y)}$ associated to the reference pixel $R_{(x,y)}$ is computed within h iterations.

As visible in Figure 3.6, candidate and reference pixels are also used to fed the respectively *Buffers for weights* within the two channels. These buffers prepare the $W \times W$ windows processed as described in Section 3.2.2 to compute the support weights. Once again, the buffer within the left channel produces one window at a time, while the buffer used by the right channel outputs d_r $W \times W$ windows in parallel, in order to support the *Pl* level of parallelism. The *Compute Weights* modules then calculate the support weights over

the selected $W \times W$ windows. As the next step, the module *Compute Dissimilarities* computes *Pl* dissimilarity values in parallel; the subsequent *SelectMin* module selects the minimum and outputs the associated disparity.

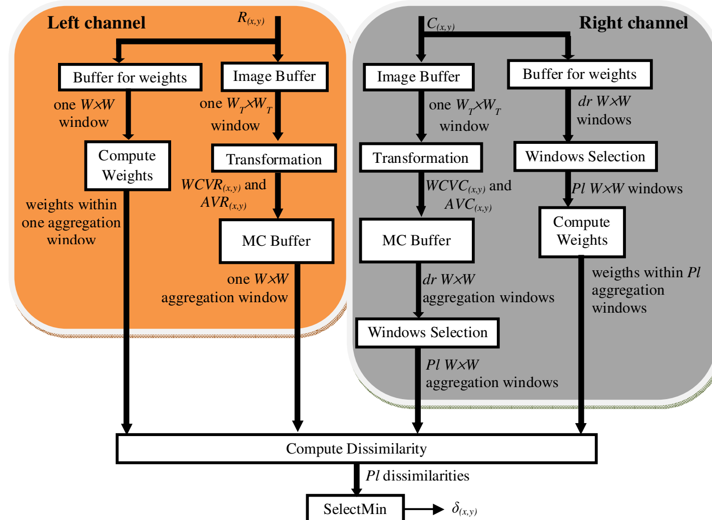


Figure 3.6 Structure of the *Disparity computation* module.

The C-code used to describe the architecture above illustrated is reported in Figure 3.7. The *Disparity computing* block can be seen at this level as a black box named *main*, where only inputs and outputs are defined: the former are **pixel_R** and **pixel_L**, while the latter is **disp_out**. All this information is well-defined in the form of parameters for the *main* function. Input and output are *hls::stream* data type and the #pragma HLS INTERFACE *axis* directives are used to instruct the synthesis tool to provide the implemented architecture with input and output AXI4-Stream interfaces.

The hardware operations above described are defined within the main *for* loop within the code. This loop has a total number of operation that depends on the number of pixels per image ($n_pixels = n \times m$) and the latency of the circuit (e.g. the number of iterations needed to obtain the first $W \times W$ aggregation window of the reference image). The #pragma HLS PIPELINE ensures that the for loop is synthesized in a pipeline fashion. The Initiation Interval (II), set to h, defines the number of clock cycles per loop iteration, e.g. the throughput of the architecture. In other words, h defines the grade of hardware reuse: the same hardware, which is able to process *Pl* disparity levels per clock cycle, is used h times to process d_r disparity levels per each reference pixel.

The structure of the *Image Buffer*, used in both the left and right channels, is depicted in Figure 3.8. It consists of one line shift buffer, storing m-W_T words each $8 \times (W_T\text{-}1)$ bits wide, and a set of $W_T \times W_T$ 8-bit registers, which can be independently accessed. The HLS description of the *Image Buffer* is also reported in Figure 3.8. The line buffer is defined as a one-dimensional array in the HLS code (line 10). In order to compact its layout, the

```
1: typedef ap_axiu<32,2,5,6> AXI_VAL;
2: #define n 480
3: #define m 640
4: #define sw 2
5: #define WT (2*sw+1) // support window size
6: #define swSAD 2
7: #define W (2*swSAD+1) // aggregation window size
8: #define n_pixels n*m
9: #define latency m*(WT-1)/2+(WT-1)/2+ m*(W-1)/2+(W-1)/2
10: void main( hls::stream<AXI_VAL> &pixel_R, hls::stream<AXI_VAL> &pixel_L,
11:                 hls::stream<AXI_VAL> &disp_out)
12: {
13:   #pragma HLS INTERFACE ap_ctrl_none port=return
14:   #pragma HLS INTERFACE axis port=pixel_R
15:   #pragma HLS INTERFACE axis port=pixel_L
16:   #pragma HLS INTERFACE axis port=disp_out
17:   AXI_VAL in_R, in_L, out;
...    // define other variables here
20:    for (int i=0;i<n_pixels+latency;i++) {
21:          #pragma HLS PIPELINE II=h //equal to the number of partitions
22:          if (i<n_pixels) {
23:             in_R=pixel_R.read(); // read the right pixel
24:             in_L=pixel_L.read(); // read the left pixel
25:          }
...        here define the operations to calculate the disparity "disp"
27:           if (i>=latency) {
28:             out.data=disp; // prepare output in accord to AXI4-Stream protocol
interface
29:             out.strb = in_R.strb;
30:             out.keep = in_R.keep;
31:             out.user = (i==latency) ? 1 : 0;
32:             out.last = (i==npixels+latency-1) ? 1 : 0;
33:             out.id = in_R.id;
34:             out.dest = in_R.dest;
35:             disp_out.write(out); // send data to the output stream
36:           }
37:      } // end for
38:} // end main
```

Figure 3.7 C-code used in the HLS tool to describe the top-level architecture of *Disparity computing* block.

#pragma HLS resource directive has been used to force the synthesizer to use Block RAMs (BRAMs), configured as Simple-Dual Port (S2P) memories, instead of Look-Up-Tables (LUTs). The #pragma HLS dependence *inter=false* directive instructs the tool to eliminate false loop dependencies, giving the information that a write and a read operation never occur on the same memory location in the same clock cycle. In such a way, the synthesizer can schedule contemporary write and read operations on the line buffer and the loops can be pipelined with the lowest iteration interval. The pointers *wr_ptr* and *rd_ptr* indicate the memory locations where the write and read operations are performed, respectively, thus making the BRAMs able to operate as FIFOs. When a new pixel enters the buffer, each pixel within the support window shifts to the next register. The pixels stored in the last column of the support window (except the one belonging to the last row) are packed into an $8\times(W_T-1)$-bit word and written at the *wr_ptr* location of the line buffer. Similarly, the word stored in the *rd_ptr* location is read from the line buffer and unpacked into (W_T-1) 8-bit pixels, which are then inputted to the registers of the first column of the support window (except the one in the first row). The pointers *wr_ptr* and *rd_ptr* are initialized to 0 and $m-W_T-1$, respectively, and then updated in a circular way after each write and read operation is completed.

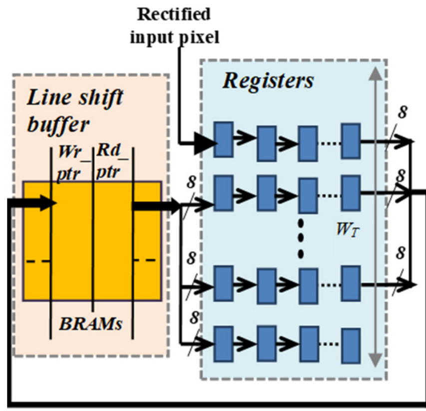


Figure 3.8 Structure designed for all the buffers within the Disparity computing block.

```
// defined outside the main function
1:void win_shift(ap_uint<8> win[WT][WT])
2:{
3:    for (int i=0;i<WT;i++){
4:          for (int j=WT-1;j>0;j--) {
5:                win[i][j]=win[i][j-1];
6:          }
7:    }
8:}
...
// within the main function
10:    ap_uint<8*(WT-1)> Buff[m-WT];
11:    #pragma HLS RESOURCE variable=Buff core=RAM_S2P_BRAM
12:    #pragma HLS DEPENDENCE variable=Buff inter false
13:    ap_uint<8> win[WT][WT];
14:    #pragma HLS ARRAY_PARTITION variable=win complete dim=0
15:    for (int i=0;i<WT-1;i++) {
16:          Buff[wr_ptr].range(8*i+7,8*i) = win[i][WT-1];
17:    }
18:    win_shift(win);
19:    win[0][0]=ap_uint<8>(in.data);
20:    for (int i=1;i<WT;i++) {
21:          win[i][0]=Buff[rd_ptr].range(8*(i-1)+7,8*(i-1));
22:    }
// update the pointers
23:    if (wr_ptr == 0)
24:        wr_ptr = m-WT-1;
25:    else
26:        wr_ptr --;
27:    if (rd_ptr == 0)
28:        rd_ptr = m-WT-1;
29:    else
30:        rd_ptr --;
...
```

It is worth noting that the designed line buffer acts as W_T-1 shift line buffers, each storing m-W_T 8-bit pixels. For typical image resolutions, only few 18kbit BRAMs are required to implement such buffers. As an example, with the 640×480 image resolution and W_T=5, each shift line buffer, which must store approximately 20kbit, can be implemented using only 2 BRAM primitives. The $W_T \times W_T$ registers of the support window are defined in the code of Figure 3.8 as a two-dimensional array (line 13). The #pragma HLS ARRAY_PARTITION with the option *DIM=0* ensures that each register is implemented using Flip-Flops (FFs) and it is made individually accessible. The *win_shift* function is called within the code (line 18) to activate the shifting of each pixel when a new input data enters the buffer. The *ap_uint<8>* data type (e.g. arbitrary precision unsigned 8-bit integer) is used for the pixels within the *Image Buffer*. In order to minimize resources requirements, it is important to properly set the bit-length of each variable. The aforementioned described approach has been adopted for all buffers used by the proposed *Disparity computing* module.

The HLS code of the *Transformation* module is reported in Figure 3.9(a). It can be seen that the generic element *WCV_el* of the vectors *WCVR* and *WCVC* is computed by simple shift operations. Moreover, the comparisons required in equation (4a) are performed in a bitwise fashion on the variable $diff_n = \lfloor \Delta c/4 \rfloor$ (lines 9-16). It has been found that

such a coding style allows reducing the number of LUTs occupied by each *Transformation* module by about 7% with respect to the case in which *diff_n* is compared to the constant values 0, 1, 2, 4, 8, 12 and 16, as reported in equation (4a). The HLS function reported in Figure 3.9(b) computes the auxiliary element AV_{NW}. The latter is the number of pixels within the NW region of the transformation windows having intensity values greater than the central pixel. The HLS functions designed to compute the others elements AV_{NE}, AV_{SW}, AV_{SE}, AV_{RC} and AV_{CC} of the auxiliary vectors AVR and AVC differ from the previous one just for the pixel indexing within the corresponding region of interest. It is worth noting that the word-length of the computed elements depends on W_T. In Figure 3.9(b), W_T=5 is assumed; therefore 3 bits are enough. For larger W_T, the word-length must change accordingly.

```
1:  ap_int<6> ACT (ap_uint<8> pixel, ap_uint<8> central) {
2:  ap_int<6> WCV_el;
3:  int diff;
4:  ap_uint<8> deltac;
5:  ap_uint<6> diff_n;
6:  diff = pixel - central;
7:  sign = (diff>=0) ? 1 : -1;
8:  deltac = ap_uint<8>(diff * sign);
9:  diff_n = deltac >> 2; // division by γc=4
10: if (diff_n[4] | diff_n[5]) {WCV_el = 0;} // diff_n >=16
11: else if (diff_n[3] && diff_n[2]) { WCV_el = sign;} //
12<=diff_n<16
12: else if (diff_n[3]) { WCV_el = sign << 1;} // 8<= diff_n <12
13: else if (diff_n[2]) { WCV_el = sign << 2;} // 4<= diff_n <8
14: else if (diff_n[1]) { WCV_el = sign << 3;} // 2<= diff_n <4
15: else if (diff_n[0]) { WCV_el = (sign << 4) - (sign << 2);} // diff_n
= 1
16: else { WCV_el = sign << 4;} // case diff_n = 0
17: return WCV_el;
```

(a)

```
1:  ap_uint<3> NW (ap_uint<8> win[WT][WT]) {
2:   ap_uint<3> AVNW = 0;
3:   for (int i=0; i<sT; i++) {
4:        for (int j=0; j<sT; j++) {
5:             if (win[i][j]>=win[sT][sT]) {
6:                AVNW++;
7:             }
8:        }
9:  }
10: return AVNW;
```

(b)

Figure 3.9 (a) Computing WCVR and WCVC vectors. (b) Computing the element AV_{NW}.

The HLS code of the *Windows Selection* module is reported in Figure 3.10. It receives as inputs the d_r aggregation windows (each referred to one of the processed d_r disparity levels) simultaneously provided by the *MC Buffer*. It can be seen that, at the *k*-th iteration, with $k=0,...,h-1$, the proper group *win_sel* of 5×6 data is outputted. It is selected from the slice *win* of $5\times(5+dr-1)$ data received as input.

```
1: void win_select(ap_int<6> win[W][dr+W-1], ap_int<6> win_sel[W][W+Pl-1], int k)
2: {
3:    for (int i=0;i<W;i++){
4:        for (int j=0;j<W+Pl-1;j++) {
5:             win_sel[i][j] = win[i][j+k*Pl];
6:        }
7:    }
8: }
```

Figure 3.10 HLS-code for the *Window Selection* block description.

In the example reported in Figure 3.11, where *W=5* and *Pl=2,* the number of produced output data is enough to arrange *Pl=2* candidate aggregation windows. Obviously, if the parallelism level increases, the module must change accordingly.

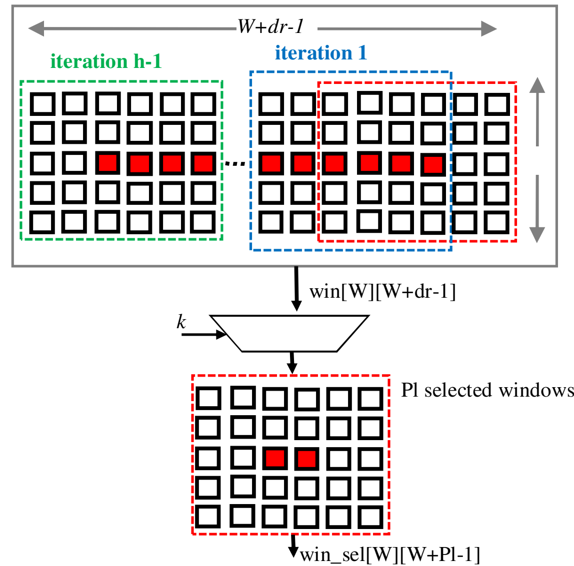


Figure 3.11 Running example for the Windows Selection module when *W*=5 and *Pl*=2.

Finally, to compute the dissimilarity, the HLS-code reported in Figure 3.12 has been written. Such a code refers to the case $W= W_T = 5$. In order to limit the number of load and store operations on input and output data as much as possible, thus avoiding potential failures of the HLS tool during the synthesis, the $W_T{\times}W_T$ 6-bit elements of each vector *WCVR* (*WCVC*) and the six 3-bit elements of its auxiliary vector *AVR* (*AVC*) are packed within one 168-bit word. Each element can be separately accesses specifying the bit positions of interest through the *.range* selection (line 22 and subsequent). The *for* loop at line 16 scans all the *Pl* candidate aggregation windows coming from the right channel to compute numerator and denominator of equation (6). It can be seen that shifting operations are exploited to perform multiplications, thus minimizing the hardware resources requirements. Conversely, the division operation (line 75) that furnishes the generic dissimilarity value *D* is performed between integer numbers that are not power of two. Using the *ap_ufixed<27,21,AP_RND>* data type, the resulting non integer dissimilarity value is treated as a 27-bit fixed-point number with a 21-bit integer part. The *AP_RND* option rounds a given non integer value to the nearest fixed-point value with the specified precision. The latter has been chosen based on an error measurement: indeed 27-bit fixed-point number with a 21-bit integer part is the minimum representation for which hardware system does not introduce significant degradation in quality result.

```
1: void ComputeDissimilarity (ap_int<168> WCVR [W][W],
2:                 ap_int<168> WCVC [W][W+P-1],
3:                 ap_uint<8> R[W][W],
4:                 ap_uint<8> C[W][W+P-1],
5:                 ap_ufixed<27,21,AP_RND> D[P]) {
6:  ap_uint<5> weightL, weightR;
7:  ap_int<7> MCW[WT*WT];
8:  ap_int<3>MCA[6];
9:  ap_int<2> sign[WT*WT+6];
10: ap_uint<21> num; // numerator of equ. (1)
11: ap_uint<13> den; // denominator of equ. (1)
12: ap_uint<10> MC;
13: ap_uint<18> par_num, par_num_sh; //w(Q,C)×MC and w(Q,C)×w(P,R)×MC
14: ap_uint<9> par_den; // w(Q,C)×w(P,R)
15: int nbWT=WT*WT*6;
16: for (int y=0;y<P;y++) {
17:    num=0;
18:    den=0;
19:    for (int i=0;i<W;i++){
20:       for (int j=0;j<W;j++) {
21:          MC=0;
21:          for (int comp=0;comp< WT*WT;comp++) {
22:             MCW[comp]=WCVR [i][j].range(6*comp+5,6*comp)-
23:                         WCVC [i][j+y].range(6*comp+5,6*comp);
24:              sign[comp]=(MCW[comp]>=0)?1:-1;
25:              MC=MCW[comp]*sign[comp]+MC;
26:          }
27:          for (int comp=0;comp< 6;comp++) {
28:             MCA[comp]=WCVR[i][j].range(nbWT+2*comp+1,nbWT+2*comp)-
29:                      WCVC[i][j+y].range(nbWT+2*comp+1,nbWT+2*comp);
30:              sign[WT*WT+comp]=(MCA[comp]>=0)?1:-1;
31:              MC=MCA[comp]*sign[WT*WT+comp]+MC;
32:          }
33:          weightR=ComputeWeights(R[i][j],R[(W-1)/2][(W-1)/2]);
34:          weightL=ComputeWeights(C[i][j+y],C[(W-1)/2][y+(W-1)/2]);
35:         // compute w(Q,C)×w(P,R) and w(Q,C)×MC
36:          if (weightL[4]){
37:            par_den=int(weightR)<<4;
38:            par_num=ap_uint<14>(MC)<<4; }
39:       else if(weightL[3]&&weightL[2]){
40:          par_den=int(weightR)<<4-int(weightR)<<2;
41:          par_num=ap_uint<14>(MC)<<4-ap_uint<14>(MC)<<2;}
42:       else if(weightL[3]){
43:          par_den=int(weightR)<<3;
44:          par_num=ap_uint<14>(MC)<<3;}
45:        else if(weightL[2]){
46:          par_den=int(weightR)<<2;
47:          par_num=ap_uint<14>(MC)<<2;}
48:        else if(weightL[1]){
49:          par_den=int(weightR)<<1;
50:          par_num=ap_uint<14>(MC)<<1;}
51:         else if(weightL[0]){
52:           par_den=weightR;
53:           par_num=ap_uint<14>(MC);}
54:         else{
55:           par_den=0;
56:           par_num=0;}
57:        // compute w(Q,C)×w(P,R)×MC
58:         if (weightR[4]){
59:            par_num_sh=ap_uint<14>(par_num)<<4; }
60:         else if(weightR[3]&&weightR[2]){
61:            par_num_sh=ap_uint<14>(par_num)<<4-
                              ap_uint<14>(par_num)<<2;}
62:         else if(weightR[3]){
63:            par_num_sh=ap_uint<14>(par_num)<<3;}
64:         else if(weightR[2]){
65:            par_num_sh=ap_uint<14>(par_num)<<2;}
66:         else if(weightR[1]){
67:            par_num_sh=ap_uint<14>(par_num)<<1;}
68:         else if(weightR[0]){
69:            par_num_sh=ap_uint<14>(par_num);}
70:         else par_num_sh=0;
71:         num=num+ap_uint<20>(par_num_sh));
72:         den=den+ap_uint<12>(par_den));
73:      }
74:   }
75:   D[y]=ap_ufixed<27,21, AP_RND>(num)/
            ap_ufixed<27,21, AP_RND>(den);
76:  }
77:}
```

Figure 3.12 The HLS-code used to compute the dissimilarity.

```
1:  void SelectMin(ap_ufixed<27,21,AP_RND> D[Pl], ap_uint<1>
minind,
2:                      ap_ufixed<27,21,AP_RND> min){
3:   min=Diss[0];
4:   for (int i=0; i<Pl; i++) {
5:        if (min>D[i]) {
6:            min=D[i];
7:            minind=i;
8:        }
9:   }
    ...
// within the main function
1:  ap_ufixed<27,21,AP_RND> currD;
2:  ap_ufixed<27,21,AP_RND> minD;
3:  ap_uint<5> currdisp;
4:  ap_uint<1> currminind;
5:  for (int i=0; i<h; i++) {
6:       ComputeDissimilarity(WCVR,WCVC, R, C, DissGroup);
7:       SelectMin(DissGroup, currminind, currD);
8:       if ((i==0) ||(currD<minD)) {
9:            minD=currD;
10:           currdisp=i*Pl+currminind;
11:      }
12: }
    ...
```

Figure 3.13 HLS-code of the *SelectMin* function and its call in the main function.

The *Pl* dissimilarity values computed in parallel are then processed by the module *SelectMin,* which generates the outputs *currD* and *currminind*, as illustrated in Figure 3.13. *currD* is the minimum between the *Pl* dissimilarities within the currently processed partition, whereas *currminind* is the index, ranging between 0 and *Pl*-1, where *currD* is located. It is important noting that, in order to process all the *h* partitions in which the disparity range is split, the functions *ComputeDissimilarity* and *SelectMin* are called *h* times. However, to take into account that at the generic i-th iteration (with $i=0,...,h-1$) the processed partition involves indices actually ranging between $i \times Pl$ and $i \times Pl+Pl-1$, the statement at line 10 of Figure 3.13 is performed to properly scale *currminind* to a value within the actual disparity range (i.e. between *dmin* and *dmax*). After *h* iterations, *currdisp* provides the disparity $\delta_{(x,y)}$.

3.4 RESULTS

3.4.1 Accuracy measurements

The accuracy achievable by the proposed algorithm was evaluated by processing the well-known Middlebury benchmarks Tsukuba, Venus, Teddy and Cones [83]. The error rates were measured for several values of W_T and W by means of purpose-written C++ routines. To evaluate the percentages of incorrect disparity values, the computed disparity maps were segmented in the typical nonoccluded (*Nonocc*), discontinuities (*Disc*), and all (*All*) regions, and then compared to the available ground truths. Samples of computed disparity maps are reported in Figure 3.14, where, darkest regions correspond to smaller disparity values, which are associated to farther objects from the stereo camera that acquired the images. Conversely, closer objects correspond to higher disparity values.

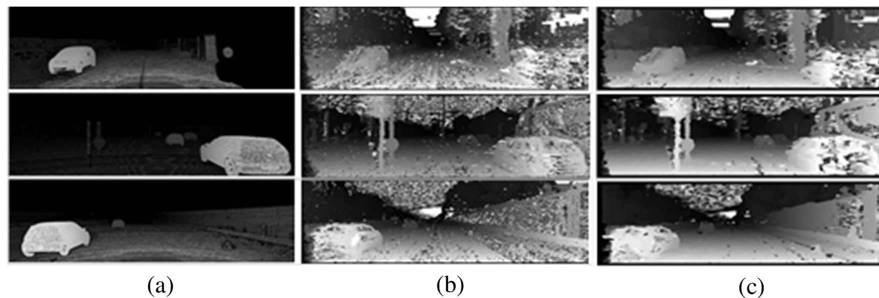


Figure 3.14 Samples of disparity maps: a) ground truths; b) obtained with W_T=W=5; c) obtained with W_T=W=13.

Average errors are summarized in Table 3.1. By comparing the proposed method to the closest competitors, it can be seen that average error rates are reduced by up to 31%, 10% and 27% in the *All*, *Nonocc* and *Disc* region, with respect to [64]. Error rates

comparable to [65] are reached even though the computational complexity of the proposed algorithm is much lower. The percentages of incorrect disparity values are also reduced by up to 56%, 63% and 35% with respect to [68]. When compared with [69], the novel algorithm exhibits error rates at least 18%, 26% and 38% lower, even when smaller window sizes *W* are used. The average error rate in the *All* region is reduced by up to 50% also with respect to [70]. It is however clear that from this comparison the winner is the algorithm demonstrated in [67].

Table 3.1 Error rates obtained by the novel stereo vison algorithm and other competitors for several test images.

	W_T	W	Tsukuba			Venus			Teddy			Cones		
			All %	Nonocc %	Disc%	All %	Nonocc %	Disc %	All %	Nonocc %	Disc%	All %	Nonocc %	Disc %
[63]	-	-	12.24	10.27	-	14.82	13.74	-	30.96	24.41	-	25.43	19.04	-
	-	-	12.26	10.31	-	12.72	11.6	-	25.65	19.64	-	19.2	25.79	-
[64]	5	5	11.8	11.4	39.1	7.02	5.49	15.5	18.9	8.09	21	18	4.79	13.54
	5	9	8.9	8.5	36.3	3.37	1.91	12.55	16.4	5.98	18.9	16.4	4.06	12.88
	5	13	7.6	7.2	35.7	2.65	1.33	10.9	15.5	5.73	18.4	15.9	4.54	13.53
	9	5	8.6	8.2	36.2	4.91	3.28	15.55	17.6	6.55	19.4	17.9	4.64	13.8
	9	9	7.23	6.84	34.9	2.96	1.46	12.6	16.1	5.64	18.71	16.4	4	13.09
	9	13	6.7	6.28	35.1	2.55	1.2	10.7	15.2	5.43	18.5	15.9	4.63	13.96
	13	5	7.5	7.1	35.9	4.15	2.5	15.8	17.6	6.48	20.2	18.4	5.36	15.84
	13	9	6.76	6.34	35.2	2.89	1.31	12.87	16.3	5.68	19.3	16.84	4.4	14.06
	13	13	6.32	5.92	34.9	2.63	1.17	11.38	15.3	5.5	19.2	16.4	5	15.1
[65]	3	3	14.86	13	17.93	11.42	9.94	22.23	21.11	12.64	25.12	14.95	5.06	13.1
	3	5	13.17	11.37	15.76	7.84	6.32	17.26	18.6	10.13	22.53	13.74	4	10.73
	3	7	11.76	10.01	14.08	5.17	3.68	14.06	17.22	8.82	21.27	13.11	3.58	9.95
	5	3	10.91	9.02	15.25	7.09	5.51	20.68	18.87	10.1	23.83	14.52	4.44	12.26
	5	5	9.97	8.12	13.44	5.35	3.77	17.3	17.68	9.07	22.43	13.74	3.9	11.06
	5	7	9.24	7.46	12.45	3.93	2.38	14.97	16.81	8.35	21.68	13.22	3.56	10.29
	7	3	9.36	7.51	15.02	4.99	3.36	21.32	18.39	9.55	24.77	14.84	4.74	13.35
	7	5	8.87	7.05	14.17	4.14	2.53	19.15	17.54	8.87	23.61	14.03	4.12	11.94
	7	7	8.27	6.52	13.06	3.44	1.88	16.74	16.68	8.14	22.6	13.49	3.81	11.2
	7	13	6.93	5.27	12.05	2.63	1.3	12.03	15.23	7.25	21.15	13	4.3	11.94
	13	3	7.41	5.65	17.58	3.95	2.31	26.1	20.63	11.91	29.62	18.27	8.61	18.88
	3	13	8.89	7.22	11.1	2.68	1.35	10.83	15.02	7.06	19.21	12.42	3.84	10.49
[66]	-	5	23	21	36	27	21	43	31	33	45	38	38	44
	-	13	14	11	30	11	11	30	27	27	35	28	27	38
	-	41	10	7	32	8	7	29	13	13	34	16	14	31
[67]	-	7	3.09	2.57	8.89	0.52	0.3	2.65	12.3	6.96	16.8	8.59	2.91	8.48
[68]	3	-	16.83	15.02	21.66	11.58	10.11	23.67	21.7	13.4	27.16	20.21	10.8	22
[69]	-	7	7.09	5.27	21.4	5.66	4.33	29.4	24.4	16.9	33.4	20.3	11.8	23.1
[70]	-	5	6.84	-	-	6.07	-	-	27.3	-	-	22.4	-	-
[71]	-	19	6.8	-	-	2.8	-	-	-	-	-	-	-	-
New	5	5	16.44	8.84	20.04	12	3.87	10.33	7.36	6.04	18.67	11.13	9.26	17.46
	5	9	15.84	8.67	21.74	11.47	3.83	10.64	4.99	3.78	12.78	10.81	9.13	18.72
	5	13	15.46	8.81	23.35	11.47	4.37	12.36	3.78	2.74	11.89	10.39	8.82	22.23
	9	5	13.38	6.97	18.31	10.27	3.56	10.58	4.32	3.09	17.75	8.58	6.66	15.31
	9	9	13.72	7.42	20.56	10.48	3.8	11.12	3.23	2.21	13.45	8.25	6.43	16.35
	9	13	13.32	7.49	21.08	10.6	4.43	12.49	2.8	1.95	12.29	8.02	6.39	20.25
	13	5	12.07	6.59	18.72	9.84	4.08	12.05	3.45	2.36	18.13	7.3	5.35	14.55
	13	9	11.86	6.68	19.27	9.51	3.98	11.82	2.67	1.73	13.96	7.33	5.49	16.35
	13	13	11.49	6.5	19.19	9.84	4.35	12.56	2.4	1.58	11.39	7.2	5.5	18.5

It is worth pointing out that most algorithms perform much worse in elaborating Teddy and Cones stereo pairs than they do in computing the disparity maps of Tsukuba and Venus. This is particularly true for the algorithms proposed in [69] and [67]. When

processing Teddy and Cones, the former shows an error rate ~4 times higher than that obtained for Tsukuba and Venus. A more pronounced difference is observed for the algorithm described in [67], which achieves ~6 times worse error rates for Teddy and Cones. This is most probably due to the fact that such benchmarks are more complex and their ground-truth disparity maps were acquired by more precise structured lighting technique. Conversely, from Table 3.1 it can be appreciated that the proposed algorithm shows a more uniform behaviour across the Middleburry benchmarks and performs even better with Teddy and Cones than with Tsukuba and Venus.

3.4.2 Implementation

The timing reported in Figure 3.15 summarizes the running of the whole system depicted in Figure 3.5. For external memory, a ping-pong strategy has been adopted, storing even and odd frames into different memory areas. The first acquired images pair is software rectified and then the results is transferred to the PL for computing the disparity map; the latter is send to the processor for refinement, while in the meantime a new pair of images is acquired, and so on. Such a process requires a total bandwidth of 25.6Gb/s for communication with the external memory, which is well below to the maximum bandwidth sustainable by the adopted platforms (that is 33.3Gb/s for Z7020 with speed-grade -1 and 41.6 Gb/s for Z7045 with speed-grade -3) [82].

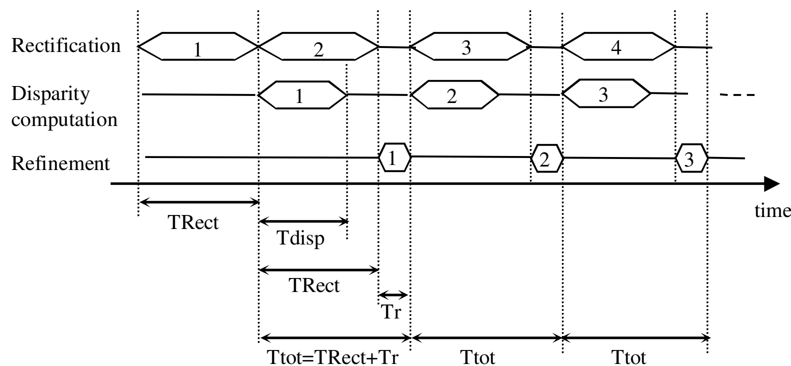


Figure 3.15 Timing of the whole process performed by the proposed stereo vision embedded system.

TRect, *Tdisp* and *Tr* illustrated in Figure 3.15, are respectively the time required to rectify the input images, to compute the disparity map and to refine it; from the timing it is possible to observe that the highest throughput can be achieved if the time *Tdisp* is entirely hidden by the rectification execution time. In this way, after the latency of $2\times TRect+Tr$, the overall system produces new results with the throughput of $Ttot=TRect+Tr$. For the

chosen 640×480 image resolution, the minimum software rectification time *TRect*, achieved compensating both radial and tangential distortions as detailed in equations (1)-(3), is *~72ms*. If the window sizes W_T and W are set to 5, to hide the time required to compute the generic disparity map in accordance with the novel algorithm, at least the parallelism level $Pl=2$ must be exploited. In this case, $Tdisp\approx 39ms$ has been measured through the *AXI_Timer* IP core visible in Figure 3.5. The refinement time $Tr\approx 5.5ms$ has been also measured. This means that, the proposed system can reach a frame rate of about 12.8fps. Achieved speed performances are limited by the rectification software process and can be certainly improved by adopting a simplified rectification method, e.g. renouncing to radial and tangential distortion compensation [69]. In this case, the time required for rectification is reduced to about *51ms* and consequently the frame rate grows up to about 17.7fps. If higher throughput rates are mandatory, alternative hardware/software partitioning strategies have to be chosen. In particular, frame rates up to 81fps and 101fps can be obtained with the XC7Z020 and the XC7Z045 device, respectively, when the image rectification and the disparity map refinement are hardware implemented, whereas the ARM Cortex-A9 processor is exploited just for configuration and communication control.

Table 3.2 summarizes implementation results obtained for several parallelism levels Pl and different windows sizes W_T and W, when the Zynq-7000 part is used. It can be observed that achieved speed performances strongly depend on Pl. In fact, at a parity of the windows sizes (i.e. W_T=W=5) a ~50% higher frame rate is reached by doubling Pl, but spending ~32% more LUTs and ~2% FFs. Conversely, resources requirements change much more significantly as W_T and W grow. This happens since, by increasing W_T and W, the number of adjacent elements (namely $(W+W_T-1)^2$) processed by the proposed algorithm grows accordingly. As an example, when both W and W_T change from 3 to 5, the number of adjacent elements processed becomes ~3 times higher, and, as visible in Table 3.2, also LUTs and FFs requirements are about 3x. Analogously, examining the case in which W_T=W=13, Pl=8 and dr=32, it can be estimated that the amounts of occupied LUTs and FFs would be about 7.7x higher than the architecture operating with W_T=W=5, Pl=8 and dr=32. Obviously, to support such large windows sizes, more advanced device families, could become necessary.

Apart the image resolution and the windows sizes, for each design mentioned in Table 3.2, it is also specified which additional operations, among the image acquisition (A), the rectification (R), the refinement (Re), and the visualization (V), are also performed on-chip. Characteristics of several competitors are also reported [64]-[67]. It is worth noting that the SoC implementation here described is the only one that includes all auxiliary circuitry needed to acquire the stereo images, to dispatch intermediate data to/from the DDR

memory, and to display the computed disparity maps. The architecture discussed in [69] appears as the cheapest one, but unfortunately the original paper does not provide accuracy measures.

Table 3.2 Implementation results for the proposed stereo vision embedded system, and state-of-the-art comparison.

	Resolution	W_T	W	**A**	**R**	**Re**	**V**	**dr**	**fps**	**Processor**	**Device**	**Resources**
[64]	640×480	5	5	N	Y	Injective Check	N	64	68	N	Virtex6 XC6VLX760	80270 Slices 112 DSPs 32 BRAM-18k
[65]	640×480	5	5	N	Y	Injective Check	N	60	54	N	Virtex7 XC7VX9880T	134153 Slices 112 DSPs 32 BRAM-18k
[66]	1280×1024	-	15	N	N	N	N	15	76	N	Cyclone II IIEP2C35F672C6	27061 LEs 398K Memory bits
[67]	1280×720	-	7	N	Y	Cross check Filling Median	N	64	60	N	Kintex-7 XC7K325T-FFG900	181866 LUTs 139704 FFs 738 DSPs 441 BRAM-36k
[69]	480×540	-	7	N	N	N	N	56	71	N	Custom TSM90nm	183k gates 6.72k Memory Bytes
[70]	1280×1024	-	5	N	N	N	N	15	76	N	Cyclone II IIEP2C35F672C6	22075 LEs 381K Memory bits
[71]	1280×720	-	19	N	N	N	N	31	117	N	Cyclone IV EP4CGX150CF23C8	62689 LEs 237K Memory bits
[72]	640×480	-	3	N	N	Cross check	N	135	80	N	Virtex5 XC5VLX50-3	12144 LUTs 12144 FFs 18 BRAM-18k
New Pl=8	640×480	3	3	Y	Y	Injective check Filling Median	Y	32	81	Y[2]	Zynq-7000 XC7Z020-CLG484	52691 LUTs 59715 FFs 93 DSPs 40 BRAM-18k 6 BRAM-36k
New Pl=4	640×480	5	5	Y	Y	Injective check Filling Median	Y	32	50	Y[2]	Zynq-7000 XC7Z045	137683 LUTs 141237 FFs 100 DSPs 129 BRAM-18k 6 BRAM-36k
New Pl=8	640×480	5	5	Y	Y	Injective check Filling Median	Y	32	101	Y[2]	Zynq-7000 XC7Z045	182933 LUTs 143223 FFs 100 DSPs 129 BRAM-18k 6 BRAM-36k
New Pl=2	640×480	5	5	Y	Y	Injective check Filling Median	Y	32	12.8 17.7	Y[1]	Zynq-7000 XC7Z045	121252 LUTs 146758 FFs 125 BRAM-18k 6 BRAM-36k

[1] In this implementation, the dual-core ARM processor is used for configuration, communication control, rectification and refinement. The 17.7 fps frame rate is achieved when the simplified rectification is exploited

[2] In this implementation the dual-core ARM processor is used only for configuration and communication control

A direct comparison with the system demonstrated in [67] that, as seen before, reaches the highest average accuracy is worthwhile. Differently from our proposal, such design exploits auxiliary card to acquire and visualize the disparity map results. It uses approximately the same LUTs and FFs resource of our system with $Pl=8$ and $W_T=W=5$, but it consumes more than 7 times DSP slices and more than 6 times memory resources.

For further comparisons with others competitors, the normalized figure of merit (FOM) defined in equation (7) is introduced. It allows taking into account simultaneously several characteristics of the compared implementations, such as the number of dissimilarity values computed per second (*Mdps)*, the average accuracy (*Acc*), the on-chip integration level of the various operations required by a complete stereo vision system (*IntF*), and finally the normalized resource utilization (*NR*).

$$FOM = \frac{Mdps \times Acc \times IntF}{NR} \quad (7)$$

Mdps combines the image resolution with the frame rate and the disparity range. Conversely, *Acc* averages the percentages of correct disparities obtained in the three regions *All*, *Nonocc* and *Disp*. *IntF* counts the number of hardware or software on-chip integrated operations, among: image acquisition and visualization; rectification; disparity computation; refinement; and interfacing with external peripherals. Finally, *NR* accounts the total number of memory bits used by each implementation normalized to the supported disparity range. It is obtained by considering that each slice of Virtex-5, Virtex-6 and Virtex-7 devices consists of four 6-input LUTs and eight FFs and it is equivalent to 264 memory bits. Analogously, each LE of Cyclone II and Cyclone IV devices corresponds to 17 memory bits, whereas the generic LUT of a Zynq-7000 device is equivalent to 64 memory bits. The total number of memory bits is then divided by d_r, leading to a normalized FOM of 0.28, 0.12, 0.83 and 0.64 for [64]-[67] respectively. On the other hand, the most efficient configuration exploiting the novel algorithm features a normalized FOM of 0.41, which is 31.7% and 70.7% higher than [64]-[65] but 50.6% and 36% lower than [66]-[67]. This result confirms weakness and strengths above discussed for each analyzed design, demonstrating that the proposed embedded system, with its high integration level and high-speed performance, well trade-off the characteristics required for the inclusion within consumer electronics and multimedia products.

3.5 THE NOVEL INTEGRAL IMAGE COMPUTING APPROACH

3.5.1 Background

The integral image (IIM) is an intermediate image representation, originally proposed for texture mapping applications in computer graphic in 1984 [83], and later taken over by the Viola-Jones face detector algorithm [85]. Since then, IIM has been efficiently exploited for fast implementation of image pyramids in multi-scale computer vision algorithms such as Speeded-Up Robust Features (SURF) [86] and Fast Approximated SIFT [87]. In such a

context, using integral images allows improving execution speed for computing box filters by substituting computationally expensive multiplications with three addition operations [85]. This allows all box filters to be computed at a constant speed, irrespective of their size, leading to an important advantage in feature detection techniques that utilize multi-scale analysis. Indeed, such algorithms generally require calculation of variable-size box filters to implement different scales of an image pyramid. As an example, SURF requires computation of 9× 9-sized box filters for implementation of its smallest scale, whereas 195×195-sized box filters have to be processed for the largest scale within the image pyramid [86]; without an integral image, computing the larger filters would take almost 500 times longer than the smallest one [87]. In [18], integral images were also efficiently used for computing the matching cost function in a window-based method for dense stereo correspondence.

In general, given an input image A, the value of its integral image at any location (x, y) can be computed as the sum of all the pixels above and to the left of (x, y), including the current location. This can be mathematically stated as in (8).

$$IIM(x,y) = \sum_{i=0}^{x}\sum_{j=0}^{y} A(x,y) \tag{8}$$

Although equation (8) involves only additions, the total number of operations significantly depend on the input image size, e.g. to process an n×m image $\cong \frac{1}{4}\times n^2\times m^2$ additions have to be performed [88]. To overcome this issue, the recursive approach introduced by Viola and Jones [85] allows reducing the total number of operations, transforming (8) into expressions (9) and (10). There, the IIM value at the generic location (x, y) is computed as the sum between IIM(x, y), i.e. the integral image value located above to the current position, and the cumulative row sum RS calculated at (x, y). The latter, as defined in (9), is the accumulation of the pixels along the x-th row within the input image A.

$$RS(x,y) = A(x,y) + RS(x,y-1) \tag{9}$$

$$IIM(x,y) = IIM(x-1,y) + RS(x,y) \tag{10}$$

Equations (9) and (10) represent a two-stage system that operates in a serial fashion: the first stage computes the cumulative row sum at a specific image location and forwards the data to the second stage for calculation of the integral image value at that particular location. It is then clear that the operation of each individual stage also depends on data produced by previous iterations, which is not very attractive for hardware implementations of real-time power-constrained embedded IoT systems.

A more exhaustive analysis of equations (9) and (10) shows that the first stage requires the cumulative row sum to be computed in a serial way for a single row of the input image, whereas the second stage is more complex as it needs data from the previous row to calculate an integral image value. Therefore, it may be possible computing the cumulative row sum for all rows independently and simultaneously, but respecting the dependency on data from the neighboring row for equation (10). Figure 3.16 illustrates an example of a 4×4 input image for which integral image values are calculated by processing all rows in parallel. It can be easily observed that the integral image value for the second pixel in the third row cannot be computed until the integral image value for the second pixel in the second row is calculated.

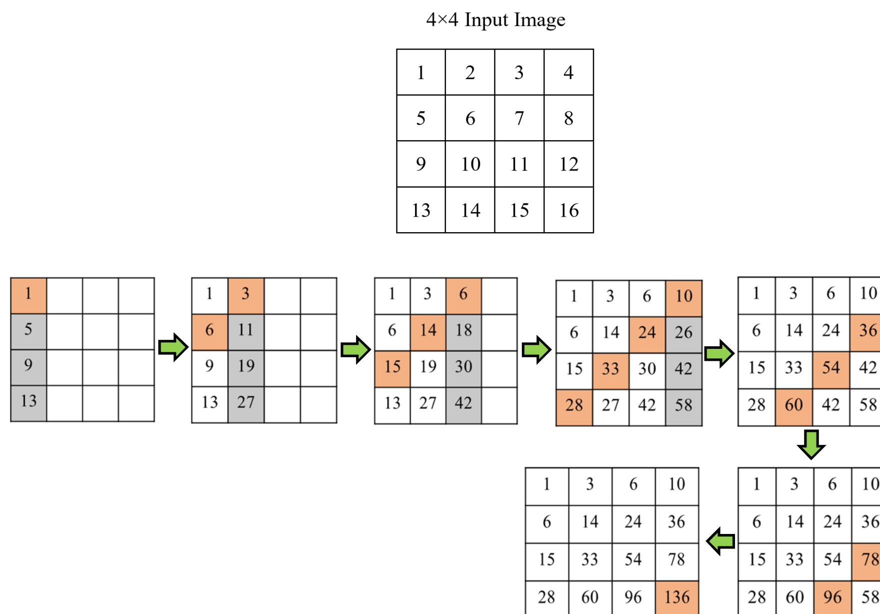
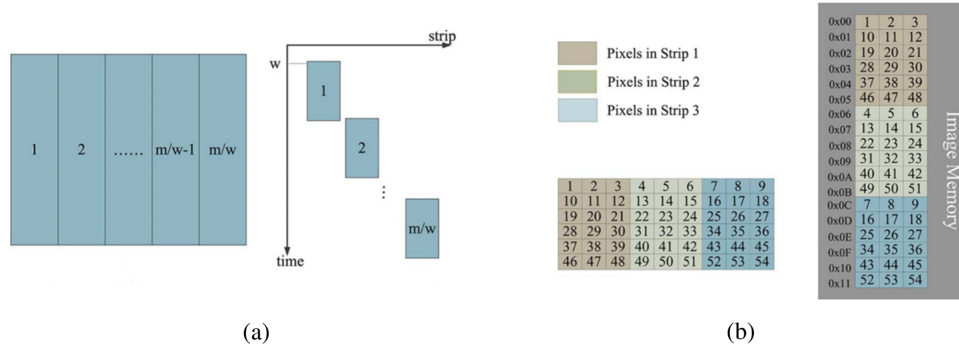


Figure 3.16 Example of IIM values calculation exploiting the recursive approach and row-parallel computation; grey squares represent locations for which only *RS* has been calculated, whereas orange-highlighted positions are related to the final IIM values updated in the current step.

The idea above discussed has been widely demonstrated in previous works concerning FPGA-based circuits for IIM computation [88], [90]-[91]. In some cases [90], further speed-up of the row-parallel based approach has been achieved by dividing the input image into horizontal strips, each having width w, as depicted in Figure 3.17(a). Such a technique allows working in parallel on the columns of each row, reducing the number of steps required to compute the whole integral image with respect to the classic serial recursive approach, but introducing two main issues. The former is related to the modality with which input pixels are transferred from the source to the custom accelerator implementing IIM calculation. Indeed, in embedded visual IoT nodes the source is typically a camera with a digitalize block that converts analog information into a raster-order stream of digital pixels.

Conversely, the architecture presented in [90] assumes that the input image is stored following the memory organization illustrated in Figure 3.17(b), thus requiring an input buffer, sized as required by the frame dimensions, and additional processing times to write and read such a memory bank. The second issue regards the choice of the width w, which should always be a divider of the number of columns m for the input image. As stated by authors in [90], for an $n \times m$ input image, this leads to a total number of steps $S_{num}=w+[(m/w)\times n]$. Considering the 4×4 input image of the example in Figure 3.16, and assuming w=4, the approach proposed in [90] would require eight steps.



(a) (b)

Figure 3.17 (a) Strips-based IIM computing approach adopted in [90]; (b) corresponding input pixels organization within the memory bank.

3.5.2 The proposed idea and its hardware implementation

As discussed in the previous paragraph, the recursive approach permits to parallelize computation of cumulative row sums referred to adjacent rows of the input image [85]. Based on such a consideration, the proposed idea [17] introduces a further level of parallelism along the horizontal dimension, defining a novel computational scheme able to speed-up the whole IIM process with respect to prior works [88],[90]-[91]. Specifically, in order to parallel process two adjacent pixels located within the same row x, operations required by equations (9)-(10) and (11)-(12) can be performed at the same time to concurrently produce IIM(x, y) and IIM(x, y+1).

$$RS(x, y+1) = A(x, y+1) + RS(x, y) \tag{11}$$

$$IIM(x, y+1) = IIM(x-1, y+1) + RS(x, y+1) \tag{12}$$

To easily demonstrate the effectiveness of the novel computing scheme, let examine the example of Figure 3.18, assuming that up to four rows and two columns for each row can be parallel processed. During the first step, the cumulative row sum is computed for the first two columns of the input image; then, the remaining two columns are processed

and in the meantime the final IIM values for locations (1,0) and (1,1) can be calculated by (10) and (12). This occurs because *RS* values previously calculated for the positions (0,0) and (0,1), being part of the first row, already correspond to the final integral image results. Steps 3-5 update the integral image values considering quaterne of pixels referred to adjacent columns and rows.

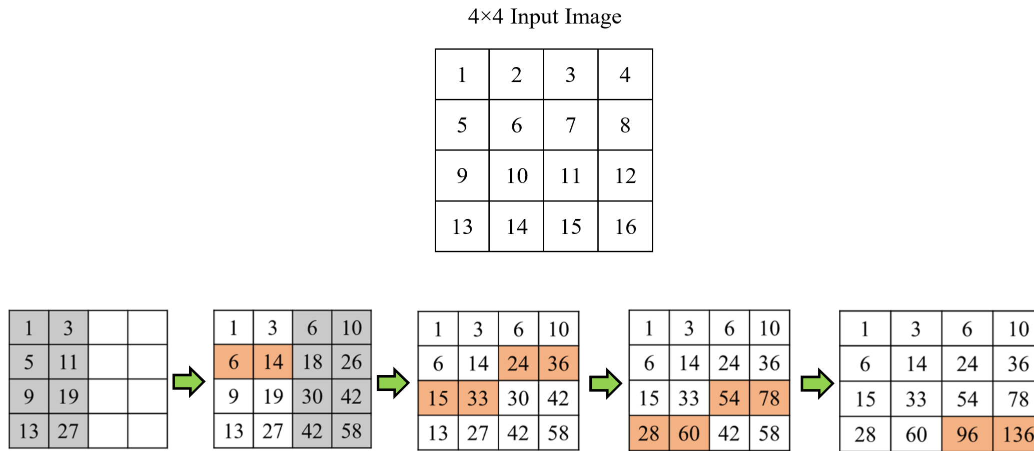


Figure 3.18 The proposed IIM computing scheme working on a 4×4 input image; grey squares represent locations for which only *RS* has been calculated, whereas orange-highlighted positions are related to the final IIM values updated in the current step.

In general, for an *n*×*m* input image, when the pixels along 4 rows and 2 columns are processed in parallel, the proposed computing scheme requires $S_{num} = \frac{m}{2} + \left(\frac{m \times n}{8}\right) + 1$ steps. Such a behavior well suites the partitioning of images with standard resolutions, whose dimensions are always divisible by two. Clearly, this approach can exploit higher parallelism to further accelerate the computation of integral images. However, even considering the example of Figure 3.16, a substantial improvement can be appreciated with respect to the approaches demonstrated in [88], [90]-[91] and described in Figures 3.16 and 3.17.

In order to hardware implement the approach described above, thus computing IIM(*x*, *y*) and IIM (*x*, *y*+1) in parallel, the number of additions must be doubled with respect to traditional row-parallel approach used in [88] and [91]. Typically, this would lead to an increased resources requirement. However, among the resources provided by modern FPGA devices, DSP slices are available. The computational scheme proposed here efficiently exploits the nice flexibilities of DSPs to increase the parallelism of the novel IIM building block without introducing detrimental effects on resources requirements. In particular, the DSPs can be configured to operate in SIMD mode, thus enabling the internal accumulator to execute the same operations on different data with the same hardware resources. Figure 3.19 schematizes a *k*-bit accumulator operating in SIMD dual-mode. The

k-bit inputs Ia and Ib are divided into two k/2-bit sub-words, which will be addressed to the two portions of the accumulator as the LSB and MSB parts. Each sub-portion also receives the proper sub-word of the previously produced output P. Which parallelism level can be exploited, obviously depends on the output word-length. In general, the data width required for representing the generic value IIM(x, y), when greyscale images are processed, is given by (13).

$$Width_{IIM} = log_2(m \times n \times 255) + 1 \quad (13)$$

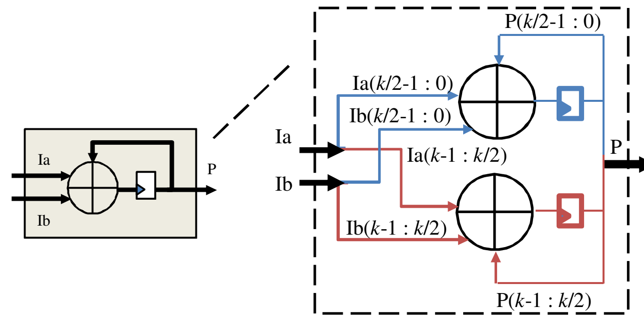


Figure 3.19 Schematization of a DSP operating as SIMD dual-mode accumulator.

Table 3.3 summarizes the data width versus the input image size. It can be seen that the cumulative row sum $RS(x,y)$ requires less bits than IIM(x,y). This is because the number of additions required computing it just depends on m. Therefore, for all the cases referenced in Table 3.3, one DSP slice operating in SIMD fashion and having $k \geq 40$ can certainly compute the cumulative row sums of two adjacent columns in parallel. This is the case for the DSP slices provided within both the Xilinx [92] and the Intel [93] devices families.

Table 3.3 Data width required for integral images computation of different sized input frames.

Image size	*RS(x,y)*	*IIM(x,y)*
320×240	17	25
640×480	18	27
1280×720	19	28
1920×1080	20	29

The hardware accelerator here proposed has been designed taking all the above considerations into account. The top-level architecture is depicted in Figure 3.20. The slices DSP_A, DSP_B, DSP_C and DSP_D perform accumulations in SIMD fashion and receive the inputs as properly prepared by the *SIMD Composition* block. The internal multiplexers are used to reset the accumulations when a new row initiates. The four k-bit accumulation

results are split into their LSB and MSB parts to be then sent to two separate *Column Adder* (CA) blocks that perform the subsequent accumulations over the columns. Each CA consists of a chain of four DSPs. Clearly, after the first four consecutive rows of the input image have been processed, the calculation of the IIM will depend on the fourth row produced in the previous round. To transfer the output of DSP_4 towards DSP_1, the row buffer is used with depth *N*-5. The eight outputs generated by the CAs furnish two adjacent pixels within the four rows processed in parallel. These values are properly packed by the *Concat Data* block that, after the initial latency of two clock cycles, outputs four streams of pixels, which are delayed from each other by one clock cycle.

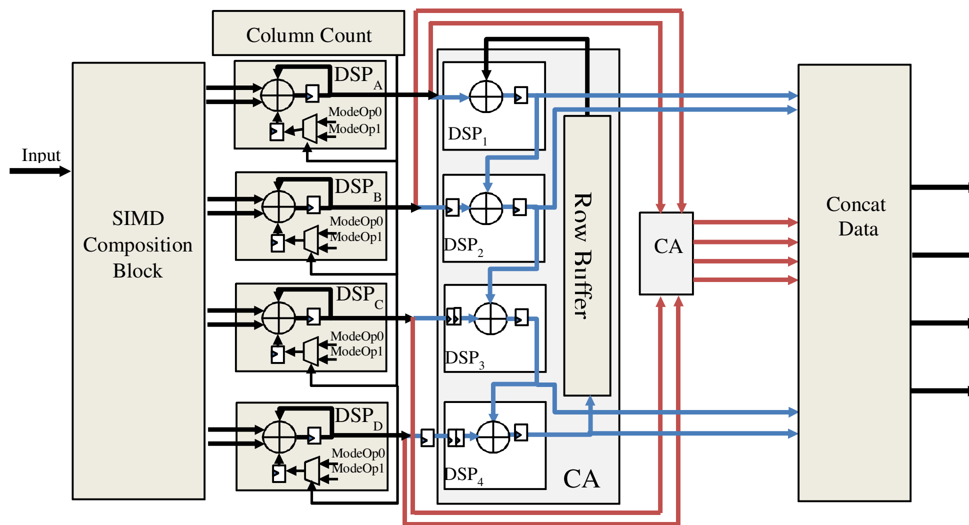


Figure 3.20 Architecture designed to implement the novel IIM computing scheme.

The example in Figure 3.21(a) allows analysing data dependencies between adjacent columns. The pixels *A*(*x*, *y*) and *A*(*x*, *y*+1) are transferred in parallel so that *RS*(*x*, *y*) and *RS*(*x*, *y*+1) are calculated at once. It is evident that *RS*(*x*, *y*+1) can be computed by adding *A*(*x*, *y*), *A*(*x*, *y*+1) and *PrevRS*(*x*, *y*+1), which is the value of *RS*(*x*, *y*+1) at the previous clock cycle. For example, as highlighted in blue, *RS*(*x*, *y*+1)=36 is produced at the fifth clock cycle by summing *A*(*x*,y)=7, *A*(*x*,*y*+1)=8 and *PrevRS*(*x*,*y*+1)=21. Similarly, *RS*(*x*, *y*) could be calculated by summing *A*(*x*, *y*) and *PrevRS*(*x*, *y*+1). As instance, the value *RS*(*x*, *y*)=28 results from the addition between *A*(*x*, *y*)=7 and *PrevRS*(*x*, *y*+1)=21. However, as pointed out in Figure 3.19, the SIMD operating mode does not allow the MSB part of *P* to be accumulated with the LSB part of *Ia* and *Ib*. This means that *PrevRS*(*x*, *y*+1) can not be used for calculating *RS*(*x*, *y*). Therefore, a different strategy has been adopted. As depicted in Figure 3. 21(b), *RS*(*x*, *y*) is computed by adding *A*(*x*, *y*), *PrevRS*(*x*, *y*) and *AD*(*x*, *y*+1), which is the value *A*(*x*, *y*+1) arrived two clock cycles before. Figure 3.21(b) also shows how the inputs are arranged within the *k*-bit words *Ia* and *Ib* to correctly perform the

required operations. It can be verified that, with this strategy, $RS(x, y)=28$ is computed adding $A(x, y)=7$, $AD(x,y+1)=6$ and $PrevRS(x, y)=15$.

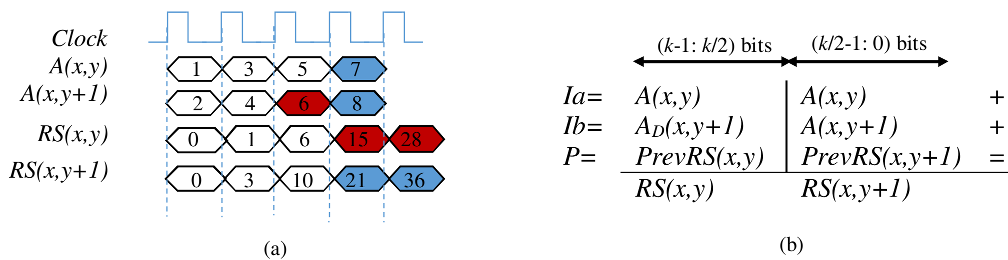


Figure 3.21 Adapting the SIMD dual-mode DSP to the proposed IIM computing scheme. (a) Analysis of the data dependency; (b) packing of the inputs.

3.5.3 Experimental results

The accelerator illustrated in Figure 3.20 has been embedded in a complete heterogeneous system realized on a Xilinx XC7Z020 SoC platform [34]. Such a design is depicted in Figure 3.22. It includes a Video Direct Memory Accesses (VDMA) IP core, configured for four channels. This IP core is provided of an AXI4-Lite interface, for communication with the ARM processor, an AXI4-Full interface for read/write operations within the external memory through the proper controller, and an AXI4-Stream interface, for data transfers to/from the custom hardware. Specifically, in the proposed design, the four-channel VDMA is used to parallel transfer four adjacent rows of the input image, from the external memory to the custom IIM circuit. To this purpose, the ARM processor configures the IP core through a software C-code, containing information about transfer modalities, address spaces to be accessed and number of bytes to transfer. In order to ensure that input pixels, which are stored within the consecutive locations of the external memory, can be transferred in raster order, avoiding unnecessary data reorganization as in [90], the

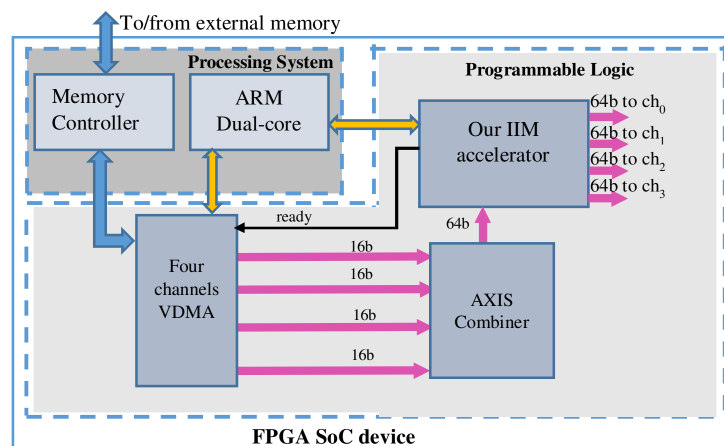


Figure 3.22 Embedded system accommodating the proposed IIM computing accelerator.

VDMA exploits a stride-based access strategy with stride equal to 4×*n*, as illustrated in Figure 3.23. Basically, starting from a specific address, which is set during the software configuration, each channel of the VDMA performs consecutively *m* read/write operations; then, it jumps a number of consecutive addresses equal to 4×*n* and repeats *m* read/write operations. Such a process ends when each channel has transferred in parallel *n*/4 rows. The four 16-bit raster ordered streams of pixels are sent to an AXIS-Combiner module that generates the synchronized 64-bit stream fed to the proposed IIM circuit. It is important underlining that, in addition to the stride strategy that enables the parallelism along the rows, each VDMA channel is made able to access two adjacent locations of the external memory, thus allowing two 8-bit column-adjacent pixels to be transferred at the same time. The four 64-bit outputs produced by our custom circuit are addressed to the four channels of the VDMA to be written into the external memory, thus leading again to a parallel transfer of four adjacent rows of the resulting IIM.

ADDR	DATA
Base+ 0×00	A(0,0)
Base+ 0×01	A(0,1)
Base+ 0×02	A(0,2)
Base+ 0×03	A(0,3)
Base+ 0×04	A(1,0)
Base+ 0×05	A(1,1)
Base+ 0×06	A(1,2)
Base+ 0×07	A(1,3)
Base+ 0×08	A(2,0)
Base+ 0×09	A(2,1)
Base+ 0×0A	A(2,2)
Base+ 0×0B	A(2,3)
Base+ 0×0C	A(3,0)
Base+ 0×0D	A(3,1)
Base+ 0×0E	A(3,2)
Base+ 0×0F	A(3,3)
Base+ 0×10	A(4,0)
Base+ 0×11	A(4,1)
Base+ 0×12	A(4,2)
Base+ 0×13	A(4,3)
Base+ 0×14	A(5,0)
Base+ 0×15	A(5,1)
Base+ 0×16	A(5,2)
Base+ 0×17	A(5,3)
Base+ 0×18	A(6,0)
Base+ 0×19	A(6,1)
Base+ 0×1A	A(6,2)
Base+ 0×1B	A(6,3)
Base+ 0×1C	A(7,0)
Base+ 0×1D	A(7,1)
Base+ 0×1E	A(7,2)
Base+ 0×1F	A(7,3)

Positions accessed by VDMA ch_0

Positions accessed by VDMA ch_1

Positions accessed by VDMA ch_2

Positions accessed by VDMA ch_3

Figure 3.23 Stride-mode strategy used to access the external memory for row-parallel IIM computing when *m*=4 and *n*=8.

Implementation results of the circuit illustrated in Figure 3.20 are reported in Table 3.4. Characterization results of all referenced competitors [88], [90] and [91] are available just for their implementations as standalone modules. Therefore, for a fair and direct comparison, also the proposed FPGA-based design has been characterized as a standalone module (i.e. without the contributions of the VDMA and AXIS-Combiner). Results

summarized in Table 3.4 show that, when 640×480 input frames are processed, the proposed IIM circuit achieves a frame rate 60% and 67.4% higher than [88] and [91], exploiting much less resources (logic and DSPs). Authors in [90] report an implementation of the strips-based approach for w=32. For this design, a frame rate of 5191 fps is achieved at a running frequency of 50MHz, which corresponds to about 9632 clock cycles. However, they do not take into account additional time required to re-organize 640×480 input pixels that, in the most favorable scenario would increase the number of clock cycles to 316832, leading to a frame rate of just 158 fps at 50MHz.

Table 3.4 Implementation results of the novel IIM circuit and state-of-the-art comparison.

	Novel IIM	[88]	[90]	[91]
FPGA	XC7Z020	Virtex 6	Cyclone IV	XC7Z020
LUTs	2113	7721	3458 LEs	NA
FFs	1047	19791		NA
Adders	12	8	95	484
Fmax [MHz]	150	147	50	100
Fps	3906.2	1915.7	5191	1272.2
Fps/Adder	325.5	239.4	54.6	2.6

In order to include also computing resources in this comparison, the speed per computing unit parameter, defined as fps/Adders, has been evaluated for all the designs in Table 3.4. In this perspective, the proposed accelerator exhibits the best trade-off between performances and resource occupancy, with a speed per computing unit ×1.36 and ×5.96 greater than [88] and [90], respectively. In comparison with [91], for VGA input images, we achieve a speed per computing unit ~124 times higher. Even more evident improvements can be reached for higher resolution. In fact, while the number of adders used in [91] is n+4, thus increasing with the image size, the hardware complexity of the proposed circuit does not vary with m and n.

Speed performances obtained for the embedded system in Figure 3.22 are reported in Table 3.5, where an estimation of the frame per seconds achievable by the competitors is also provided, assuming the same embedded platform XC7Z020 to guaranteeing comparable operating conditions. As mentioned in previous paragraphs, the DDR controller integrated within the XC7Z020 SoC sustains a maximum bandwidth equal to 33.3 Gb/s [82]. Obviously, this constraint differently affects the maximum running frequency achievable by each design implementation to support the adopted parallelism level. Results show that, except for the design presented in [91], the standalone accelerators could be integrated within a complete embedded system only decreasing their maximum operating frequency. This would be necessary to ensure that data are transferred to/from the external memory continuously, without requiring interruptions by the external memory

Table 3.5 Speed performances evaluation for a complete embedded system on the XC7Z020 SoC.

	Novel IIM	[88]	[90]	[91]
Fmax [MHz]	65	98	10	100
Fps	1692.7	1277	1038.2	1272.2

controller. Table 3.5 shows that the proposed accelerator is the least affected by the bandwidth limitation. And in fact, it allows achieving the highest frame rate of 1692.7fps for 640×480 image resolutions. Conversely, the design in [90] is the most affected, with a five times lower operating frequency, due to the high number of data (e.g. w=32) that is supposed to be processed at once. Concerning the power consumptions, the Xilinx Power Analysis Tool has been adopted to execute an estimation based on the switching activity of internal signals for a typical input pattern. Results show that a dynamic power of 7 mW is dissipated to process 640×480 frame resolutions at 150MHz, which is well below the 9 mW dissipated per frame, at the parity of resolution, by the accelerator proposed in [90] .

Finally, the performance of the proposed architecture has been also compared with a pure software implementation of the classic recursive approach [85]. A C-code running on the ARM dual-core processor has been used to evaluate the time required to complete the IIM computing task for several frame resolutions. The processor, running at 666.66MHz, takes 0.95 ms, 3.82 ms, 11.8 ms and 26.5 ms for the resolutions 320×240, 640×480, 1280×720 and 1920×1080, respectively, thus being, for the same input frame resolutions, about 10 times slower than the proposed embedded system running at 65MHz.

3.6 SUMMARY

In this chapter a novel stereo vision algorithm suitable for the implementation within real-time embedded visual IoT nodes has been presented. Its custom hardware architecture has been designed adopting heterogeneous FPGA-based SoC as the target platforms. Such a novel design has been exploited in the realization of a complete smart camera system, including both image sensor and processing unit. The latter has been realized taking into account several hardware/software partitions. For the hardware part, HLS- and VHDL-level descriptions have been conjunctly adopted. Quality tests, performed on several common benchmarks, have demonstrated that the proposed algorithm reaches an accuracy level similar or above those achieved by competitive methods, suitable for hardware implementation. It has also been demonstrated that the proposed method performs even better in outdoor scenes, such as those of interest for autonomous vehicle applications. When realized by using a parallelism level Pl=8, it requires ≈80%, ≈32%, ≈11% and ≈12%

of LUTs, FFs, DSPs and BRAMs available on a Zynq-7000 XC7Z045 device, respectively, and it allows a frame rate up to 101 640×480 frames per second to be sustained.

This chapter also introduces a novel parallel scheme for computation of integral images, which have been largely exploited in several previous works for features extraction (i.e. in face detection applications [85]) as well as to efficiently perform stereo matching [18], [94]-[96]. The new approach has been hardware implemented on a Zynq-7000 XC7Z020 SoC, designing an energy-efficient accelerator that requires just 7mW per 640×480-sized frame at 150MHz clock frequency. When accommodated within a complete heterogeneous embedded system, the novel design achieves a frame rate of 1692.7fps at 65MHz, outperforming both state-of-the-art FPGA-based competitors and the pure software-based algorithm implementation.

4 EFFICIENT CCA APPROACHES FOR FPGA-BASED EMBEDDED SYSTEMS

4.1 BACKGROUND

One of the main advantages introduced by the edge computing paradigm in IoT applications is surely the possibility of reducing the amount of data transmitted from sensor nodes to cloud, with significant improvements in terms power consumption, transfer time and privacy. The aforementioned benefits are even more impressive when visual IoT is the target field. Indeed, as discussed in Chapter 2, images and videos have a rich information content, thus continuous streaming of raw frames causes battery discharging in a very short time [25]. In this context, the in-node approach allows sensed data to be locally analyzed to extract features of interest by means of embedded computing capabilities, enabling an energy-efficient transmission to the cloud.

As illustrated in Figure 4.1, Connected Component Analysis (CCA) is a technique used to extract connected components in a binary image, and their features of interest, such as area, bounding boxes, center of gravity etc., which will be then processed depending on the specific application. To this purpose, CCA combines a labeling process [96], responsible for distinguishing different objects in the source binary image by assigning a unique label to all pixels that refer to the same object, with a features extractor that transforms the labeled image into synthetic data for the next high-level computation.

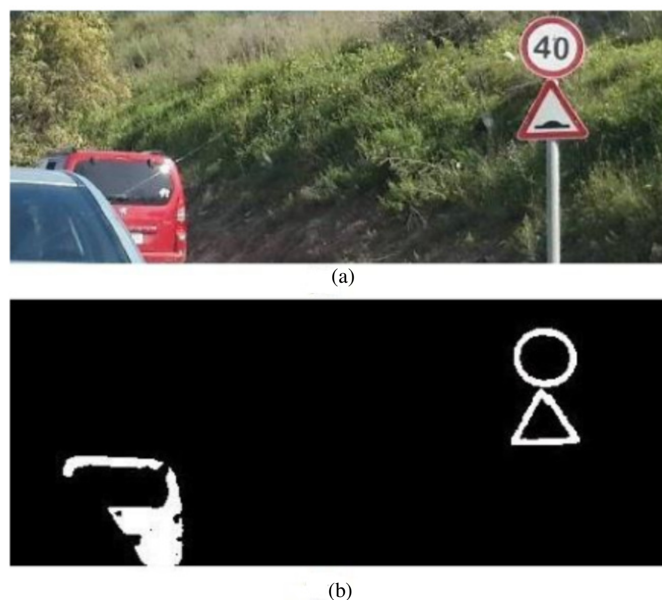
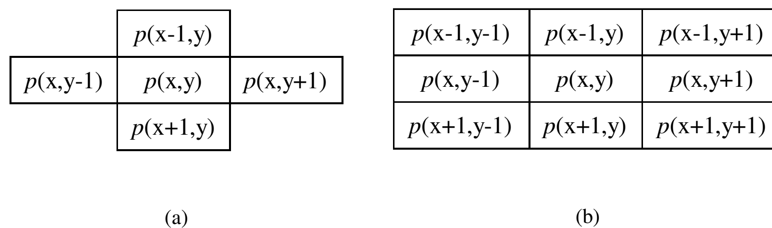


Figure 4.1 An example of detecting multiple objects within a real scene.

The labeling algorithm scans the input image in raster order and processes each pixel $p(x, y)$ with its neighborhood. Typical neighbors consist of 4-connected or 8-connected pixels [98], as shown in Figure 4.2. Considering that the input image is streamed in raster order, it is easy to understand that, when the pixel $p(x, y)$ is processed, only the pixels located within previous rows and columns are actually available. During the scan, a new label is assigned to each newly encountered unconnected foreground pixel $p(x, y)$. In the presence of only one neighbor labeled pixel, such a label is connected also to $p(x, y)$, whereas, if two neighbor pixels are labeled differently, a collision occurs.



(a) (b)

Figure 4.2 Pixel connectivity for a four-connected (a) and an eight-connected (b) set.

An example is reported in Figure 4.3 to clarify how the labeling process works when 8-connected neighbors are used. The case (a) occurs when the neighborhood of the current foreground pixel contains only background pixels: in this case, a new label is assigned to $p(x, y)$. The case (b) highlights that, when an already labeled pixel is part of the current processed neighborhood, $p(x, y)$ inherits its label. Finally, the case (c) identifies a critical event, known as collision, which occurs when the current neighborhood contains two differently labeled pixels. This means that all the pixels previously marked with labels 2 and 3, and so thought to belong to separate objects, are instead sub-portions of the same object. Therefore, all those pixels must be assigned to the same label.

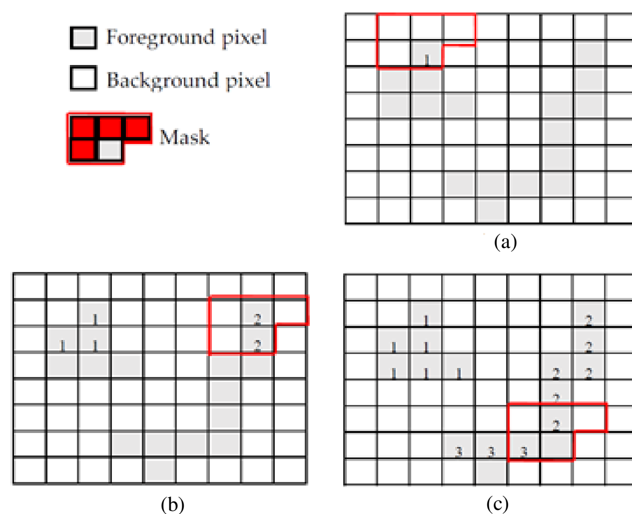


Figure 4.3 Example of labeling process for (a) a new label assignment; (b) an inherited label; and (c) a collision.

Resolving the chains of label collisions is the main challenge in all labeling algorithms. Usually, one of the following strategies is adopted [99]:

- Label-equivalence: in a first image scan, provisional labels are assigned to each pixel, following the criteria described in the above example. All label collisions are stored. After the first scan is completed, label collisions are resolved by finding a representative label for each group of equivalent labels for which collisions were detected. Then, each pixel of the input image can be correctly re-labeled.
- Label-propagation: the first unlabeled foreground pixel is found and a new label is assigned. Then, such label is propagated to all foreground pixels connected to it. This technique requires multiple scans of the input images and/or accessing the pixels in an irregular order depending on the shapes of connected components.

It is then clear that a pure software implementation of the labeling process is often insufficient for the strict performance and energy requirements of IoT nodes. To address this issue, in the last few years, designers have directed their efforts towards the design of hardware modules for use in specialized image processing architectures. Unfortunately, due to their irregular access to image pixels, label-propagation methods are not suitable for pipelined and hardware implementations. Conversely, label-equivalence algorithms are very time-consuming, due to the second scan required to correct collided labels with the corresponding equivalences.

4.2 RELATED WORKS

As discussed in the previous paragraph, in the presence of complex geometric shapes, labeling can require sequential operations to properly manage collisions. However, in some cases [44][100][101], only features of interest extracted during the second step of the CCA process are actually subsequently elaborated. In this perspective, the intermediate labeled output image is not strictly necessary [102], thus multiple scans of the input frame can be avoided and one-scan CCA algorithms can be efficiently exploited to improve performances of the application. In the following, a review of both two-scan and one-scan CCA hardware-oriented algorithms is provided.

4.2.1 Two-scan CCA methods

Traditional two-scan CCA performs a first forward scan in order to assign provisional labels to all foreground pixels and stores the collisions information in a table. The latter is

typically a look-up-table in which the label and its root are kept. After a phase during which equivalences between labels are solved within the table, the second forward scan occurs to solve the collisions by reading the solved equivalences. Besides the need for two full image scans in such a scenario, label equivalences may be stored multiple times in the table, thus affecting the achievable overall performance. Recently, to speed-up the hardware implementation of the classic two-scan algorithms, several strategies have been investigated [103]-[108], each having its own strengths and weakness.

The architecture presented in [103] exhibits a relatively low latency, but it was specifically designed and optimized to work only on a particular class of objects, called *k*-concave. Moreover, it uses two buffers to store a sub-image of the input frame on which a three-step process is performed to provide labels. The design proposed in [104] minimizes the amount of memory required to store equivalent labels by working on a run-length encoding representation of the input binary image. However, this approach also requires additional pre- and post-processing operations, which adversely affect the latency [105]. In order to introduce a high parallelism level, thus increasing the achievable speed performance, the strategy proposed in [106] partitions the input image into equally sized patches. Then, a local provisional label assignment is applied in parallel to each patch during the first scan, and a local set of equivalences is generated. Finally, a fusion step occurs to merge in parallel all the sets of local equivalences. Unfortunately, as a drawback, the architecture proposed in [106] requires large memory resources. As an example, when 720p frames must be labeled, 9846 kbits are required, thus limiting the implementation of further processing in the case of an embedded system. It is worth noting that the mask used to define the neighborhood of the generic foreground pixel to be labeled can influence the achievable overall performance. In fact, it determines the number of accesses to the equivalence table to write/read the collisions. For instance, in [107]-[108] an extended neighborhood composed by five 2×2 blocks of pixels is used. The label assignment is then performed through a decision tree. For this particular mask, the pixels in the neighborhood are accessed multiple times during the scan phase since the same strategy is iterated for each foreground pixel in the input image.

Obviously, to complete the CCA process, whatever labeling algorithm has been adopted, the additional computational step is required to extract the features of interest of the objects in the input image, such as boundaries, position, center of gravity, etc.

4.2.2 One-scan CCA methods

One-scan approaches work as follows: the input image is scan in raster-order and the labeling process is applied as described in Figure 4.3; when a collision is identified,

corresponding labels equivalences are stored to calculate the features for the recognized connected components afterwards. However, complex cases, as illustrated in Figure 4.4, can occur. In such a case the collision chain 4 → 3→ 2 → 1 takes place, which means that the label 4 is equivalent to 3, which is equivalent to 2 that in turn is equivalent to 1. It is then clear that, when a similar condition occurs, recognizing that all the foreground pixels belong to the same connected component could require several steps.

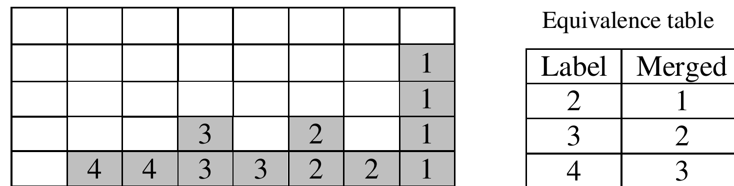


Figure 4.4 Example of critical path containing a chain of collisions with length 3.

In order to resolve the chain, [102] proposes a backward merging technique, in which collisions encountered in each row of the binary image are stored into a stack in the reverse order with which they occur. At the end of each scan line, these collisions are read from the stack to be processed in the Equivalence Table by substituting equivalent labels with their representative smallest label. It is worth noting that resolution of collisions based on [102] requires three clock cycles per equivalence, which represents a bottleneck in streamed image processing. Authors in [109] try to optimize the conventional CCA algorithm by recycling labels after every row in order to reduce memory requirement significantly. However, both [102] and [109] do not take into account specific patterns that result in incorrect labeling. Such conditions, which are due to the so-called *stale* labels, require further processing. A *stale* label is characterized for not yielding its representative label in the Equivalence Table. A typical example is illustrated in Figure 4.4. When the pixel *P* has to be processed, label 3 has been already looked up to 2 in the previous row. This in turn has been merged with 1 earlier in the current row. In this case, all pixels labeled with 3 are translated to 2, which leads to an incorrect assignment for *P*.

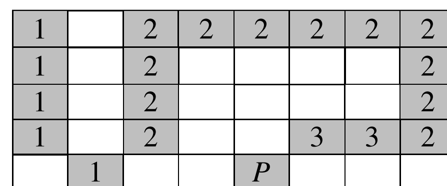


Figure 4.5 Example of a *stale* label.

A way to solve the above described issue has been introduced in [97], where two flags are used to detect and manage the *stale* labels. Based on several optimization strategies for label recycling and translation, the hardware architecture proposed in [97] allows memory

resources to be significantly reduced. Nevertheless, the additional time required to resolve label equivalences after each scan line leads again to a limited throughput with up to 20% overhead for the worst-case image.

In order to eliminate this throughput limitation, several implementations have been proposed [110]-[113]. The single-pass CCA algorithm presented in [110] uses a run-length encoding technique for labeling connected components while representing equivalences with linked lists. In this way, resolution of equivalences is simplified as it involves only pointer redirection operations and removes the chaining problem. A run-length based CCA approach was also adopted in [111] for reducing the number of labels to be managed. However, in that case authors simultaneously process two adjacent rows of the input image to generate runs for the current row and indexes of equivalent runs with respect to the previous row, introducing further memory resources. Malik *et al.* [112] proposed a labeling pipeline architecture to remove the blanking period for label merging by dividing the operations into two stages. The former scans pixels, assigns labels, and collects provisional features. The latter merges the labels based on the information from the first stage. Although this architecture is able to hide the variable merging period with the second stage, which depends on the complexity of connected components, it also requires doubling the memory occupancy with respect to [102]. Conversely, strategy proposed in [113] allows performing the label merging on the fly by means of an optimized row buffer, designed in the form of a shift-register.

A hybrid hardware-software implementation of a single-scan CCA algorithm has been presented in [114]. In such an implementation, a DRAM is used to store the provisional labeled image, whereas block RAM buffers are used to temporarily store label-connection table and feature data table. The algorithm is executed over three pipelined stages. The above approach has been implemented on a Zynq AP-SoC [34] containing a dual core ARM processor. In order to make the overall architecture more flexible and able to extract new more features, but limiting the overall performances in practical environments, the label-connection and the feature data tables are updated by software routines running on the ARM processor.

Hardware implementations of parallel single-pass CCA have been proposed in [105] and [115]. There, the input binary image is divided into vertical slices that are parallel processed. Despite to the high performances achieved, this parallel solution CCA requires massive resources for high degree of parallelism.

Despite of the clear advantages introduced by one-scan CCA techniques compared to the two-scan counterpart [116], there exits some applications [117]-[119] for which having a correctly labeled binary image is crucial. For this reason, the efforts of this research have been also addressed to the design of an efficient hardware solution able to

mitigate effects of the second scan, required by traditional labeling approaches, on performances and power consumptions.

4.3 THE PROPOSED EMBEDDED SYSTEM FOR ONE-SCAN CCA WITH COMPLETE LABELING

The main idea exploited by the proposed architecture [19]-[20], targeted for heterogeneous FPGA-based SoCs, is to overlap portions of the CCA computations with unavoidable control activities run by the processor cores. Figure 4.6 shows the case in which CCA and the subsequent computation on the labeled frame are implemented by custom modules within the PL together with two Direct Memory Access (DMA) cores that are responsible for the data transfers to/from the external DDR memory. Devices needed to capture the source images from a camera, to perform the binary segmentation, and to store the resulting binary image into the external DDR memory are omitted for clarity.

The activities of the two DMA cores are managed by software routines executed on the PS. The typical running of such a system takes place within the following steps: **i)** the DMA_0 is configured to read raw image data from the DDR memory and to stream it to the CCA module; **ii)** the output data stream produced by the CCA module is written to the DDR memory again by means of DMA_0 (if a conventional two-scan CCA approach is used, to obtain a labeled frame, this step should be repeated twice); **iii)** the DMA_1 is configured by the software routine running on the PS (this phase lasts for several microseconds); **iv)** finally, the DMA_1 reads the labeled frame from the external memory and streams it towards the subsequent processing module.

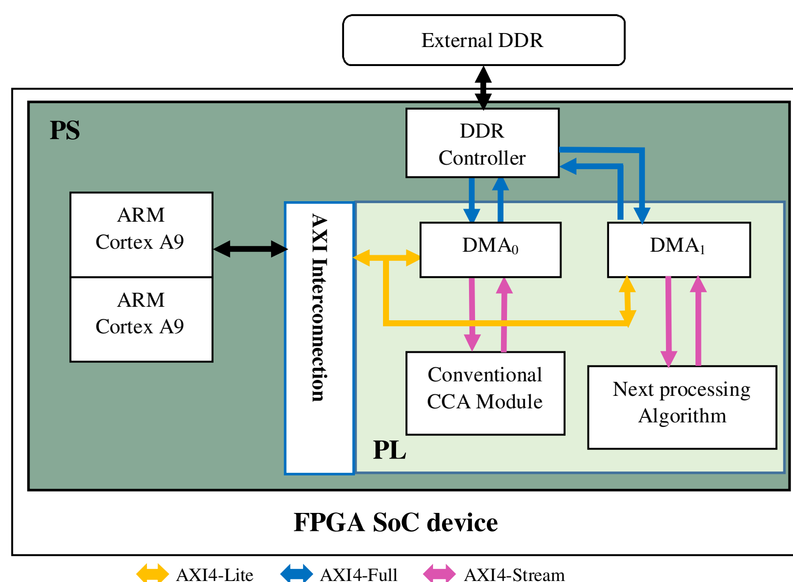


Figure 4.6 Heterogeneous embedded system including a conventional CCA module.

However, if the module performing the next processing has to scan again the labeled image provided by the CCA block, the PS must configure the DMA_1 core and, in the meantime, the label collisions can be resolved. The overlap of the collisions resolution with the configuration actions of the PS avoids multiple scans of the input image and any other step limiting the frame rate of the whole system.

4.3.1 System architecture

Based on the above considerations, the embedded system depicted in Figure 4.7 has been designed. Even though the top-level environment seems the same as that described in Figure 4.6, it can be easily appreciated that the data-flow is significantly different. As in the traditional scheme of Figure 4.6, DMA_0 is instructed to transfer input binary pixels to the novel CCA circuit in the streaming fashion. The provisionally labeled pixels provided by the CCA are then streamed towards DMA_0 and finally stored into the DDR memory. Provisional labels are also shared with the *Features extractor* module, which processes temporary features as labels are outputted by the *Provisional Labeling* block. When the last pixel within the current frame is processed, the *2D Table*, which stores information about previously assigned provisional labels and collisions, is updated and the *Translator LUT* is

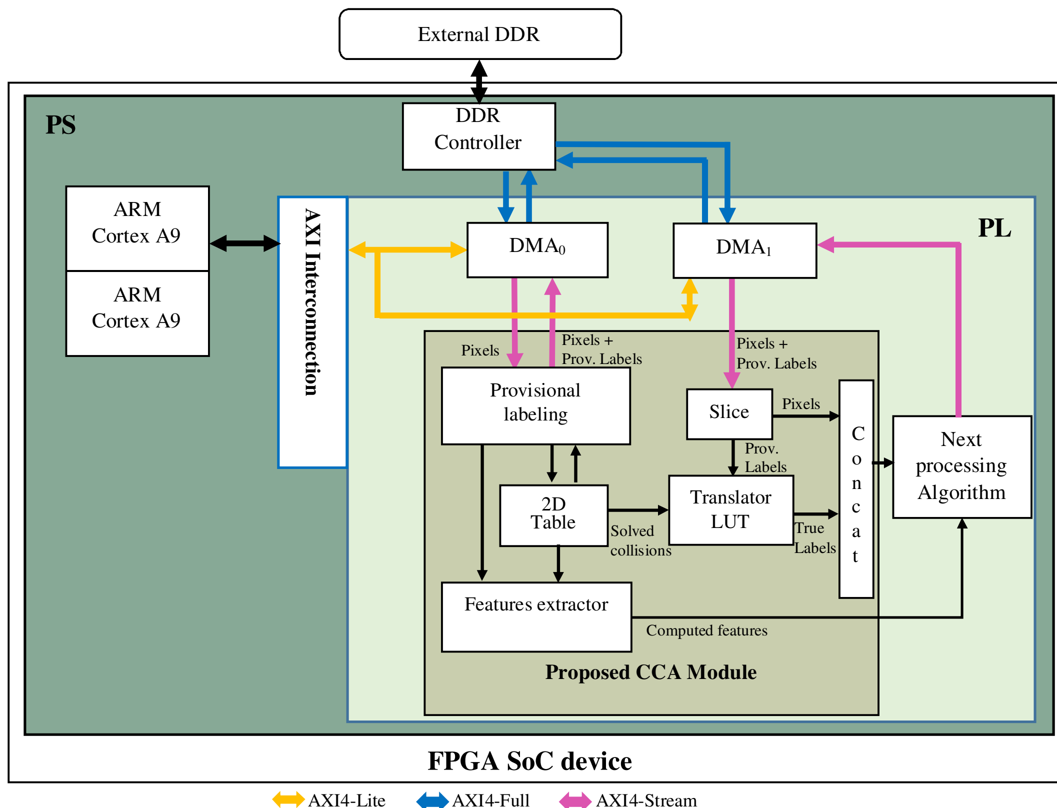


Figure 4.7 Heterogeneous embedded system including the proposed CCA module.

written to maintain only solved collisions. The latter are also used by the *Features extractor* in order to properly update the provisional features, which are transferred to the generic *Next Processing Algorithm* module. In the meantime, the DMA_0 releases the ARM processor to software programming the DMA_1 transaction. DMA_1 streams out provisionally labeled pixels to the *Slice* module that is used to partition data received within the pixel value and its provisional label. The latter is translated on-the-fly to a true label that is then concatenated to the original pixel. The truly labeled frame is then inputted to the next processing module without waiting for the canonical two frames transfer time. This has been made possible by overlapping the collisions resolution phase with the execution of the software code that programs the activity of DMA_1. It is also worth noting that, in contrast to the traditional one-scan approaches presented in [114] and [116], the proposed *Features extractor* is purposely designed to manage solved collision chains rather than partial information, avoiding multiple accesses to the memory resources that store the features. In the following, more details about sub-blocks composing the proposed CCA module will be provided.

4.3.2 Custom modules

The *Provisional Labeling* circuit illustrated in Figure 4.8 is made able to process a modified version of the standard 4-connected neighborhood that includes also the pixel at position (x-1, y+1). As verified through several tests performed in Matlab, the 4-connected neighborhood modified in this way allows the number of collisions to be reduced, thus accelerating the collision resolution phase. Clearly, this choice also impacts on the on-chip memory occupancy, because the modified 4-connected neighborhood reduces the possibility that a foreground pixel, being wrongly identified as unconnected, may be assigned a new label.

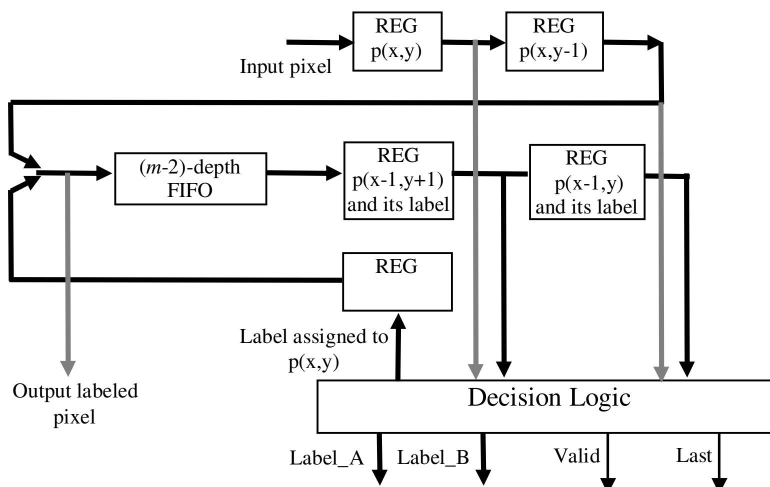


Figure 4.8 Design of the Provisional Labeling circuit proposed in this work.

The module receives the binary input pixels in a stream fashion, forms the appropriate neighborhood around the current foreground pixel $p(x, y)$, assigns to it a provisional label and finally merges the pixel and its label information on the output stream. The Decision Logic performs the following operations:

- If the neighborhood of the current pixel $p(x, y)$ contains only background pixels, a new label is assigned to it. New labels are generated in ascending order.
- If the neighborhood contains only a labeled foreground pixel, $p(x, y)$ inherits the label associated with such pixel.
- If the neighborhood contains two foreground pixels associated to different labels, that is a collision occurs, the smallest label is assigned to $p(x, y)$. It is worth noting that the chosen neighborhood cannot contain three colliding labels.

The (m-2)-depth FIFO stage visible in Figure 4.8 is used to correctly form the neighborhood of each input pixel. The *Provisional Labeling* module provides the output signals *Label_A* and *Label_B*. They assume the same value when a new label is assigned to the current pixel $p(x, y)$. Conversely, different values of *Label_A* and *Label_B* flag that they are two colliding labels. In this case, the label assigned to $p(x, y)$ is the minimum between *Label_A* and *Label_B*. The furnished labels are validated by the *Valid* signal. When the input image scan is completed, the *Last* signal is asserted and the collision resolution phase is started. It is worth noting that, thanks to the adopted structure, after the latency of only two clock cycles is elapsed, the provisional labeling operation provides a novel output every clock cycle.

During the first raster-scan, the *2D Table* is written as provisional labels are assigned and collisions are identified. Basically, the proposed *2D Table* can be seen as $N_L \times N_L$ bit-positions, with N_L being the number of usable labels. When a foreground pixel is found, a label is assigned and two conditions can occur on the *Label_A* and *Label_B* outputs produced by the *Provisional Labeling* circuit:

- *Label_A=Label_B;* in this case the element located at (*Label_A, Label_A*) of the *2D Table* is set to 1.
- *Label_A≠Label_B*, which identifies the collision between two labels in the current neighborhood; in this case both positions (*Label_A, Label_B*) and (*Label_b, Label_A*) of the table are asserted.

The circuit implementing the *2D Table* is depicted in Figure 4.9. It can be seen that it is implemented as a binary register file composed by $N_L \times N_L$ D-flip-flops. An auxiliary circuitry is also used to perform the required writing and updating actions. The *Writing Controller* receives the signals *Label_A, Label_B* and *Valid* from the provisional labeling circuit. It performs the storing actions as above described, generating proper row and

columns indexes to access the single flip-flop. Then, when the *Last* signal is received, the update phase is started to resolve all collisions chains.

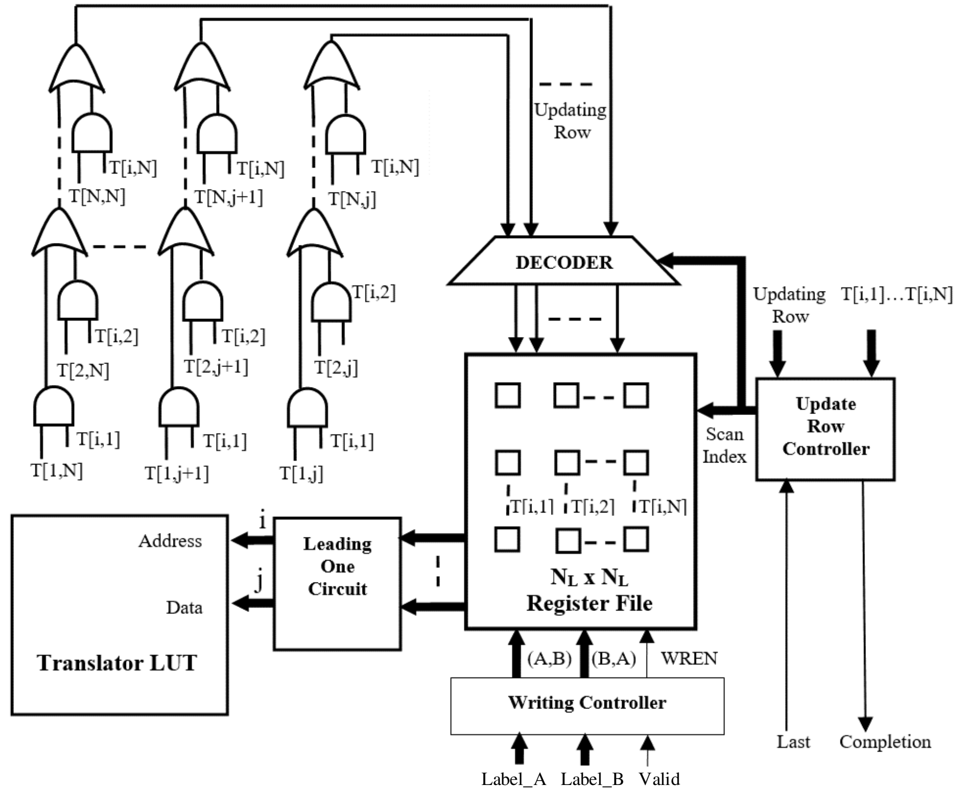


Figure 4.9 Architecture of the 2D binary table with updating capability.

The *Update Row Controller* scans the register file row-by-row and establishes when the current row *i* is completely updated by comparing it with the *Updating Row* bus produced by the AND-OR logic circuit also depicted in Figure 4.9. Practically, the current value *T[i,j]* enables the transfer of the *j*-th row to the flip-flops within the *i*-th row of the register file. Each row *i* of the table is provided with a *Leading-one* module, establishes the lowest column index *j* corresponding to the element *T[i,j]* set to 1. When the table update is completed, N_L column indexes, each corresponding to a specific row, are provided. This information represents the criterion with which provisional labels have to be translated into *True Labels* for the processed frame. Therefore, the above N_L column indexes are stored in a Look-Up-Table (*Translator LUT*) realized by means of $log_2(N_L)$ 1-bit SRAM-blocks. The *Translator LUT* is written in synchronous mode using the provisional label as the generic memory address and the true label as the data to store. At the end of the writing phase, the *Completion* signal is asserted and the *Translator LUT* can be easily read in asynchronous mode by inputting a label to be translated on the address bus to receive the true label as the output data.

Finally, the *Features extractor* module and its interaction with the *Provisional Labeling* circuit are illustrated in Figure 4.10. Although the specific implementation refers to the

extraction of bounding boxes coordinates, with the same principle other features can be extracted by connected components within the input image. The *Features extractor* module stores features information in 4 dual-port memory banks realized by using fabric LUTs. Such memory banks are addressed by using the current label to be processed. During the provisional labeling operations, our *Features extractor* module receives, for each foreground pixel, the signals *Valid*, *New_Label*, *Label_A* and *Label_B*, with the *Row* and *Column* indexes that identify the coordinates of the pixel. In this step, the *Completion* signal from the register file is driven low, and the *Mux_Sel* generated by the *Control Unit* enables the transmission of the minimum bewteen *Label_A* and *Label_B* to the synchronous *WRADDR* and the asynchronous *RADDR* inputs of each LUT. When a new label is assigned by the *Provisional Labeling* circuit, the WE inputs are asserted and the current *Row* and *Column* indexes are stored. In such a condition, no reading action is performed. Otherwise, if a valid label has been assigned and the signal *New_Label* is low, the location *min*(*Label_A*, *Label_B*) of each memory bank is read; the output SPO is compared with the current *Row*/*Column* index; and the input WE is eventually asserted for updating the horizontal/vertical coordinates of the object identified by such a label.

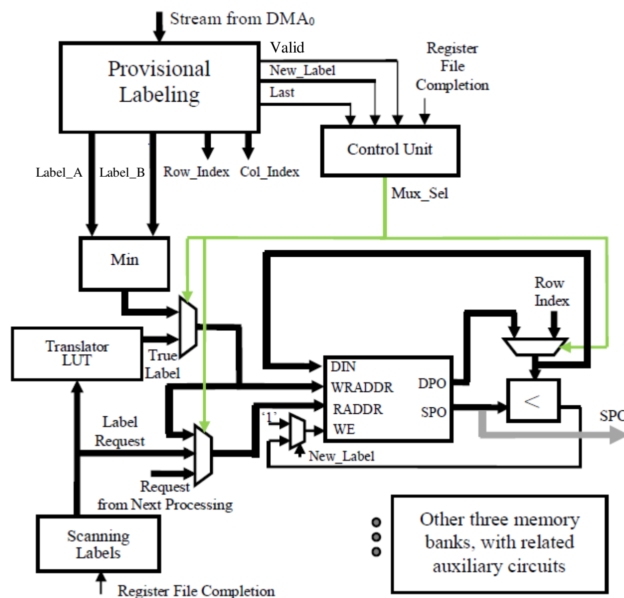


Figure 4.10 Circuit designed for extracting coordinates of bounding boxes for each connected component within the input image.

The circuit above described requires one clock cycle to store the provisional coordinates for each foreground pixel. When the *Last* signal is received from the *Provisional Labeling* block, the module ends the first phase. Surely, at this point, no collision has been resolved yet, and the four memory banks could contain false information about the bounding boxes.

The second step starts when the *Completion* signal is generated by the *2D Table*, which means that the *Feature extractor* module can access the *Translator LUT* in order to correct

the provisional coordinates based on the true label. In order to perform this operation, the *Scanning Labels* block scans the assigned labels and feeds the *Translator LUT* that outputs the corresponding *True Label* for each *Label Request*. Such data are now transmitted to the *WRADDR* and *RADDR* inputs of the memory banks, respectively. Whenever *Label Request* and *True Label* are different, a collision is identified, thus two parallel reading operations are performed and the SPO and DPO outputs are compared in order to verify if an update is required at the *True Label* address. Finally, the *Next Processing* module can access the fully labeled image and extracted features without waiting for a second scan of the image frame. This is a very nice property that characterizes our design with respect to all other known solutions.

4.4 RESULTS

4.4.1 Implementation

The above described system architecture has been implemented in both the ZedBoard Development Board and the ZC706 Evaluation Board. The former contains the Xilinx Zynq-7000 XC7Z020-CLG484 SoC, based on Artix-7 configurable cells, whereas the latter contains the XC7Z045FFG900-2 version, based on Kintex-7 cells. Custom circuits have been implemented for 640×480 frame resolution, with N_L spanning from 32 to 128 that well suite several applications, like the Traffic Sign Recognition (TSR).

Table 4.1 summarizes the characteristics of the proposed CCA module and reports the results obtained from the comparison with state-of-the-art competitors. It can be easily observed that, as expected, the proposed architecture achieves frame rates significantly higher than [103]-[104], [114], [116]. These advantages are well above the obvious speedup due to the more advanced technology used here, as clearly demonstrated by the technological-independent parameter Cycles/pixel. It is also worth noting that the hardware resources requirement of the system demonstrated in [114] is related only to the provisional labeling, since the label collisions are managed by a software routine that uses further external memory resources that are not accounted in Table 4.1. Moreover, instead of the labeled frames it provides just several features that suffice several applications, but are not fully functional to TSR. Therefore, the design proposed in [114] is directly comparable just with our *Provisional Labeling* circuit, also characterized in Table 4.1.

Table 4.1 Implementation results of the proposed embedded system for one-scan CCA with complete labeling.

	Device	F [MHz]	LUT	FF	BRAM [bit]	Mpps	Cycles/ pixel
Complete CCA							
[103]	Stratix	72	10.7k		400k	57.6	1.19
[104]	Virtex IV	49.7	649	641	1142k	23.5	2.011
[116]	Virtex II	27	7589	936	272k	12.5	2.054
New N_L=32	XC7Z045	225	4551	1882	0	212	1.011
New N_L=64	XC7Z045	225	18144	5005	0	212	1.011
New N_L=128	XC7Z045	225	76886	17444	0	212	1.011
Provisional Labeling only							
[114]	XC7Z020	124.2	452	608	90k	117	1.011
New	XC7Z020	142.8	404	770	0	134.6	1.011

To measure energy consumption, the Xilinx Power Analysis Tool has been used, considering the switching activity of the internal signals when the theoretical worst-case image is processed. In such a situation, the dynamic power consumed by the proposed CCA circuit when managing N_L=64 labels and running at 225MHz on the XC7Z045 chip is 102 mW. Unfortunately, only the reference [112] cited in Table 4.1 reports information about the power consumption. In particular, it consumes 32 mW of dynamic power to process 86 frames per second. This means that, while the design presented in [116] dissipates on average 372μJ per frame to extract just the center of gravity from the connected components, the novel implementation proposed here consumes on average just 138.4μJ per frame to perform the complete labeling and to extract the bounding boxes.

4.4.2 Experiments for Traffic Sign Recognition applications

A representative example of performances demanding application which requires CCA operation is the Automated Traffic Sign Recognition (TSR): one of the most basic functions requested to the future Advanced Driver-Assistance Systems (ADASs) for autonomous driving. Traffic signs provide significant visual information, such as speed limitations, drive directions, presence of obstacles, roadway access, and many others. A TSR system should be able to perform the sign detection and the subsequent classification/recognition as fast as possible, by elaborating images captured by cameras mounted on the vehicle. In the detection phase, captured images are pre-processed, segmented on the basis of particular attributes, such as shapes and/or colors, and finally labeled. In the classification/recognition phase, several feature extraction methods and machine learning algorithms are adopted to determine the membership classes of the traffic signs previously detected.

In order to test the capacity of the proposed CCA accelerator within such an application field, several images from the German Traffic Signs Recognition Benchmarks (GTSRB) [119] have been used as inputs. Some samples are reported in Figure 4.11. Due to their different resolutions, test images have been pre-elaborated in the Matlab environment to be randomly placed within 640×480 frames and then segmented. Performed tests allowed simulating the typical scenario in TSR environments, where one or more signals detected in a frame are distinguished by labeling before sending the labeled frame to the classification engine. By extensively using the *Internal Logic Analyzer* (ILA), it has been verified that the time required in the worst case to write the *Translator* LUT, being less than 5μs, is practically hidden by the DMA_1 configuration process. Results obtained by these tests are summarized in Table 4.2, which also reports the number of objects in the frame, the number of assigned provisional labels and the number of clock cycles needed to update the *2D Table*. It is important to underline that the time required for resolving labels collisions chains is directly related neither to the number of objects in the image nor to the number of labels provisionally assigned (or the number of collisions found). Instead, the number of clock cycles needed to update the *2D Table* exclusively depends on the length of the collisions chains found in the frame. From this point of view, the worst scenario occurs when: **i)** only one object is in the frame; **ii)** all the available labels are provisionally assigned; and **iii)** each label collides with only one of the others assigned labels. Although this is an absolutely unrealistic situation, the *2D Table* has been artificially forced to assume

Figure 4.11 Samples of benchmarks images from the GSTRB [119] database.

Table 4.2 Number of clock cycles required by the update step for several GSTRB [119] images.

Image	#Objects	#Prov.Labels	#Clock Cycles for updating the 2D Table
00002/00009_00027	1	19	264
00017/00024_00029	2	17	190
00040/00003_00029	2	23	178
00024/00004_00027	1	9	192
00013/00016_00028	1	12	156
00035/00039_00028	1	14	287
00016/00000_00025	4	23	163
00018/00004_00023	3	8	197
00024/00006_00027	8	20	137
00015/00001_00029	1	12	149

such an unrealistic content. In this case, assuming N_L=64, the total time required to write the *Translator* LUT is ≈18µs. As a consequence, the *Completion* signal outputted by the *2D Table* would delay the start of the DMA_1 transfer by ≈3µs, thus making the maximum sustainable frame rate equal to 723fps, instead of 724fps reachable when the resolution time is totally overlapped with the software code execution.

4.5 THE NOVEL LOW-COST CCA IMPLEMENTATION

As discussed in the subparagraph 4.2.2, one-scan approaches allow features of interest to be extracted by the provisionally labeled image. In this case, the second scan, needed to substitute false labels with their equivalent ones, can be avoided. It is then clear that one-scan CCA introduces benefits for both performances and energy dissipations. However, as stated in the review presented in 4.2.2, state-of-the-art hardware implementations of one-scan CCA are often inefficient. Some prior works [97][109][116] are not able to achieve a unitary throughput (cycles/pixel), limiting or even discarding the effect of removing the second scan. Some others [97][110][113][114][116] require a negligible amount of on-chip memory resources, which hinder their implementation within energy and resources constrained embedded visual IoT nodes.

In the following, the novel CCA approach [21], proposed to overcome the aforementioned problematic, is described. It is worth noting that both algorithmic- and hardware-level optimizations are here introduced with respect to the state-of-the-art hardware oriented single-scan methods. Such improvements, which make the novel CCA accelerator suitable to be easily integrated within heterogeneous visual embedded systems based on energy-efficient and resources-constrained platforms, have been verified in the very multifaceted aerospace context, and more specifically for applications related to the relative navigation of satellites.

4.5.1 Algorithmic-level optimizations

The one-scan solution proposed in [21] is based on runtime processing input binary pixels, by simultaneously assigning provisional labels, managing equivalent labels, and updating features data. The whole process makes use of auxiliary tables to take into account equivalences that occurred during the scan to store temporary features information. Such tables are named *Translator LUT* (TL) and *Features Table* (FT), and their size depends on the maximum number of labels the architecture is allowed to assign, i.e. N_L. Without loss of generality, in the following, extracting the area of connected components within the input image is referred as an example of applications, but the same approach can be used

to also extract other features, such as the bounding box, the center of gravity, etc. Clearly, in these cases, the management of the auxiliary tables changes accordingly.

The pseudo-code reported in Figure 4.12 describes the proposed labeling algorithm. It can be seen that each background pixel is zero-labeled, whereas, when a foreground pixel *P* is inputted, its 4-connected neighborhood is evaluated. Then, if *P* is surrounded by background pixels, it receives a new generated label *Lj* (with j=1, …, N_L−1) and the *Translator LUT* (TL) is updated so that TL(*Lj*) = *Lj*. Conversely, if the current neighborhood contains just one foreground pixel already labeled, such a label is used as the address to access the *Translator LUT*, and the translated label is assigned to *P*. In the critical situation, the neighborhood contains two foreground pixels, associated to two colliding labels *Lx* and *Ly*. In such a case, the minimum between TL(*Lx*) and TL(*Ly*) is assigned to *P* and, to update the *Translator LUT* with the newly discovered equivalence, TL(max[TL(*Lx*), TL(*Ly*)]) is set to min[TL(*Lx*), TL(*Ly*)]. This ensures that: **i)** the correct translated label is always propagated in the labeled image, **ii)** no long chains are generated

```
Input: the n×m binary image Im
Outputs: the n×m provisionally labeled image LI
       the N_L-element translator LUT TL
TL(0)=0;
New_label=1;
for (i=0; i<n; i++)
  for (j=0; j<m;j++)
    P=Im(i,j);
    //Form the neighborhood of P
    if (i=0) //P belongs to the first row
      Lx=0;
    else
      Lx=LI(i-1,j);
    if (j=0) //P belongs to the first column
      Ly=0;
    else
      Ly=LI(i,j-1);
    if(P=0) //P is a background pixel
      Lj=0;
    else //Check the neighborhood of P
      if (Im(i-1,j)=0 && Im(i,j-1)=0)
        Lj=New_label;
        TL(Lj)=Lj;
        New_label=New_label+1;
      elsif(Im(i-1,j)=1 && Im(i,j-1)=0)
        Lj=TL(Lx);
      elsif(Im(i-1,j)=0 && Im(i,j-1)=1)
        Lj=TL(Ly);
      else
        if(Lx= Ly)
          Lj=TL(Lx);
        else //Store the new equivalence in TL
          Lj=min(TL(Lx), TL(Ly));
          TL(max(TL(Lx), TL(Ly)))=Lj;
        end
      end
    end
    LI(i,j)=Lj;
  end
end
```

Figure 4.12 Pseudo-code of the proposed one-scan CCA algorithm [21].

within the *Translator LUT*. In this way, multiple complex chains of colliding labels are avoided.

The approach adopted to manage the table FT is similar to what is mentioned above. When no collision is detected, and the generic label TL(*Lj*) has been assigned to the current pixel, the *Translator LUT* is again accessed to resume TL(TL(*Lj*)). This allows taking into account that, due to a prior collision, the label *Lj* may have been recognized as equivalent to another label, which causes an update of the *Translator LUT*. Then, to increase the area pixel count of the detected object, FT(TL(TL(*Lj*))) is incremented by one. Conversely, when a collision between TL(*Lx*) and TL(*Ly*) is detected, the minimum (*Lmin*) and the maximum (*Lmax*) among TL(TL(*Lx*)) and TL(TL(*Ly*)) are calculated. This allows identifying those cases in which: (i) *Lx* is greater (smaller) than *Ly*, while TL(*Lx*) is smaller (greater) than TL(*Ly*), and (ii) colliding labels *Lx* and *Ly* have been recognized in previous collisions as equivalent to the same label. If *Lmin* = *Lmax*, the content FT(*Lmin*) is simply incremented by one, otherwise FT(*Lmin*) + FT(*Lmax*) + 1 is stored in FT(*Lmin*) and FT(*Lmax*) is zeroed. The process continues until all the input binary pixels are processed.

The example reported in Figure 4.13 aims to clarify how the proposed algorithm works. After the scan of the first row of the input image is completed, TL(1) = 1, while FT(1) = 1 because only one foreground pixel with label 1 has been counted (Figure 4.12(b)). The scan of the second row does not introduce particular conditions to be managed. Then, for the second foreground pixel in the third row, a collision between *Ly* = 2 and *Lx* = 3 occurs. The minimum label between TL(*Ly*) = 2 and TL(*Lx*) = 3 is now assigned to the current pixel. Then, as shown in Figure 4.12 (c), the *Translator LUT* and the *Features Table* are updated by writing TL(3) = 2, FT(TL(TL(2))) = FT(TL(TL(3))) + FT(TL(TL(2))) + 1 = 4. Finally, the pixel count previously assigned to label 3 (e.g. FT(TL(TL(3)))) is zeroed. Figure 4.13(d) details the tables update when the next collision (1,2) occurs. It is important to note that, because of the previous collision between labels 2 and 3, the count referred to the latter is now transferred to the connected component identified by label 1. Figures 4.13(e) and 4.13(f) illustrate the last two significant occurrences in the process. It can be seen that the last foreground pixel is surrounded by pixels labeled with 2. In this case, the adopted strategy assigns TL(2) = 1 to the current pixel, which interrupts the propagation of incorrect labels and the formation of unmanageable collision chains.

The proposed algorithm has been tested through a Matlab software routine that develops the pseudo-code reported in Figure 4.12. Hundreds of benchmarks with different sizes and patterns have been processed to extract the area feature, the bounding boxes, the centroid, and more features, for each recognized connected component. Some examples of test images are illustrated in Figure 4.14 for both (a) critical patterns, typically generating

					1
		2	*2*		*1*
	3	*2*	*2*	*2*	*1*
4	*2*	*1*			

(a)

(b)

L_j	$TL(L_j)$
0	0
1	1
2	not used
3	not used
4	not used

L_j	$FT(L_j)$
0	0
1	1
2	0
3	0
4	0

(c)

L_j	$TL(L_j)$
0	0
1	1
2	2
3	2
4	not used

L_j	$FT(L_j)$
0	0
1	2
2	2+1+1=4
3	0
4	0

(d)

L_j	$TL(L_j)$
0	0
1	1
2	1
3	2
4	not used

L_j	$FT(L_j)$
0	0
1	2+6+1=9
2	0
3	0
4	0

(e)

L_j	$TL(L_j)$
0	0
1	1
2	1
3	2
4	2

L_j	$FT(L_j)$
0	0
1	9+1+1=11
2	0
3	0
4	0

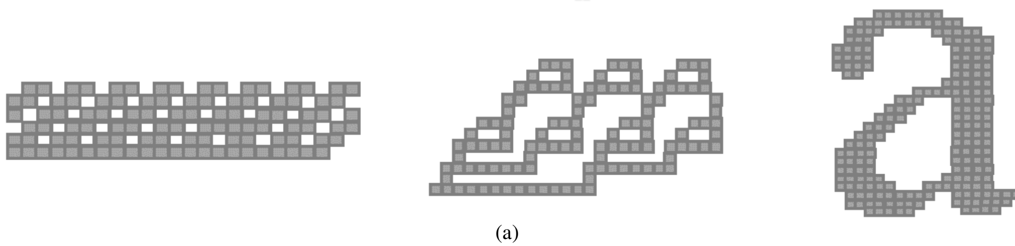
(f)

L_j	$TL(L_j)$
0	0
1	1
2	1
3	2
4	2

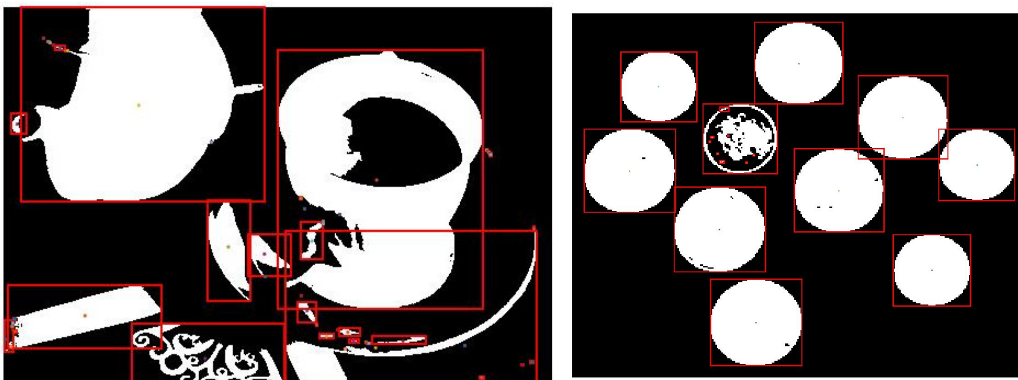
L_j	$FT(L_j)$
0	0
1	11+1=12
2	0
3	0
4	0

Figure 4.13 (a) The input binary image labeled by the novel algorithm. Status evolution of the *Translator LUT* and *Feature Table* (b–f).

complex collision chains, and (b) real frames, for which also extracted bounding boxes and centroids are visible. In all the examined cases, the features extracted by exploiting the proposed algorithm perfectly match those produced by using the functions available in the MATLAB Image Processing Toolbox (e.g., *bwlabel* and *regionprops*). This demonstrates that, as expected, the algorithmic-level optimizations introduced do not affect the quality of the CCA decisions with respect to the conventional approach.



(a)



(b)

Figure 4.14 Sample images used to extract: (a) area features, (b) bounding boxes, and centroids.

4.5.2 Hardware circuit

The top-level architecture of the accelerator implementing the novel CCA algorithm is depicted in Figure 4.15. It has been designed to be included in heterogeneous FPGA-based systems. Therefore, input and output interfaces were realized to sustain standard AXI transactions [80]. Furthermore, on the completion signal an interruption is delivered to the host processor. All internal data buses are endowed with appropriate auxiliary signals used to assert data validity. The external data dispatcher establishes if the CCA circuit is ready to process input pixels through the *Ready* signal received from the *Control Unit*, which orchestrates the whole running of the architecture until the last pixel of the input frame is received. This condition is identified through the appropriate *Last_pixel* and *Valid_Pixel* signaling.

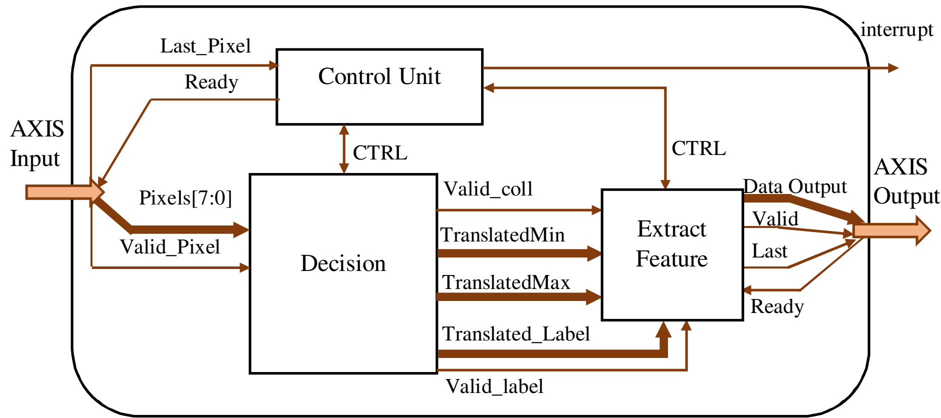


Figure 4.15 Top-level architecture of the proposed CCA accelerator [21].

The *Decision* module, which accommodates the *Translator LUT*, transfers information concerning assigned and equivalent labels to the *Extract Feature* module through the *Translated_Label*, *TranslatedMin*, and *TranslatedMax* data buses. The latter are *nbl*-bit wide, with $nbl = \lceil log_2(N_L - 1) \rceil$. When the last pixel of the input image has been processed in raster order and features of interest are available, an interrupt on completion is generated. Then, features are sent for subsequent elaborations, while the whole architecture is reset. For our specific application, results are transferred through an output AXI-Stream interface.

As illustrated in Figure 4.16 the *Decision* module is composed by a line buffer, the *Translator LUT*, and a decision logic. The former, which consists of two registers (R_1, R_2) and a FIFO, locally stores the provisional labels already assigned to previously scanned pixels and required for correctly arranging a four-connected neighborhood at each clock cycle. It is worth noting that the FIFO stores at most *m*-1 labeled pixels, with *m* being the number of columns in the input image. Since background and foreground incoming pixels

are assumed being equal to 0 and 255, respectively, R_1 is a simple flip-flop storing the LSB of the 8-bit input pixel. In order to establish if a label must be assigned or not to the current pixel, the decision logic evaluates the content of R_1 and the labels, if any, assigned to the neighboring pixels and available within R_2 and at the output of the FIFO. As visible in Figure 8, these labels are directly connected to the inputs ADDRA and ADDRB of the multi-port memory block acting as the *Translator LUT*. Corresponding asynchronous outputs DOA and DOB are sent to the decision logic block that implements the rules detailed in the previous section. Such a block assigns the correct label to the current pixel, which is then stored in R_2. In the next clock cycle, the translated value of the label stored in R_2 enters the FIFO. This strategy avoids label collision chains to be left unresolved. When a collision is detected, the *Translator LUT* needs to be updated, as discussed in the previous subparagraph. To this purpose, the minimum (MIN) and maximum (MAX) between DOA and DOB are calculated and, in the next clock cycle, the *Translator LUT* is updated through the DIND port, by writing TL(MAX) = MIN. This information is then transferred to the *Extract Features* block after a second access to the *Translator LUT*, performed in a pipeline fashion, through the ports ADDRF and ADDRG. After a minimum-maximum calculation, DOF and DOG, are outputted as *TranslatedMin* and *TranslatedMax*. Furthermore, the translated content of R_2, made available through the output port DOE, is opportunely delayed and then sent to the *Extract Features* block by the data bus named *Translated Label*.

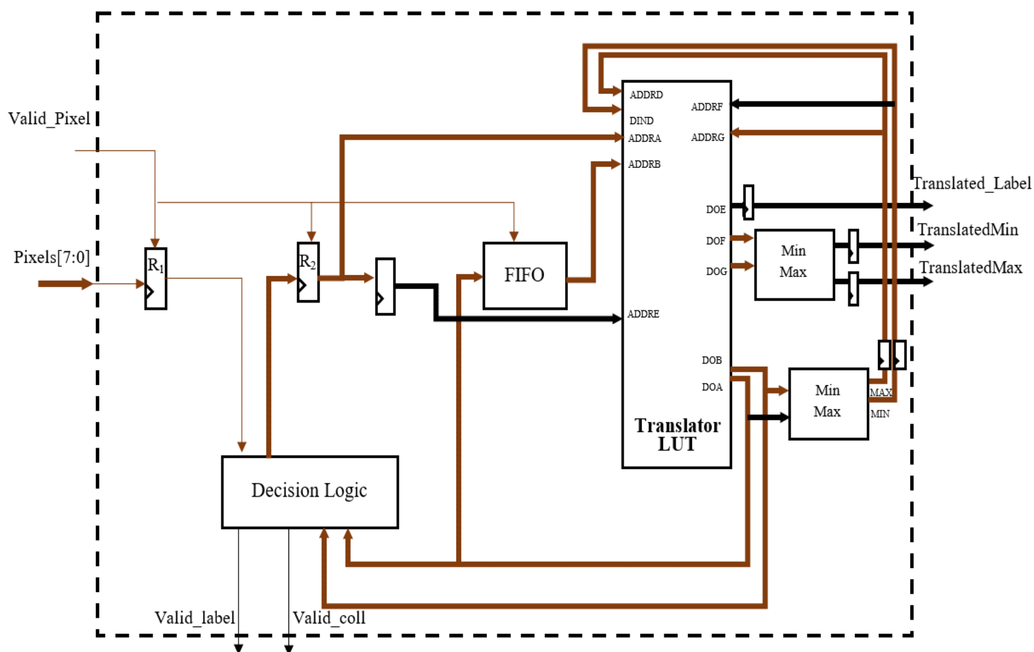


Figure 4.16 The *Decision* circuit designed for the novel CCA.

The Extract Feature module is structured as depicted in Figure 4.17. It consists of the *Features Table* and the *Zeros Table*, besides the control circuits used for generating write enable (WEAFT and WEBZT) signals and AXIS protocol signals. The *Zeros Table*, used as auxiliary memory to allow a double writing operation within the same clock cycle, is implemented as a one-bit Simple Dual Port memory of size N_L, and it is initialized to 1. Such a memory intervenes in the case of a collision between *TranslatedMin* and *TranslatedMax*. As described in sub-Section 4.5.1, in this case, the position FT(*TranslatedMax*) must be reset, while the updated count is written in FT(*TranslatedMin*). To perform this action, its *TranslatedMax* location is zeroed through the port DINB.

Figure 4.17 The *Extract Feature* circuit designed for the novel CCA.

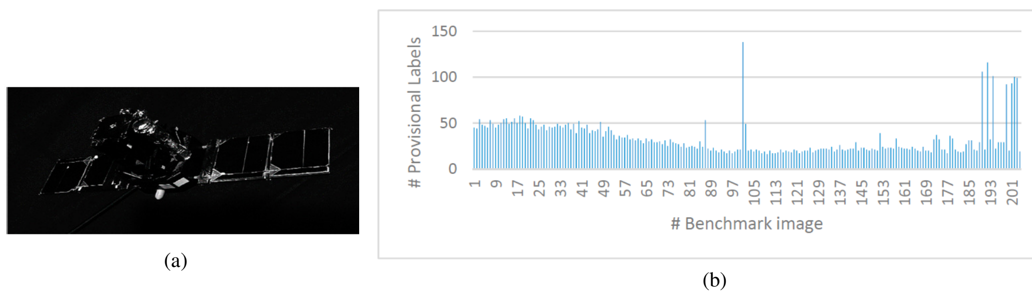
When a valid translated label appears on the *Translated_Label* bus, the area consistency of the connected component associated with that label is read by means of the port ADDRA of the *Features Table*. It is incremented by one and rewritten through the same port during the next clock cycle. On the other side, when a valid collision between *TranslatedMin* and *TranslatedMax* is detected, the *Features Table* is asynchronously accessed through the ports ADDRA and ADDRB. The contents outputted on DOUTA and DOUTB are preliminary AND-ed with the contents uploaded from the *Zeros Table* at the same addresses; then, they are summed and incremented by 1. The final result is stored in the *Features Table* at the next clock cycle through the port DINA.

When all pixels are processed, the *Control Unit* activates the *AXIS FSM* that scans the *Features Table*, transmits results over the AXIS-Stream interface, and simultaneously re-initializes the memory for the subsequent elaboration. It is worth noting that, while in the referred implementation we used an AXI-Stream output interface, for several simpler

applications, a cheaper AXI-Lite interface could suffice, which further saves logic resources.

4.5.3 Experimental results for aerospace applications

The accelerator described in 4.5.2 has been integrated within a more complex image processing system for aerospace applications and implemented on a Zynq-7000 FPGA SoC [33]. The embedded system used to test the proposed CCA accelerator includes a DMA that transfers 640×480 input frames from the DDR external memory to the custom circuit, and writes results of the feature extraction process again into the DDR. Several hundreds of benchmark images significant for the specific application has been thoroughly analyzed to appropriately choose all design parameters. Figure 4.18 reports a sample test image and a summary of the number of assigned labels obtained by means of Matlab simulations over 204 benchmark images. Such an investigation suggests that NL=256 well satisfies the specific application. Furthermore, a data word of 16-b was chosen for the Features Table memory.



(a)

(b)

Figure 4.18 (a) An example of benchmark image; (b) Histogram of the number of provisional labels assigned by the proposed algorithm to the examined image dataset.

Table 4.3 summarizes results obtained by comparing the proposed CCA architecture with several implementations existing in the literature. It is worth pointing out that none of the cited works implements AXI interfaces for direct inclusion in heterogeneous FPGA-based systems. It is also important to note that, to improve resources efficiency, all internal memory banks are implemented by using 352 LUTs of distributed RAM, instead of Block-RAM. If Bounding Boxes are the features of interest, 176 more LUTs of distributed RAM for the *Features Table* are required to accommodate the coordinates of connected components instead of their area pixel counts. Since different FPGA device families were used in references [97],[109] and [110] a direct comparison with the novel CCA accelerator in terms of speed performances and resources requirements would be unreliable. However, both the number of managed labels N_L and the number of clock cycles per pixel reported in Table 4.3 are not related to the prototyping platform used. For what concerns the number of clock cycles per pixel, it can be observed that the proposed design, as well as the

architecture presented in [110], reaches a unitary throughput, which is significantly higher than [94] and [109], independently of the used technology.

Table 4.3 Characterization of the proposed CCA accelerator [21] and other FPGA-based one-scan CCA designs.

	[109]	[97]	[110]	New	
Technology	Virtex II (150 nm)	Kintex-7 (28 nm)	Virtex II (150 nm)	XC7Z020 (28 nm)	
Image Size	640 × 480	256 × 256	640 × 480	640 × 480	
Feature	A [1]	BB [2]	BB	A	A
N_L	128	130	320	256	128
LUTs	1757	493	654	760	506
FFs	600	296	227	787	774
BRAM [kb]	72	108	92	0	0
f [MHz]	40.64	185.59	97.07	100	140
Cycles/Pixel	1.25	1.25	1	1	1
Mpps	31	141.59	92.57	95.37	133.5

[1] Area, [2] Bounding Boxes

For a fair discussion, we introduce the resource efficiency parameter as defined in (14). It can be seen that the maximum number of labels N_L is related to the amount of resources, expressed in terms of kbits, and multiplied by the number of clock cycles per pixel. In order to calculate the kbits contribution, a weight to each FPGA resource is associated for the different technologies. As an example, the Kintex-7 FPGA used in [94] is equivalent, in terms of area occupancy, to the Artix-7 technology integrated within the XC7Z020 SoC employed for the proposed implementation. More in detail, each CLB of the Series-7 family contains two slices, and each slice is composed by 4 six-input LUTs and 8 flip-flops. By associating each flip-flop to one bit and each LUT to 2^6 bits, the generic slice is accounted as 264 bits. Conversely, for the Virtex 2-based designs [109] and [110] the occupied kbits take into account that each CLB includes four slices, each having 2 four-input LUTs and 2 flip-flops. That means Virtex-2 devices allow generating boolean function with at most four inputs (or alternatively implementing distributed memory with at most 2^4=16 positions) by a single LUT.

$$Resource_{EFF} = \frac{N_L \times 1024}{Resource\ [kb] \times \frac{Cycles}{Pixel}} \tag{14}$$

Figure 4.19 plots the resource efficiency achieved by the compared designs. Obtained results demonstrate that the proposed implementation, with its higher efficiency, allows the best trade-off between hardware requirements, speed performances, and computational capability to be achieved.

Memory requirements are further analyzed in Table 4.4, where the number of bits used to represent labels and features are indicated with *nbl* and *nbf*, respectively. Such a comparison shows that, in contrast to references [97] and [109], the novel architecture does not need the merger table since the *Translator LUT* is made able to implement both merging and translation operations. It is also worth noting that, unlike the CCA architectures

demonstrated in works [109] and [110], the approach proposed here does not need further auxiliary tables.

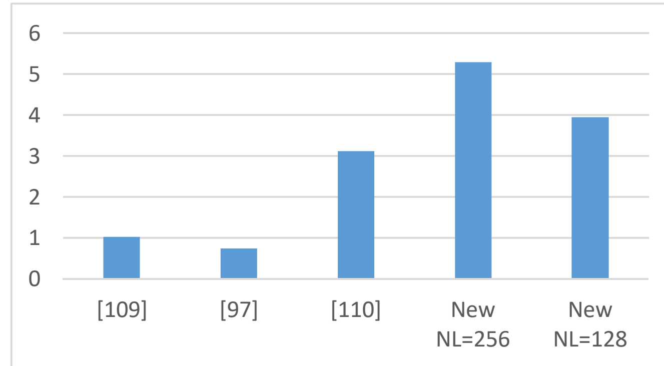


Figure 4.19 Resources efficiency for different FPGA-based one-scan CCA accelerators.

Table 4.4 Memory requirement of several CCA architectures processing *n*×*m* input images.

Resource	**Memory Requirements (bits)**			
	[109]	[97]	[110]	New
Row Buffer *RB*	$m \times nbl$	$\left\lceil \frac{m}{2} \right\rceil \times nbl$	$2 \times m$	$1 + (m-2) \times nbl$
Collision Table *S*	$N_L \times nbl$	$m \times nbl$	-	-
Merger Table *MT*	$m \times nbl$	$N_L \times (nbl + \lceil log_2(n) \rceil)$	-	-
Translator LUT *TL*	$N_L \times nbl$	-	-	$N_L \times nbl$
Reuse FIFO *R*	-	$N_L \times nbl$	-	-
Label Stack *LS*	-	$m \times \left\lceil \frac{nbl}{10} \right\rceil$	-	-
Valid flags *V*	-	nbl	-	-
Active tags *E*	-	m	-	-
Next Table *NT*	-	-	$N_L \times nbl$	-
Head Table *HT*	-	-	$N_L \times nbl$	-
Tail Table *TLT*	-	-	$N_L \times nbl$	-
Data Table *DT*	$2 \times N_L \times nbf$	$N_L \times nbf$	$N_L \times nbf$	$N_L \times nbf$

4.6 SUMMARY

CCA transforms an input raw image into synthetic data, which is a crucial step within smart visual IoT nodes. For this reason, designing high-performance and energy efficient architectures is becoming a hot focus in the research field. In this chapter, two designs have been presented to efficiently implement the CCA task within heterogeneous embedded systems. The former [19][20] is thought to adopt the well-known one-scan CCA algorithm in addition to a complete labeling of the input raw image without introducing the second

scan required by traditional techniques. This has been made possible by overlapping the collisions resolution phase with the software actions performed by the processor to configure the DMA core. Results demonstrated that, in a typical application of traffic sign recognition, when 640×480 images are received as inputs, the proposed design operates at the 225MHz running frequency, thus it extracts coordinates of the bounding boxes processing up to 724fps while consuming just 138.4μJ on average.

The second architecture has been designed to hardware implement an innovative one-scan CCA algorithm [21]. In this case, both algorithmic and architectural optimizations are introduced. When the proposed CCA accelerator processes 640×480 binary images captured in the space environment, it processes 325.5 frames per second at 100MHz on the Xilinx Zynq-7000 XC7Z020 SoC. When compared to most relevant state-of-the-art FPGA-based designs, the novel architecture exhibits the highest resource efficiency and the best trade-off between hardware requirements, speed performances and computational capability.

5 INFERENCE OF DEEP CNNS ON HETEROGENEOUS SOCS

Deep Learning (DL), being a strong analytic tool for huge volume of data, is surely one fundamental element in the rapid development of the edge computing IoT paradigm [121]. In contrast to traditional machine learning techniques that failed with the problem of mining reliably data from a noisy and complex environment, deep learning models allow extracting accurate information from raw sensed data that can be transferred from the node to the cloud without incurring in communication overhead. Moreover, exploiting deep learning in edge computing saves the user's privacy. Indeed, synthetic data provided by deep learning models has different semantics compared to the source data. As an example, it is very hard to capture the original information contained in a raw frame starting from the features extracted, layer-by-layer, by a CNN model.

Unfortunately, deploying deep learning, and in particular CNN algorithms, on smart IoT sensor nodes is not trivial, due to several aspects to be taken into account, e.g. energy consumption, performances, connectivity, etc. In such a context, the choice of the computing platform plays an important role. As broadly discussed in Chapter 2, general-purpose CPUs are not suitable for running CNNs, because of the low performances and energy efficiency. Many researchers have proposed FPGA- [52]-[54], GPU- [122], and ASIC-based [123]-[124] implementations of hardware accelerators for CNNs. As GPUs feature high memory bandwidth and throughput, as well as an exceptional efficiency in floating-point matrix-based operations, they are preferable in the training phase of CNNs. However, GPU-based accelerators consume significant amounts of power: for the same given functional design, the consumption of a single GPU is tens or even hundreds times higher than the power consumption of the FPGA counterpart [125]. On the other hand, compared with GPU-based accelerators, FPGA and ASIC hardware designs provide limited memory resources, I/O bandwidths, and computational capabilities. However, they can achieve reasonable performances with lower power consumption [10], [51]-[52], [123]-[124], although the development cycle, cost, and flexibility are not satisfactory in ASIC-based acceleration of deep learning networks [13], [32].

In recent years, the reconfigurability, the good energy-efficiency, the reduced development time of high-performance products, as well as the availability of high-level synthesis tools, made FPGAs the preferred platforms for realizing hardware accelerators of CNNs [10], [22], [52]-[54], [126]-[129]. In the following, a more exhaustive state-of-the-art is provided.

5.1 BACKGROUND AND RELATED WORKS

One of the most representative examples of applications of deep CNNs is the task of image classification [24], [122], [130], which allows extracting features from images and classifying them into a certain group of categories. A CNN extracts features of interest, such as corners, edges, circular arch, etc., from 2D input images by performing convolutions. The output of the feature extractor is then inputted to the classifier that determines the likelihood that a specific object into the input image might belong to one of the classes defined by the CNN model. Generally speaking, such a deep CNN is composed by a certain number of cascaded convolutional layers (CONVs), each structured as schematized in Figure 5.1(a). The generic CONV layer receives a set of M 2D arrays of input data, named input feature maps (*ifmaps*), and produces a set of N 2D output feature maps (*ofmaps*). It is worth noting that, to furnish the generic *ofmap*, each *ifmap* is filtered using a distinct 2D $K \times K$ convolution kernel and the results are combined by a pixel-wise addition. The N intermediate *ofmaps* obtained in this way are further processed by applying a non-linear function, such as the rectified linear unit (ReLU) [131], which sets negative values to zero, while keeping positive values unchanged. Afterwards, normalization or quantization and sub-sampling are optionally performed. The Mc *ofmaps* produced by the last convolutional layer are finally processed by the classifier, typically implemented by the so-called fully-connected (FC) layers. As schematized in Figure 5.1(b), also the generic FC layer applies filters on the *ifmaps*, but in this case the filters are of the same size as the *ifmaps*. Moreover, each *ifmap* is multiplied by a distinct kernel computing a matrix product (MP); the resulting Mc matrix products are added and then processed by the ReLU module to furnish one element of the $1 \times 1 \times Nc$ *ofmap*. Usually, also the classification is performed through several cascaded FC layers.

In the last few years, several deep CNN models have been demonstrated, each exploiting differently cascaded CONV and FC layers to achieve even greater accuracy [132]-[135]. However, as discussed in [136], state-of-the-art deep CNNs have an important issue in common: convolution operations consume over 90% of the computational time, thus becoming the critical task responsible for limiting reachable speed performances. For this reason, the design of hardware convolutional engines able to accelerate deep CNNs recently received a great deal of attention.

The most obvious hardware solution to accelerate convolutions is maximizing resources utilization and parallelism. To this purpose, a possible strategy consists in directly mapping CNNs onto FPGA devices, as in [136]. Unfortunately, such a technique has been demonstrated only on small-sized networks [138]-[139], which may be easily accommodated into logic fabric without filling all the available resources. Conversely,

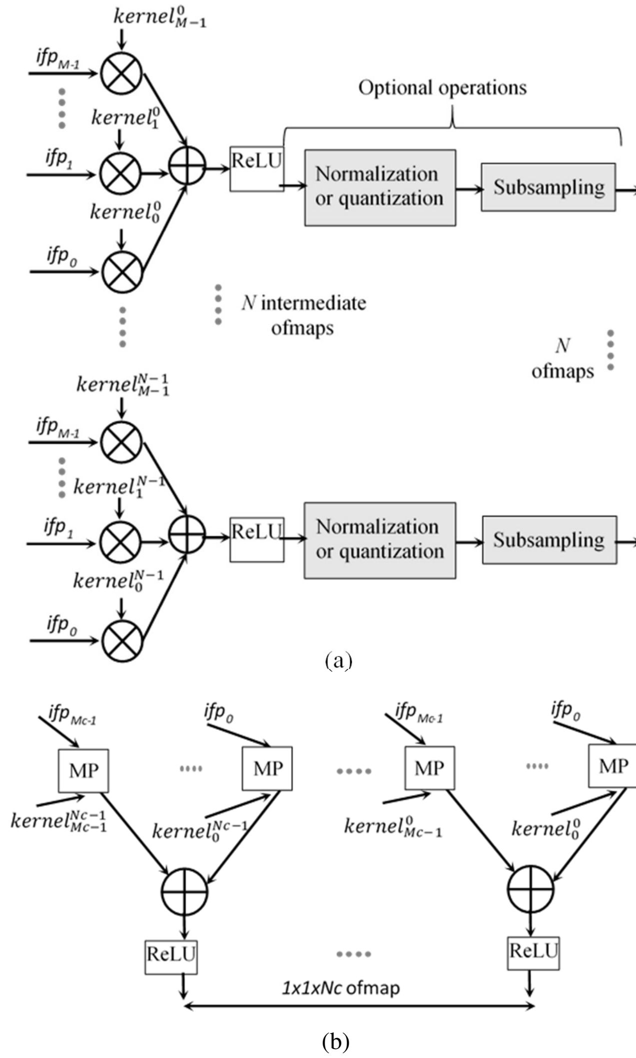


Figure 5.1 Computations performed by (a) a CONV layer and (b) a FC layer.

architectures in [52]-[53], [127]-[129] have been designed for hardware accelerating large-scale CNNs. In that case, moderate performances were achieved by occupying hundreds of on-chip DSPs and BRAMs, with negative effects on power consumptions. It is then clear that, when saving resources and power consumption are desirable design targets, poor performances are achieved, as demonstrated in [54].

Several studies [52]-[54], [127] discuss automated design methodologies that offer good performances-resources trade-off. Such approaches make use of high-level synthesis tools that abstract the algorithm description (e.g. the multiple convolution operations required by a CONV layer are reduced to six nested loops). On the other hand, such tools often do not allow a low-level control on each hardware component to be maintained (e.g. the synthetized design is not efficiently mapped within the target FPGA device).

The memory footprint is another challenge in CNN accelerators, since FPGAs are memory-limited and external supports have to be used for storing the huge amount of network parameters. Several techniques have been adopted to mitigate such an issue,

including binary weights [140], network pruning [141], and data quantization [142]. The latter allows memory requirements to be reduced, improving energy efficiency and performances, and enabling hardware accelerators to support large-scale CNNs on systems with a limited memory budget. State-of-the-art quantization methods typically refer to fixed-point representations for CNN inference. Gysel *et al.* [142] demonstrated that large CNNs can be quantized to 8-bit for both weights and layer outputs, while keeping the accuracy loss below 1% compared to its 32-bit floating-point counterpart. Exploiting 8-bit reduced precision CNNs allows: **i)** reducing the number of accesses to the external memory by storing intermediate results on-chip [129], and **ii)** using DSP slices available within modern FPGAs to realize fast double [22], [144]-[146] MAC architectures. Unfortunately, the above-mentioned existing designs require auxiliary operations to correct the output of each DSP block used to perform multiplications, that, as it will be pointed out later, cause detrimental effects on speed performances, resources requirements and power consumption. Furthermore, the architecture proposed in [144] uses a memory access policy that does not allow exploiting the high-performance continuous data streaming for input and output data transfers. Thus, such a kind of accelerators cannot be easily integrated within real-time heterogeneous embedded systems, like those described in [10] and [128].

In the next paragraph, the architecture proposed in [10] for accelerating the inference of 8-bit reduced-precision CNNs is described. The circuit is purposely designed to support SIMD operations, and in particular it implements an efficient double MAC architecture able to overcome the issues arising from [144]-[146]. Table 5.1 summarizes parameters and acronyms used in this chapter.

5.2 The proposed SIMD CNN Accelerator

The proposed CNN is made able to process T_M *ifmaps* in parallel to produce T_N *ofmaps* contemporaneously. Thus, the generic layer receiving M *ifmaps* and producing N *ofmaps* completes its operations within $ns=(M/T_M)\times(N/T_N)$ computational steps.

The top architecture is illustrated in Figure 5.2. The *SIMD buffer* catches the flow of input data, as ruled by the AXI stream (AXIS) protocol, and, after a known latency depending on the number of columns W in the *ifmaps*, prepares T_M convolution windows for the subsequent *SIMD Convolution Engine* (*CE*). In the meantime, the module *Store Kernels* reads from the external memory, through AXI-Full transactions, the $K\times K\times T_M\times T_N$ kernel coefficients required to perform all parallel convolutions and then provides them to the *SIMD CE*. Obtained results are accumulated by exploiting a local memory buffer. The above task is performed by the *Accumulate* module that finally outputs the intermediate

Table 5.1 Acronyms and parameters used to within the proposed SIMD CNN accelerator [10].

ACRONYMS		
CONV	Convolutional layer	
ifmap	Input feature map	
ofmap	Output feature map	
ReLU	Rectified Linear Unit	
FC	Fully connected layer	
MAC	Multiply accumulate	
DMAC	Double MAC	
CE	Convolutional Engine	
FSM	Finite State Machine	
AXI	Advanced Extensible Interface Protocol	
AXI4-Full	Interface used for multiple memory mapped transactions with high-throughput	
AXI4-Lite	Interface used for single memory mapped transactions	
AXI4-Stream	Interface for high-speed streaming	
PARAMETRS		
M	Number of *ifmaps* of a CONV	The number of *ifmaps* received by the generic convolutional layer as input
N	Number of *ofmaps* of a CONV	The number of *ofmaps* produced by the generic convolutional layer as output
$K\times K$	Kernel size	The size of the generic convolution kernel
M_C	Number of *ifmap* of a FC	The number of *ifmaps* received by the generic FC as input
N_C	Number of outputs of a FC	The generic FC produces a $1\times1\times N_C$ *ofmap*
$K_C\times K_C$	Kernel size	The size of the generic convolution kernel of the generic FC
T_M	Parallelism level on *ifmaps*	The number of *ifmaps* processed in parallel by the generic CONV
T_N	Parallelism level on *ofmaps*	The number of *ofmaps* outputted by the generic CONV contemporaneously
ns	Computational steps	The number of computational steps required to complete the generic CONV: $ns = \frac{M}{T_M} \times \frac{N}{T_N}$
$H\times W$	Size of the generic *ifmap*	H and W are the number of rows and columns within the generic *ifmap*
ifp_j	The generic *ifmap*	The *j*-th *ifmap*, with j=0, …, M-1, received by the generic CONV as input
$ifp_j(h,w)$	The generic value of ifp_j	The value located at the row h and the column w within the *ifmap* ifp_j
$C_m^n(s,t)$	The generic kernel coefficient	The kernel coefficient located at the row s and the column t within the K×K convolution kernel (s and t vary from 0 to K-1) and used to filter the *ifmap* ifp_m when the *n*-th *ofmap* is computed
nr	Radius of the convolution window	$nr = \frac{K-1}{2}$
FS	Feature size	The parameter $FS = \frac{W}{2}$ furnished by the FSM
Y, Z	DSP operands	The input operands of the generic DSP
b	DSP operand bit width	Number of bits of the operand Y
d	DSP operand bit width	Number of bits of the operand Z
N_{DMAC}	DMAC parallelism level	Number of DMAC blocks in the generic processing element
u	Accumulator bit width	Number of bits of the accumulator integrated within the generic DSP

ofmaps. On the basis of configuration data provided by the PS, as ruled by the AXI-Lite protocol, and depending on the signals *Flags*, produced by the computational modules during their operations, the *FSM* feeds all blocks with control signals (*CTRL*) orchestrating their activities appropriately for the current step of the computation. When the last step has been performed, the accumulation and the access to the local memory are disabled and the *ReLU & Quantization* module is activated since all the intermediate steps are completed. The quantized *ofmaps* are optionally sub-sampled by the *Pooling* module and then outputted.

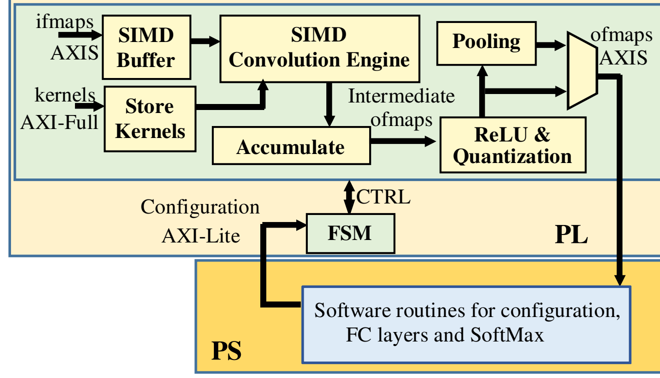


Figure 5.2 Top-architecture of the proposed CNN accelerator [10].

Input data and kernel coefficients are numerically represented by using unsigned and signed 8-bit fixed-point formats, respectively. In order to exploit efficiently the SIMD paradigm in accordance with the resources provided by low cost devices, two couples of 8-bit values belonging to two different *ifmaps* are accommodated within one 32-bit word. More exactly, the *ifmaps* (*ofmaps*) are stored within the external memory in the raster scan order applying the data packing strategy schematized in Figure 5.3 for the generic convolutional layer. In this way, two *ifmaps* ifp_j and ifp_{j+1}, with $j=0, \ldots, M-1$, can be stored within a memory area starting at the address $BAddr_j$ and consisting of $H \times \frac{W}{2}$ 32-bit locations, with $H \times W$ being the size of the *ifmaps* processed by the current convolutional layer. In fact, two adjacent elements of ifp_j are interleaved with two adjacent elements of ifp_{j+1}. The proposed architecture is designed so that *ofmaps* are outputted already packed as above described. Therefore, no specific data re-adjustment is required between convolutional layers.

Memory Address	Four 8-bit values packed in a 32-bit word
$BAddr_j$	$ifp_j(0,0)$ $ifp_j(0,1)$ $ifp_{j+1}(0,0)$ $ifp_{j+1}(0,1)$
$BAddr_j + 4$	$ifp_j(0,2)$ $ifp_j(0,3)$ $ifp_{j+1}(0,2)$ $ifp_{j+1}(0,3)$
⋮	⋮
$BAddr_j + 4\times(H \times \frac{W}{2})$	$ifp_j(H\text{-}1,W\text{-}2)$ $ifp_j(H\text{-}1,W\text{-}1)$ $ifp_{j+1}(H\text{-}1, W\text{-}2)$ $ifp_{j+1}(H\text{-}1,W\text{-}1)$

Figure 5.3 Data packing strategy to store data into the external DDR memory.

A slightly different packing strategy is used for the kernel coefficients, which are indeed packed within 64-bit words to transfer from the external memory eight coefficients at the same time. In order to explain the arrangement adopted to store $K \times K$ convolution kernels into the external memory, let consider the indices s and t, ranging between 0 and $K-1$; m varying from 0 to $T_M - 1$; and n ranging between 0 and T_N-1. In this way, the generic kernel coefficient can be named $C_m^n(s,t)$ to indicate that it contributes to the computation

of the n-th *ofmap* and that it is located at the position *(s,t)* within the kernel used for filtering the m-th *ifmap*. The convolution kernels are stored within a memory area starting at the address *BCAddr* and following the strategy schematized in Figure 5.4 for the case in which T_M=8. It can be seen that coefficients having homologous positions within distinct eight kernels are packed in the same 64-bit word and stored at the proper address *CAddr* that can be obtained by (15), depending on n, s, t, K and T_M.

$$CAddr = (BCAddr + 8 \times (s \times K + t) + 8 \times n \times K \times K) \times \frac{T_M}{8} \quad (15)$$

Memory Address *CAddr*	**Eight 8-bit coefficients packed in a 64-bit word**
BCAddr	$C_0^0(0,0)\ C_1^0(0,0) \ldots C_6^0(0,0)\ C_7^0(0,0)$
BCAddr+8	$C_0^0(0,1)\ C_1^0(0,1) \ldots C_6^0(0,1)\ C_7^0(0,1)$
⋮	⋮
BCAddr+ 8×(K×K-1+K-1)	$C_0^0(K-1,K-1)\ C_1^0(K-1,K-1) \ldots C_6^0(K-1,K-1)\ C_7^0(K-1,K-1)$
⋮	⋮
BCAddr+ T_{N-1}×8×K×K	$C_0^{TN-1}(0,0)\ C_1^{TN-1}(0,0) \ldots C_6^{TN-1}(0,0)\ C_7^{TN-1}(0,0)$
⋮	⋮
BCAddr+ 8×(K×K-1+K-1)+ T_{N-1}×8×K×K	$C_0^{TN-1}(K-1,K-1)\ C_1^{TN-1}(K-1,K-1) \ldots C_6^{TN-1}(K-1,K-1)\ C_7^{TN-1}(K-1,K-1)$

Figure 5.4 Data packing strategy adopted for storing the 8-bit convolution kernels (case T_M=8).

5.2.1 Architecture of the reconfigurable SIMD buffer

The SIMD paradigm has been adopted to make the proposed accelerator able to parallel process two adjacent values of each of the T_M *ifmaps* received as input, and to furnish two adjacent values of distinct T_N *ofmaps* in parallel. In order to do this, a window consisting of $K \times (K+1)$ values must be patched over each *ifmap* to accommodate two adjacent convolution windows. The SIMD reconfigurable buffer depicted in Figure 5.5(a) uses T_M instances of the internal buffer *IBuff*, each consisting of *K-1* FIFOs and $K \times (nr + 1) + nr + 2$ registers, with $nr = \frac{K-1}{2}$ being the radius of the convolution window. The input stream received by the buffer transfers T_M couples of 8-bit data that must be correctly prepared for executing parallel SIMD convolutions adopting the zero-padding at the borders of the *ifmaps*. For this reason, the input stream is split within T_M different 16-bit streams each sent to one module *IBuff*. The latter further splits each incoming 16-bit data into two 8-bit data that feed two different pipes: the former consisting of the *nr+1* registers R_1, …, R_{nr+1}, and the other one composed by the *nr+2* registers R_{nr+2}, …, R_{2nr+2} and R_{aux}.

The latter is required to correctly pair incoming values when *nr* is odd. In order to better explain why this is necessary, let consider the example of Figure 5.5(b) that shows the case in which the generic *IBuff* receives a *4×8 ifmap* and arranges *3×3* convolution windows. It is important to note that, due to the zero padding, the first incoming pair of adjacent values A_1 and B_1 do not have homologous positions within the two highlighted convolution windows. In fact, A_1 is homologous to the padding zero value, whereas B_1 is homologous to A_2. To guarantee that the incoming data will be multiplied by the correct kernel coefficients, they must be properly recoupled before reaching the $FIFO_1$. This is done through five registers, R_1, R_2, R_3, R_4 and R_{aux} as shown in the timing diagram also illustrated in Figure 5.5(b). It is easy to verify that, when *nr* is even, incoming data are already correctly paired. In this case, the register R_{aux} has no effect and the module *CONCAT* leaves the incoming pairs of adjacent data unchanged. The data-path then goes on through the subsequent FIFOs and registers that furnish data depending on *nr*, as summarized in Figure 5.5(c), where the symbol '&' is used to indicate concatenations of two 8-bit registers.

The module *Recognize Borders* establishes if the central values within the current convolution windows belong to the borders of the *ifmaps*, thus requiring the zero padding. This information is easily obtained by the *Counters* module of Figure 5.5(a) that traces the location *(r,c)* occupied by the current central values within the *ifmaps*. If necessary, appropriate values of the current convolution windows are masked with zeros before reaching the output stream.

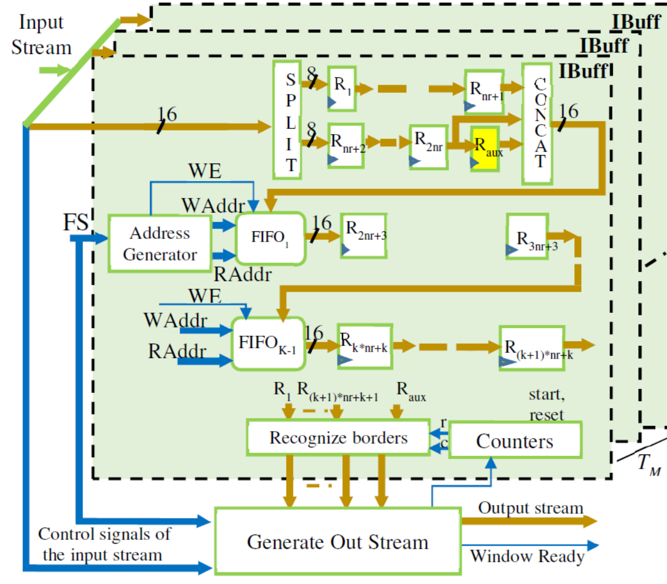
In order to make the SIMD buffer reconfigurable, i.e. adaptable to the sizes of *ifmaps* throughout all the convolutional layers of the accelerated CNN model, FIFOs are realized using blocks of RAMs (BRAMs) configured as dual-port memories. The module Address Generator furnishes the addresses *RAddr* and *WAddr*, where data must be read and write, respectively, taking into account the FIFOs depth as defined in (16), with $FS = \frac{W}{2}$ being provided as a control signal by the external FSM.

$$FIFO_{depth} = FS - (K - 1) \tag{16}$$

As shown in (17), also the latency required by the SIMD buffer to prepare the first T_M couples of convolution windows depends on *FS*.

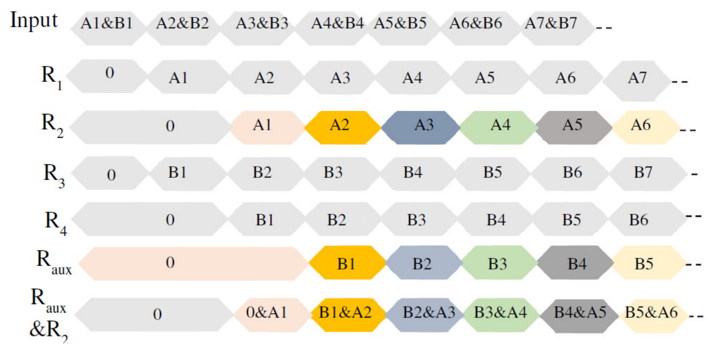
$$Latency_{Buffer} = \big((K - 2) \times FS\big) + (K - 1) \tag{17}$$

The module *Generate Out Stream*, depicted in Figure 5.5(a), uses *FS* to determine the latency and, depending on it, to correctly generate the signal *Window Ready*; to start and reset the Counters module; and to properly produce the output AXI4-Stream then



(a)

0	0	0	0	0	0	0	0	0	0
0	A1	B1	A2	B2	A3	B3	A4	B4	0
0	A5	B5	A6	B6	A7	B7	A8	B8	0
0	A9	B9	A10	B10	A11	B11	A12	B12	0
0	A13	B13	A14	B14	A15	B15	A16	B16	0
0	0	0	0	0	0	0	0	0	0



(b)

Data provided when *nr* is odd			
R_{nr+3} & R_1	…	R_{2nr+2} & R_{nr}	R_{aux} & R_{nr+1}
R_{2nr+3}	…	R_{3nr+2}	R_{3nr+3}
…			
$R_{k*nr+k+1}$	…	$R_{(k+1)*nr+k}$	$R_{(k+1)*nr+k+1}$

Data provided when *nr* is even			
R_{nr+2} & R_1	…	R_{2nr+1} & R_{nr}	R_{2nr+2} & R_{nr+1}
R_{2nr+3}	…	R_{3nr+2}	R_{3nr+3}
…			
$R_{k*nr+k+1}$	…	$R_{(k+1)*nr+k}$	$R_{(k+1)*nr+k+1}$

(c)

Figure 5.5 The proposed reconfigurable SIMD Buffer: (a) the architecture; (b) the strategy used to recouple input data; (c) data provided by the registers depending on *nr*.

transferred to the CE. At the same time as the convolution windows prepared by the SIMD

buffer, the kernel coefficients involved in the current computational step must be provided by the *Store Kernels* module visible in Figure 5.2. Kernel coefficients are uploaded by the external memory during the latency time introduced by the SIMD buffer and locally stored. The *Store Kernels* module stores $K \times K \times T_M \times T_N$ 8-bit coefficients packed within 64-bit words as above shown in Figure 5.4 and receives the signal *Window Ready* furnished by the SIMD buffer to establish when the coefficients must be sent to the *SIMD CE*.

5.2.2 Architecture of the SIMD Convolutional Engine

The proposed *SIMD CE* can be realized using any kind of DSP slices providing at least one $b \times d$ multiplier and one u-bit accumulator (with $b \geq 25$, $d \geq 8$ and $u > 33$). However, making the DSPs able to perform parallel MACs operating in SIMD fashion it is not a trivial task. Let's indicate with A, B and C two adjacent packed elements of a generic *ifmap* (i.e. A=ifp_j(h,w) and B=ifp_j(h,w+1)) and a kernel coefficient, respectively. To the above-mentioned purpose, two independent products A×C and B×C have to be computed in parallel. As schematized in Figure 5.6, the input values A and B need to be re-arranged within the b-bit input Y of a DSP interposing eight zero bits to each other and zeroing the remaining MSBs of Y to guarantee that the operand A is always treated as an unsigned value. Conversely, the d-bit operand Z is used to input the sign extended 8-bit coefficient C. When the latter is negative, the DSP applies the 2's complement notation to the overall result instead of the two separate products, thus making necessary an increment by one on the product A×C to compensate the introduced error. This issue has been solved taking into account that the products A×C and B×C are accommodated within the (b+d)-bit output of the multiplier occupying the (b+d-16) MSBs and the 16 LSBs, respectively. Due to this, the correction can be done adding the auxiliary u-bit operand X by using the accumulator internal to the same DSP slice that performed the multiplication. In order to increment by one the product A×C while leaving B×C unchanged, when C is negative, it is sufficient making X equal to 2^{16} asserting only its 17-th bit. Conversely, when C is positive, X must

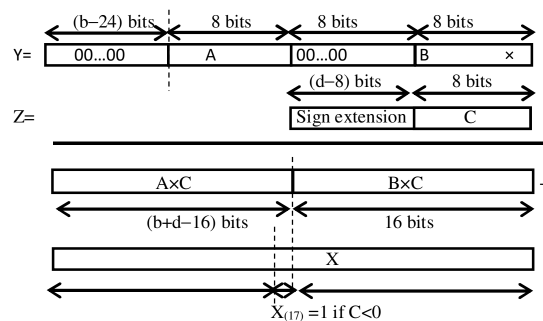


Figure 5.6 SIMD multiplication with a common operand C and one guard-bit.

be set to zero. In this way, the cascaded DSPs, used to perform double MAC operations as below described, can complete their operations without leaving their dedicated fast chains, as instead occurs in [145].

As depicted in Figure 5.7, the module *SIMD CE* proposed here consists of T_N Processing Elements (PEs) each receiving the convolution windows furnished by the *SIMD Buffer* and its own kernel coefficients provided by the *Store Kernels* module. In this way, distinct T_N *ofmaps* are computed in parallel. The *SEPARATE* module routes the data streamed by the *SIMD Buffer* to the PEs while whereas the module *Generate Out Stream* arranges the results as ruled by the AXI4-Stream protocol. The *SIMD buffer* transfers to the *CE* $K\times K\times T_M$ pairs of 8-bit data, which are re-arranged over $K\times K\times T_M$ b-bit words.

As visible in Figure 5.7, the generic PE consists of N_{DMAC} DMACs, each responsible for processing $\frac{K\times K\times T_M}{N_{DMAC}}$ b-bit data through as many DSP slices configured to perform SIMD MACs. Each DSP running as a MAC receives one packed b-bit operand and one kernel coefficient C as inputs and computes two parallel 16-bit products $A\times C$ and $B\times C$ as above shown in Figure 5.6. In order to perform the subsequent accumulations correctly, each SIMD result is re-arranged over u bits by the module *INSERT GUARD BITS*. The latter sign extends the 16-bit product $B\times C$ to $\frac{u}{2}$ bits, and left shifts the 16-bit product $A\times C$ by $(\frac{u}{2}-16)$ positions. In this way, $(\frac{u}{2}-16)$ guard-bits are introduced between the two independent products, thus allowing up to $2^{(\frac{u}{2}-16)}$ accumulations to be performed in SIMD fashion. The $\frac{K\times K\times T_M}{N_{DMAC}}$ u-bit data obtained in this way are then dispatched to the subsequent

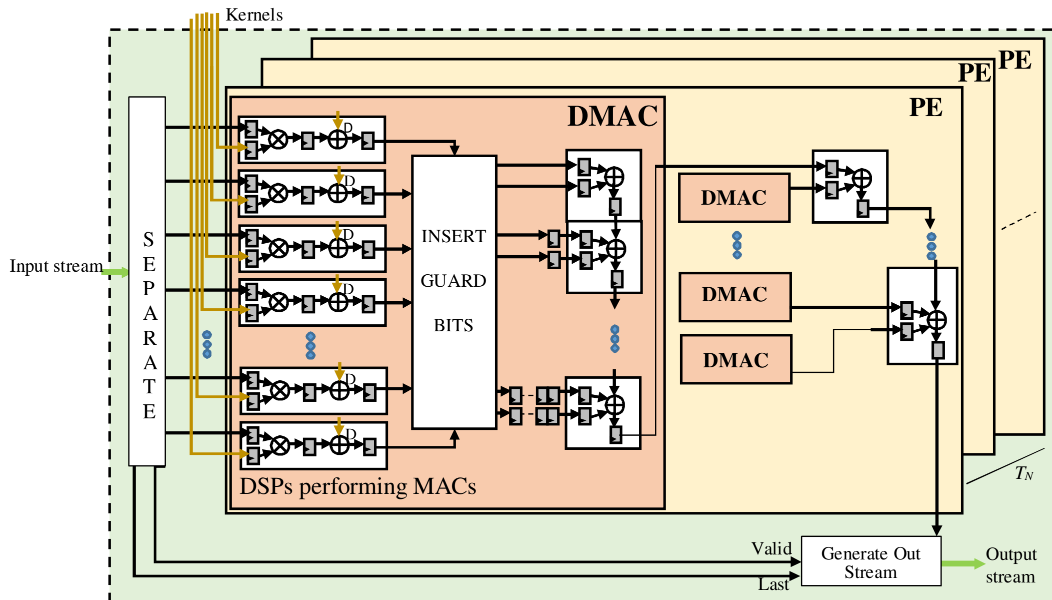


Figure 5.7 Circuit proposed for the SIMD CE.

$\frac{K \times K \times T_M}{2 \times N_{DMAC}}$ DSPs configured as accumulators. Two further cascaded DSP slices then accumulate the results produced in parallel by the four DMACs involved in the generic PE. Finally, taking into account the AXI4 protocol requirements, if $u<64$, the two $(\frac{u}{2})$-bit adjacent values packed in the output of the PE are sign extended to 32 bits; re-arranged within one 64-bit word; and streamed out, with the results coming from the others PEs.

5.2.3 Architecture of the ReLU and Pooling blocks

To produce the final *ofmaps*, the results furnished by the *SIMD CE* at the generic computational step must be accumulated to the previously computed ones. The module *Accumulate* has been purposely designed to this aim. As illustrated in Figure 5.8, it consists of T_N instances of the module ACC, each responsible for the computation of one *ofmap*. The signal *En* is kept asserted by the external *FSM* until the last computational step is performed, thus enabling the accumulations and the storage of the computed partial results within on-chip Dual Port BRAMs, whose read and write addresses are provided by two binary counters. The generic module ACC operates as follows: when valid data incomes from the *SIMD CE*, the local dual port memory is also accessed in reading mode; the 64-bit data read *Dout* is split into two 32-bit words to be accumulated through two parallel DSPs to the current input data. If the signal *En* is asserted, the accumulation results are re-packed within the 64-bit word *Din* and stored into the Dual Port BRAM for the next step. Conversely, when the last step is performed, *Din* is streamed out towards the next computational module to perform quantization and ReLU.

Figure 5.8 The *Accumulate* circuit.

Figure 5.9 shows that the *ReLU & Quantization* module designed for processing T_N *ifmaps* in parallel. More in detail, referring to the generic R&Q sub-module, it can be seen that each 64-bit input word is split within two 32-bit data to which the rectified activation function is applied. The ReLU is simply performed by using two multiplexers that, depending on the most significant bits of the 32-bit data received as input (e.g. their sign bits), set the outputs DM_0 and DM_1 equal either to the inputs or to zero. The 8-bit quantized results Q_0 and Q_1 are then produced in parallel and finally concatenated within a 16-bit word to be streamed out, as visible in Figure 5.2, towards either the external memory or the *Pooling* module.

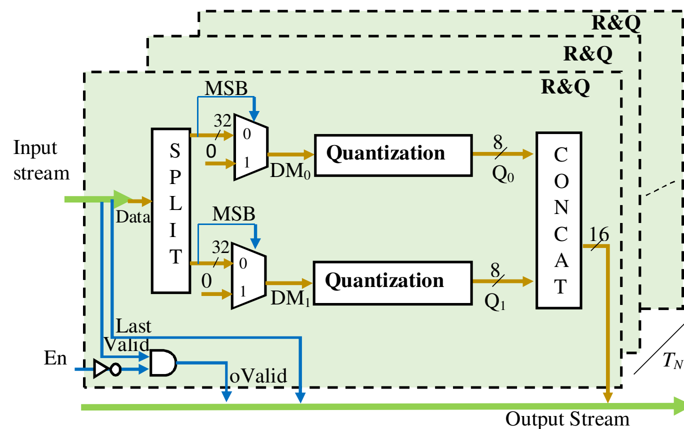


Figure 5.9 The ReLU & Quantization circuit.

The *Pooling* module depicted in Figure 5.10 can perform the downsampling by applying either the Max Pooling [147], or the Average Pooling [148], or the Stochastic spatial sampling [149]. Considering that each 16-bit data of the input stream corresponds to two adjacent values of the same *ifmap*, just two 16-bit registers are required to form the generic 2×2 downsampling window. The *FS*-1 depth FIFO is implemented using BRAMs addressed through the module Address Generator taking into account the feature map size *FS* that is changed by the external FSM in accordance with the current convolutional layer. When the downsampling window is ready, depending on the *Pool_type* parameter selector, also provided by the FSM to select the desired downsampling methodology, the 8-bit data to be used for replacing the 2×2 window in the *ifmap* is furnished. The *Pooling* module produces the first valid result after *FS+1* clock cycles and then furnishes a new output every clock cycle until two consecutive rows of the received *ifmap* are processed. During the subsequent *FS* cycles, the circuit just waits for the next downsampling window. Then, a new output value is produced at each clock cycle until two further rows are processed, and so on. The pooled *ofmaps* are finally streamed out towards the external memory.

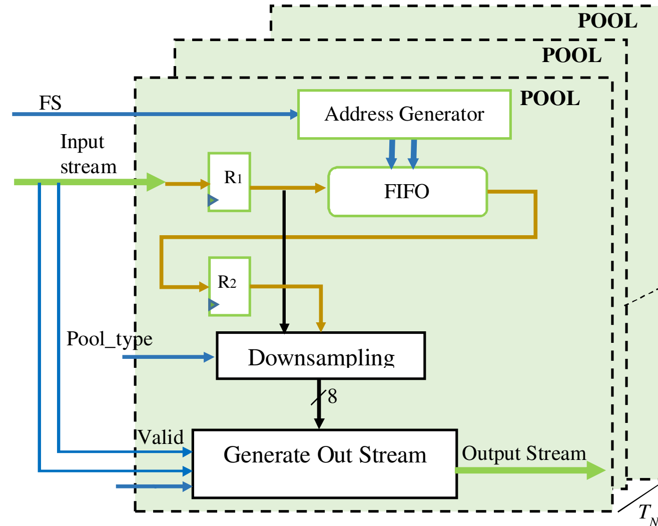


Figure 5.10 The *Pooling* circuit.

5.2.4 Software acceleration of FC layers and Softmax function

FC layers are executed by means of purpose-designed software routines run by the Processing System. The pseudo code reported in Figure 5.11 exploits the NEON multimedia engine [150] supporting SIMD and vector floating-point instruction sets. Such capability is available within both Xilinx [34] and Intel [33] SoC devices.

SIMD and vector instructions, as well as data types, supported by the NEON library allowed eight different computations in parallel to be performed. As an example, it can be seen that type *int32×4_t* is used to define a 128-bit packed word (e.g. *vecsum*), in which four 32-bit signed integer numbers can be accommodated to be processed in SIMD fashion. The generic FC layer computes the *Nc*-element array *simdsum*. To calculate its *j*-th 32-bit element, the proper kernels coefficients are transferred from the external memory to the 64-bit variable *Coeff* through the appropriate reading instruction (e.g. *Xil_In64* when Zynq-7000 devices are used). The uploaded data is re-arranged as eight distinct 8-bit signed coefficients through the instruction *vreinterpret_s8_s64*. Analogously, the *ifmaps* values are prepared in the variable *Ifmap8x8*. The instruction *vmull_s8* is then used to multiply within only one clock cycle corresponding elements in the packed words *Coeff8×8* and *Ifmap8×8*, thus assigning the eight 16-bit results to the corresponding elements of the variable *prod_q*. To complete the FC layer, the above operations must be repeated *nPC* times, as many as the 64-bit packets containing the kernels coefficients that must be read from the external memory, and each resulting *prod_q* must be accumulated to the previous ones. Obviously, more than 16 bits are required to correctly represent the accumulations

results. However, at most eight 16-bit data can be packed within the largest supported 128-bit word length. Therefore, in order to perform the subsequent accumulations

```
…
uint64x1_t Ifmap;
int64x1_t Coeff;
uint8x8_t Ifmap8x8;
int8x8_t Coeff8x8;
int16x8_t prod_q;
int16x4_t sum16x4, sum2_16x4;
int32x4_t vecsum={0,0,0,0};
int32x4_t vecsum2={0,0,0,0};
int32 simdsum[Nc];
int8x8_t simdsum8[Nc/8];
u16 nPC=Mc*Kc*Kc/8;

for (j=0; j<Nc; j++) {
   simdsum[j]=0;
   for (i=0; i<nPC; i=i+1){
            Read Coeff from the external memory
      Coeff8x8=vreinterpret_s8_s64(Coeff);

      //Required only the first FC layer
      Read Ifmap from the external memory
      Ifmap8x8=vreinterpret_u8_u64(Ifmap);

      prod_q=vmull_s8(Coeff8x8,Ifmap8x8);
      sum16x4=vget_high_s16(prod_q);
      sum2_16x4=vget_low_s16(prod_q);
      vecsum=vaddw_s16(vecsum,sum16x4);
      vecsum2=vaddw_s16(vecsum2,sum2_16x4);
    }
    simdsum[j]=simdsum[j]+vecsum[0]+vecsum[1]+vecsum[2]+
         +vecsum[3]+vecsum2[0]+ vecsum2[1]+vecsum2[2]+vecsum2[3];

    //ReLu and Quantization
    if (simdsum[j] < 0)
       simdsum8[j/8][j%8]=0;
   else simdsum8[j/8][j%8]=simdsum[j]>>24;
}
…
//Softmax applied to the output simdsum8 of  the last FC layer

float32x4_t vexp[125], L[125];
float32x4_t vexp2[125], L2[125];
float32x4_t sumexp;
float totsum=0;
for (j=0; j<125; j=j+1){
          for (i=0; i<4; i=i+1){
      vexp[j][i]=exp((simdsum8[j][i])>>nf);
      vexp2[j][i]=exp((simdsum8[j][i+4]) >>nf);
          }
   sumexp=vaddq_f32(vexp[j],vexp2[j]);
   totsum=totsum+sumexp[0]+sumexp[1]+sumexp[2]+sumexp[3];
}
for (u32 j=0; j<125; j=j+1){
  L[j]=vmulq_n_f32(vexp[j],1/totsum);
  L2[j]=vmulq_n_f32(vexp2[j],1/totsum);
}
```

Figure 5.11 Pseudo-code used to accelerate FC layers and the Softmax function.

correctly, the instructions *vget_high_s16* and *vget_low_s16* are preliminarily executed to split *prod_q* into two different variables *sum16×4* and *sum2_16×4*, each consisting of four 16-bit signed data. The instruction *vaddw_s16* is then executed to perform two separate accumulations each providing four 32-bit signed results packed within the variables *vecsum* and *vecsum2*. The eight 32-bit results obtained in this way are summed together providing the result *simdsum[j]*. The latter is then linearly rectified and quantized to 8 bits. Finally,

the packed variable *simdsum8* is furnished. The next FC layer is then performed executing the same instructions, excepted those highlighted in red, which must be executed only when the first FC layer is performed. Conversely, for any FC layer following the first one, the results of the previous layer contained in the variable *simdsum8* must be used as the input values. For the *i*-th element of the array *simdsum8* outputted by the last FC layer the classification likelihood is computed by applying the softmax equation given in (18), taking into account the number *nf* of fractional bits used in the fixed-point representation.

$$L(simdsum8[i]) = \frac{\exp\left((simdsum8[i])/2^{nf}\right)}{\sum_{j=1}^{Nc3} \exp\left((simdsum8[j])/2^{nf}\right)} \quad (18)$$

5.3 RESULTS

5.3.1 Implementation

The proposed accelerator is platform independent. However, without loss of generality, in this subparagraph the Xilinx Zynq-7000 devices are referred as target, as they have been used in the prototyping phase of this research work.

Figure 5.12 illustrates the block diagram of the resultant heterogeneous embedded system. The PS Master General Port M_GP0 is used to configure all subsystems realized in the PL. The latter also directly accesses the DDR memory controller available within the PS by means of the four 64-bit bidirectional High Performance (HP) ports. Furthermore, the Accelerator Coherency Port (ACP) makes the PL able to perform 64-bit coherent accesses to the DDR memory space by means of a Snoop Control Unit (SCU). The four HP ports manage the *ifmap/ofmap* data flow from/to the external DDR memory by means of four DMAs, while the ACP provides kernels coefficients data transfers through a CDMA.

In particular, the system performs the following main activities:

- the software running on the PS uses the port M_GP0 to configure the DMAs and the CDMA IP cores through the AXI4-Lite protocol. Each one receives an appropriate task to transfer a certain amount of data from/to a specific area within the external DDR memory. The port M_GP0 is also used to configure the accelerator by setting the number and the size of *ifmaps* and *ofmaps* for each layer of the accelerated CNN, as well as the type of pooling to be applied, then starting its operations;

- the AXI-Streams coming out from the four DMAs are synchronized by the AXIS-Combiner within a single data stream; contemporaneously, the CDMA transfers the kernels coefficients related to the current convolutional layer from the DDR to the *Store Kernels* module;
- the combined stream is purposely split by the AXIS-Broadcaster$_0$ into T_M separate streams to sustain the parallelism level on buffered *ifmaps* delivered to the CNN accelerator;
- the output stream produced by the CNN accelerator is then separated into four 32-bit streams by the AXIS-Broadcaster$_1$ and moved to the external DDR by DMAs, thus properly preparing the *ifmaps* for the next convolutional layer;
- the software routine run by the PS finally performs the FC and Softmax function.

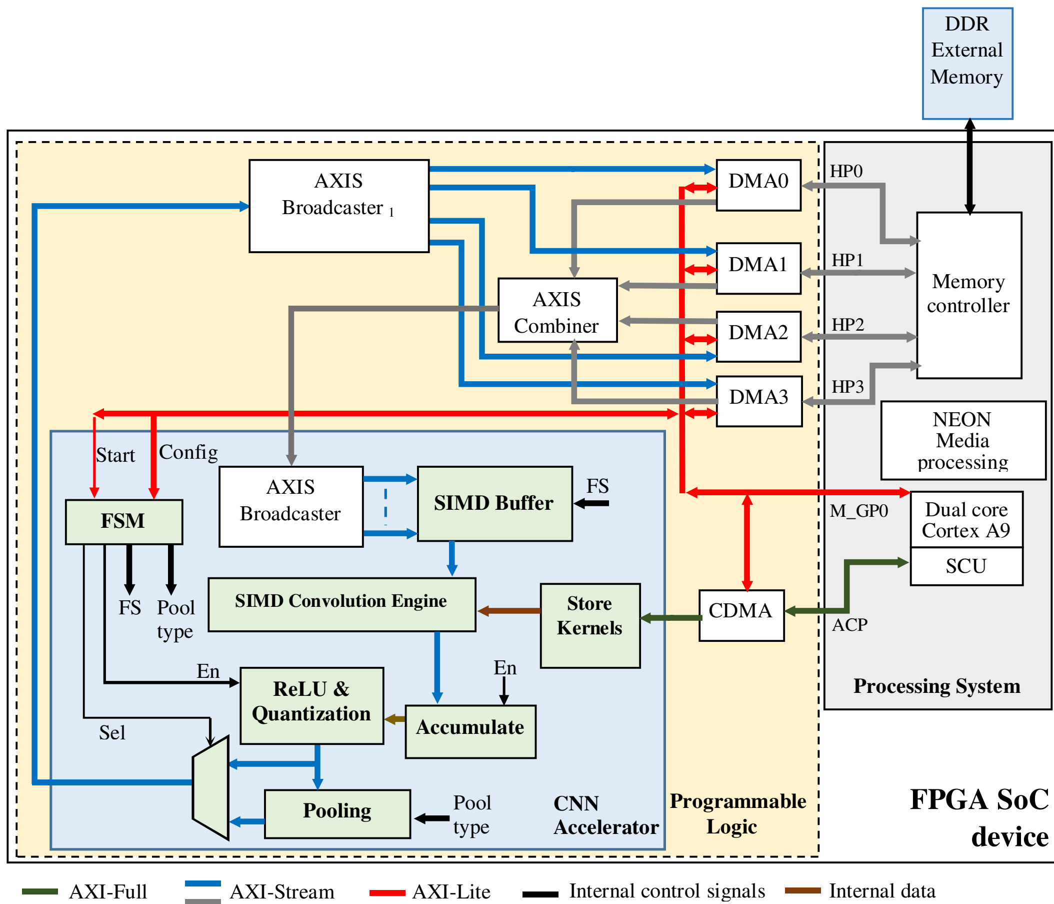


Figure 5.12 The top-level architecture of the proposed embedded system on Zynq-7000 devices.

Maximum sustainable memory bandwidth and resources available in the XC7Z020 and XC7Z045 devices, chosen as the targets for hardware prototyping, limit the level of parallelism of the accelerator. The implementations presented in the following are able to process T_M=8 *ifmaps* in parallel. The cheaper XC7Z020 can sustain a parallelism level on *ofmaps* T_N=2. In this case, the CNN accelerator furnishes in parallel two adjacent 8-bit

values for each produced *ofmap*, thus the results can be packed within one 32-bit data stream and then transferred towards the DDR memory involving only one of the instantiated DMAs also for writing operations. Moreover, the total bandwidth requirement is 2.9GB/s at 150MHz that is the maximum DMA performance on the chosen device. This transfer rate is well below the maximum 4.16GB/s supported by DDR memory controller [82]. Conversely, the wider XC7Z045 device makes available an amount of DSPs and BRAMs sufficient to increase T_N to 8 and its DDR controller provides a maximum bandwidth as high as 5.2GB/s [82] at 167MHz.

5.3.2 Experiments on the VGG-16 model

Table 5.2 summarizes resources requirements, computational capabilities and performances achieved by the implementations here presented and by several state-of-the-art competitors when the design is targeted to accelerate the VGG-16 CNN model [132]. This model performs 13 convolutional layers, all using 3×3 kernels, 3 fully connected layers and the Softmax function as defined in (18) to complete the image classification task.

From data reported in original reference papers, it can be concluded that the systems proposed in [54] and [126] achieve the lowest power consumption. However, the time needed to perform the complete CNN elaboration is well above 1 second. Conversely, among the compared competitors, the architecture demonstrated in [127] reaches the highest throughput. It requires only 43.2ms to complete its task and uses more than 1500 DSPs running at 200MHz on an Arria10 GX1150 device.

Table 5.2 is divided into two portions: in the former, architectures realized on low-cost devices are collected; whereas, the latter summarizes the behaviors of prototypes implemented on high-end chips. All the competitors are characterized in terms of resources requirements, number of operations performed per second (Gops), Density Efficiency (DE), times required to execute CONVs, FC layers and Softmax function, and Power Efficiency (PEff). For the sake of fair comparisons, it is worth underlining that the architectures presented in [52], [54] and [129] are proposed as standalone accelerators. Thus, they do not take into account either hardware resources required for the integration within a complete embedded system, or the time needed for transferring data from/to an external memory and the latency needed for properly buffering input data. Moreover, the designs proposed in [53] and [129] execute only the convolutional layers.

In both sets of competitors, the proposed architecture shows the most favorable performance-power trade-off, reaching the highest PEff. Among the XC7Z045 implementations, that proposed here also exhibits the highest DE. The standalone design

presented in [129] achieves a DE just 13% lower, but at the expense of FFs and BRAMs requirements 22% and 48% higher.

Table 5.2 Characterization of the proposed CNN embedded system compared to most relevant prior works.

	Design	Frequency [MHz]	Gops	DSPs	Other resources			DE (Gops/ DSPs)	CONVs Time [ms]	FCs+Softmax Time [ms]	PEff (Gops/W)
					LUTs	FFs	BRAMs [Mb]				
New ($T_M=8$, $T_N=2$) XC7Z020	ES[1]	150	95.5	220 (100%)	13455 (25.3%)	19129 (18%)	3.44 (70%)	0.434	376.3	48	38.5
[52] XC7Z020	SA[2]	150	84.3	190 (86.3%)	29867 (56%)	35489 (33%)	3 (61%)	0.443	364	NP[4]	24.1
[53] XC7Z020	ES	125	48.53	220 (100%)	NA[3]	NA	NA	0.22	633	NP	27.7
New ($T_M=8$, $T_N=8$) XC7Z045	ES	167	425.32	880 (97.8%)	30161 (13.8%)	47832 (10.9%)	12.9 (67.5%)	0.48	84.5	48	135
[54] XC7Z045	SA	150	36.8	197 (21.8%)	18578 (8.5%)	8049 (1.84%)	0.773 (4%)	0.186	1639.3	NP	79.7
[52] XC7Z045	SA	214	137	780 (86.6%)	182616 (84%)	127653 (29%)	17.08 (87%)	0.175	224.6	NP	14.2
[53] XC7Z045	ES	125	155.81	855 (95%)	NA	NA	NA	0.182	249.5	NP	38.8
[129] XC7Z045	SA	150	374.98	900 (100%)	113672 (52%)	240640 (55%)	19.16 (100%)	0.416	82.03	NP	NA
[128] XC7Z045	ES	140	169	864 (96%)	88154 (35.1%)	61250 (14.1%)	11.25 (58.7%)	0.195	181.8	72.6	16.9
[126] XC7Z100	ES	60	17.2	128 (6.3%)	229000 (83%)	107000 (19%)	13.6 (51.1%)	0.134	2269*	*included	27.4
[127] GX1150	ES	200	715.9	1518 (100%)	141312 (32%)	NA	43.6 (82%)	0.47	43.2*	*included	NA

[1] Embedded system; [2] Standalone accelerator; [3] Not available; [4] Not performed

As detailed above, in the realized accelerator, FC layers and Softmax are perfomed by the PS. The 666MHz ARM Cortex A9 processor, provided with the NEON multimedia engine, performs these operations within an overall time of ~48ms. It has been proven that the SIMD approach detailed in 5.2.4 requires a computational time ~47% lower with respect to the conventional non-SIMD implementation. In comparison with the embedded system described in [128], the proposed architecture achieves a density efficiency more than twice, and it executes the convolutional layers and the subsequent FC and Softmax within times ~53% and ~34% lower, respectively. The above advantages are obtained by using ~43% and ~22% less LUTs and FFs, respectively, with comparable amounts of DSPs and BRAMs.

5.4 SUPPORTING NON-UNIFORM KERNEL SIZES

Characterization of the proposed SIMD accelerator for CNNs has been made taking into account a uniform kernel size across the layers, e.g. VGG-16 performs only 3×3 convolutions. However, state-of-the-art CNN algorithms feature cascaded CONV layers characterized by different kernel sizes and feature map dimensions [133]-[135]. This leads the design of efficient accelerators to be more complex. Several previous works [52] [151]-

[154] implemented a single hardware coprocessor optimized for the collective performances of all CONVs within a specific CNN model. As detailed in the previous paragraphs, the typical scheme consists of **i)** an input buffer to transfer *ifmaps* values to the subsequent processing block; **ii)** a convolutional engine, responsible for multiple parallel convolutions on a set of T_M *ifmaps* to calculate T_N *ofmaps* values; **iii)** an output buffer, suitable to store the T_N *ofmaps* before the next operation. Among these blocks, the convolutional engine could be affected by the different kernel dimensions involved in a CNN model, whereas the input and output buffers mainly suffer the different feature maps dimensions. Moreover, if the input buffer is used to prepare sliding windows required for the convolution, its design must also take into account the kernel size. It is then clear that an accelerator purposely designed to accomplish operations within a specific layer is not necessarily suitable when different convolution sizes are involved [155]. For instance, the design presented in [52] performs 84.3 GOPs on the uniform VGG-16 [132] with 8-bits quantization, while the throughput decreases down to 41.1 GOPs for CNNs that work with different kernel sizes across the layers.

Recent studies [155]-[156] observed that multiple hardware co-processors, each purposely designed to support a different layer, perform better than a single generic engine. However, such a strategy introduces cost overhead and increases the number of accesses to the off-chip memory. To solve this problem, a Multi-Convolutional Layer Processor (M-CLP) can be organized as a single engine divided into segments [155]. The generic CLP within the generic segment sequentially processes some layers, each having its own independent data. The segment ends its computation when all CLPs complete the processing of their layers. Similarly, in [156] a layer-folding structure is exploited for the convolutional engine. However, this strategy requires that foldable layers are adjacent within the CNN model and they must have the same kernel dimension. The design strategy demonstrated in [157] makes a fixed co-processor able to manage different kernel sizes by excluding the kernel size K from the parallelism levels. In this case, the convolutional engine surely supports all possible values of K, but at the cost of a detrimental effect on the number of cycles per *ofmap* value.

As shown in [158], the runtime reconfiguration also represents a valid alternative to make a parallel convolutional engine able to work with different kernel sizes. There, two general-purpose cores implemented within the FPGA fabric were used to reconfigure the accelerator, although a dual-core ARM processor was available within the realization platform. Furthermore, the design presented in [158] supports just 3×3 and 5×5 kernels and it was not adaptable to the scaling applied on the feature maps by the pooling.

5.4.1 The proposed reconfigurable design

The accelerator presented in this paragraph allows processing differently sized *Conv* layers by efficient reconfigurable computational patterns. In other words, the proposed convolution engine is made able to adapt itself to the different arrangements with which convolution windows are processed depending on the current dimensions *W* (i.e. the *ifmaps* width) and *K*. This is translated in an effective T_M-T_N parallelism that scales as the sizes involved in the current processing increase.

In order to identify the most favourable reconfiguration pattern, a design analysis has been conducted focusing on bandwidth requirements and throughput achievable for several kernel sizes (*K*=3, 5, 7, 9, 11) and T_N-T_M pairs. As shown in Figure 5.13, where the 200MHz operating frequency is taken as target, for almost all the cases the bandwidth *B* required to transfer T_M (T_N) *ifmaps* (*ofmaps*) from (to) the external memory ranges from 5 to 16 GBytes/s. Critical exceptions are the 3×3 configurations where one among T_M and T_N is set to 64, which would sustain a 27.42 GBytes/s bandwidth. It is important noting that the requested bandwidth plays an important role in the scenario of embedded systems for visual IoT. In such a context, the energy cost related to data transmission strongly impacts on the overall consumptions, therefore minimizing the bandwidth demand represent a key point for the design of efficient accelerators. However, reducing the amount of data that is transferred in parallel to the convolutional engine has a negative influence on the number of Giga operations executed at a time (GOPs/s). Focusing on the latter, Fig. 5.13 suggests that performances achieved when $T_N > T_M$ are better than the case $T_N < T_M$. Based on such a consideration, for each evaluated kernel size, the highest GOPs/s/*B* value has been searched to identify the best compromise among performances and bandwidth requirement.

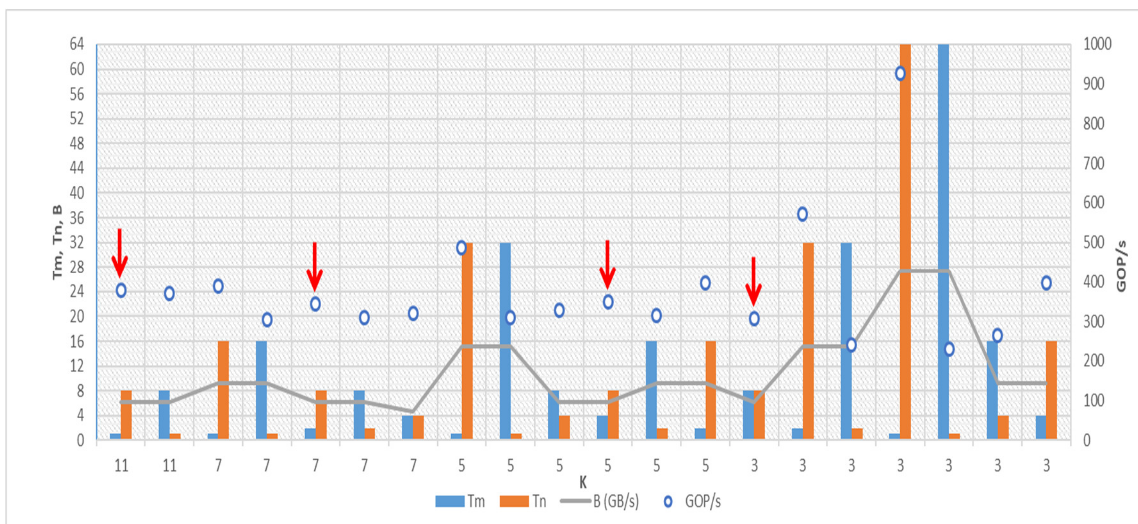


Figure 5.13 Throughput and bandwidth analysis for different kernel dimensions.

The optimal configurations, marked with red arrows in the graph of Figure 5.13, have been used to characterize the novel reconfigurable convolution engine: T_N=8, T_M=2 for k=7; T_N=8, T_M=4 for k=5; and T_N=8, T_M=8 for k=3. It is important clarifying that the larger 11×11 kernel size, being exploited only by obsolete CNN models like AlexNet [122], is not taken into account in the following description.

The possibility of reconfiguring the input buffer depending on the current *ifmaps* size has been already proved within the SIMD accelerator. In addition, the novel circuit create the current set of convolutional windows by supporting kernel sizes ranging from 1 to 7.

The novel run-time reconfigurable buffer is depicted in Figure 5.14. Input pixels are streamed to the *RegArray* block, which also receives the kernel size, identified by the input signal *kSize*. Internally, the *RegArray* contains 50 16-bits registers that can be properly packed to store one 16-, two 32- or four 64-bit words, respectively, depending on the current value of *K*. In such a way, the same architecture is reused by adapting the number of pixels processed at once to the kernel size (i.e. the larger *K*, the lower the number of pixels transferred in parallel to the convolutional engine). For the proposed reconfigurable computational pattern, all the registers within the *RegArray* block are used when *K*=5; conversely, for *K*=7 and *K*=3 just one and 14 16-bits registers are bypassed, respectively.

As visible in Figure 5.14, each line of registers is fed by the data coming from the *Line Shift Buffer* that is properly sliced and multiplexed to the correct package of registers. Similarly, the output of each line is transferred to the *Line Shift Buffer*, after being concatenated. The *Line Shift Buffer* is a single block RAM that, based on *K* and the *ifmaps* size *W*, emulates the behaviour of *K*-1 line buffers each *W*-*K* deep. This is made possible thanks to read and write pointers internally managed by the architecture through a proper counter. These pointers provide the read and write addresses through which proper memory locations of the block RAM used in *Line Shift Buffer* are accessed sequentially with the FIFO policy. As reported in Table 5.3, with respect to the traditional design of the line buffer composed by *K*-1 FIFOs, the adopted strategy saves up to 72kbits of on-chip

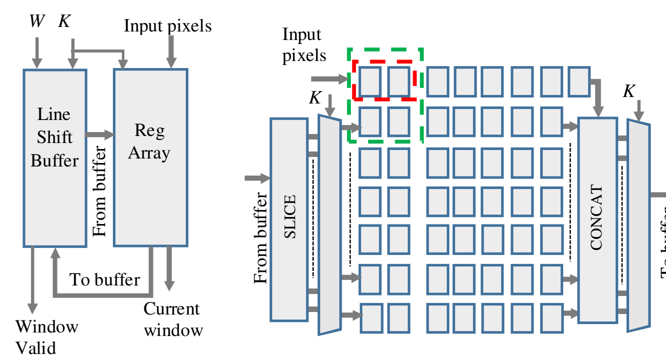


Figure 5.14 Schematic diagram of the proposed reconfigurable buffer for non-uniform kernel sizes across CNN layers.

memory for feature map sizes equal to 224; in such a case, the underutilization factor (i.e. the percentage of bits actually used with respect to the total number of bits that the implemented block RAMs make available) is reduced by up 19%.

Table 5.3 Underutilization analysis for different design strategies of the input buffer for W=224.

<table>
<tr><th rowspan="2">Strategy</th><th colspan="3">Theoretical Requirements (kbits)</th><th colspan="3">Actual Requirements [1] (kbits)</th><th colspan="3">Underutilization Percentage</th></tr>
<tr><th>k=3</th><th>k=5</th><th>k=7</th><th>k=3</th><th>k=5</th><th>k=7</th><th>k=3</th><th>k=5</th><th>k=7</th></tr>
<tr><td>Traditional</td><td rowspan="2">27.62</td><td rowspan="2">27.37</td><td rowspan="2">20.34</td><td>144</td><td>144</td><td>108</td><td>80.8%</td><td>81%</td><td>81.16%</td></tr>
<tr><td>Ours</td><td>72</td><td>72</td><td>72</td><td>61.6%</td><td>62%</td><td>71.7%</td></tr>
</table>

[1]Number of implemented BRAMs × 36kbits

The top-level architecture of the proposed *CE* with reconfigurable computational pattern is shown in Figure 5.15(a). Basically, it consists of four *ConvMAC* blocks running in parallel, whose outputs are transferred to a chain composed by just two DSP slices to perform the final accumulation. Each *ConvMAC* block receives a specific set of filters and the current window for T_M/4 *ifmaps*. This means that the CE processes up to 8, 4 and 2 *ifmaps* at a time for the K input 3, 5 and 7 respectively.

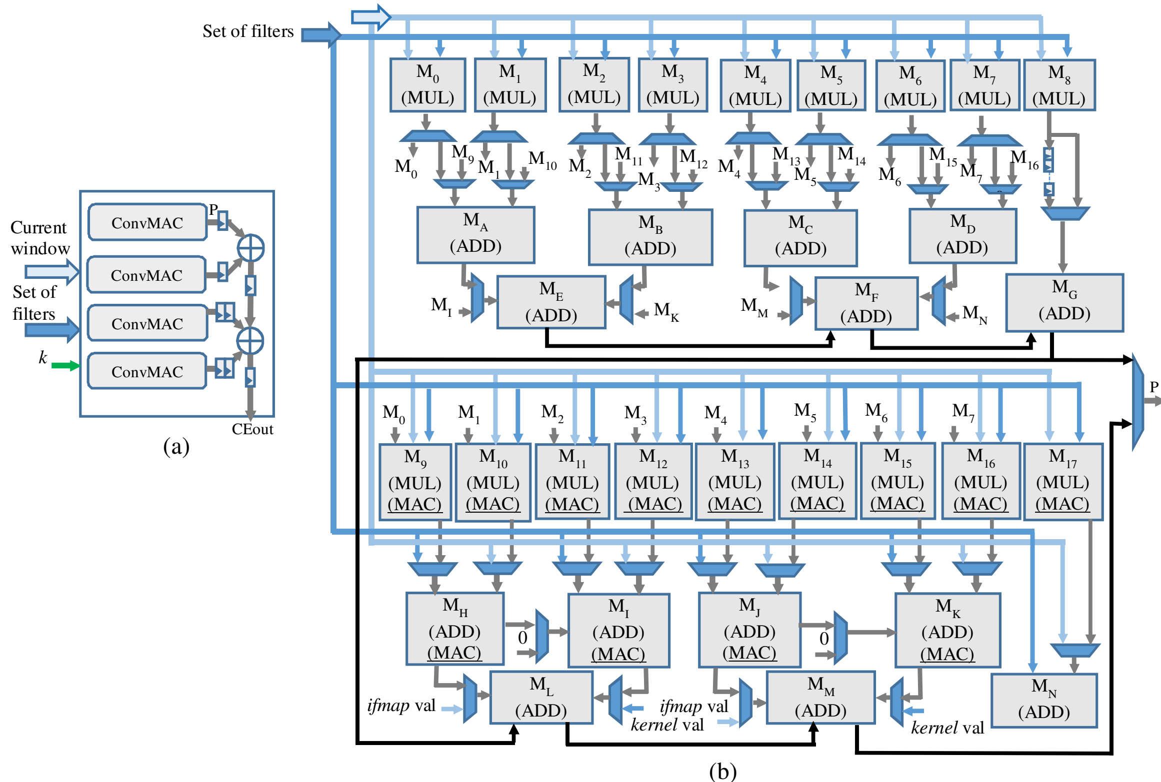


Figure 5.15 Design of the novel reconfigurable CE. (a) the top-architecture; (b) details of the ConvMAC block.

The detailed design of the *ConvMAC* block is illustrated in Figure 5.15(b). Basically, two main working modes are possible depending on the current K input. The former is enabled when K=3. In this case, all the DSP blocks DSP_0-DSP_{17} perform MAC operations

on proper kernel coefficients and convolutional widow values, whereas the DSP slices DSP_A-DSP_P are used just as adders. In this case, DSP_A DSP_B DSP_C DSP_D DSP_H DSP_I DSP_L and DSP_M compose the first level of an adder tree. The outputs produced by such a first level are sent to DSP_E DSP_F DSP_G DSP_N DSP_O DSP_P that are cascaded connected. Therefore, the output P will correspond to the data produced by DSP_P. Implementing a fully-pipelined architecture, the *ConvMAC* block configured for K=3 exhibits an initial latency of 16 clock cycles.

When K=5, DSP_0-DSP_{17} continue to perform MACs with properly dispatched kernel and *ifmaps* values, but in this case DSP_9-DSP_{16} receive also the outputs produced by DSP_0-DSP_7 (highlighted in red in Figure 5.15(b)) to be added with the result of the current MAC operation performed by DSP_9-DSP_{16}. The outputs produced by DSP_9-DSP_{16} are inputted to DSP_A-DSP_D in pairs. It is worth noting that, in such a configuration, DSP_H-DSP_P execute MACs, instead of simple additions, whose results are finally accumulated by the chain composed by DSP_E-DSP_F-DSP_G. This computational pattern allows performing 5×5 multiplications in parallel, using the remaining DSP slices to add intermediate MAC results and achieving an initial latency of just 12 clock cycles for the single *ConvMAC* block.

Concerning the computational pattern reconfigured for K=7, the architecture as illustrated in the previous case is still valid. The main difference is related to DSP_{17} that is keep in idle for the first two *ConvMAC* blocks composing the cascaded *CE* in Figure 5.15(a). This allows performing a total number of MACs equal to 98 (i.e. 24+24+25+25) within the *CE*, as required to process 7×7 convolutional windows sliced from two *ifmaps*. Clearly, in such a case the initial latency of the *ConvMAC* block is again 12 clock cycles.

In order to verify the correct working mode of the above described reconfigurable modules, a complete embedded system to accelerate CNN inference has been implemented on heterogeneous SoCs. It is worth noting that the proposed reconfigurable design is platform independent. Furthermore, the proposed design is independent from the precision used to represent feature maps and weights. During the prototyping phase we chose 8-bits fixed-point quantization for both weights and feature maps that, as above stated [143], demonstrates an acceptable accuracy compared to the 32-bit floating-point representation.

The block diagram of the referred heterogeneous embedded system is very similar to that described in the previous paragraph. Basically, as illustrated in in Figure 5.16, the PL implements the custom hardware circuits to accelerate the operations typical of a CONV layer. Due to the high amount of data, *ifmaps* and weights values are stored in an external memory, which is accessible by the memory controller integrated within the PS section. Two DMA IP cores are used to connect the PL with the memory controller, avoiding the processor action. The ARM processor transfers specific parameters of the current processed CONV layer to the *Config* IP core. The latter orchestrates the operations on the PL: it

receives the number of *ifmaps* (*ofmaps*) to process (to produce), as well as the kernel size and the dimension of *ifmaps* used for that layer, and provides the *CTRL* signals to the reconfigurable modules.

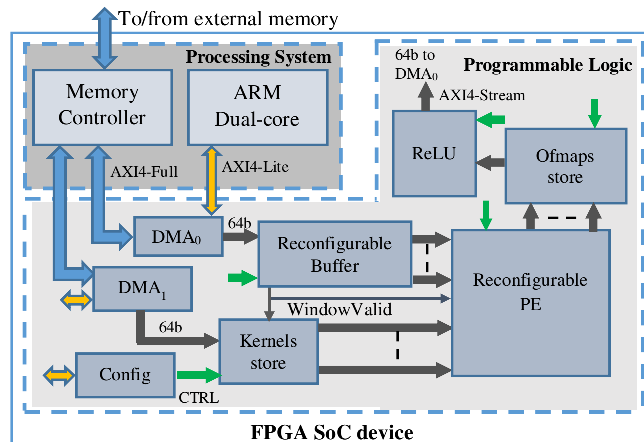


Figure 5.16 The heterogeneous embedded system accommodating the proposed reconfigurable buffer and CE.

The set of T_N filters needed to compute a block of T_M *ifmaps* is transferred to the block *Kernels store* by DMA_1; these weights will be inputted to the *Reconfigurable Processing Element* (*PE*) only when the *WindowValid* signal is driven high by the buffer. DMA_0, transmits the *ifmaps* values as 64-bits words to the reconfigurable buffer. Clearly, these 64 bits may be organized in several modes, based on the effective T_M parallelism. As an example, for T_M=4, up to two adjacent 8-bits values of the four *ifmaps* can be arranged within each 64-bits word. In this way, the time required to transfer input data is halved. The reconfigurable buffer, after its internal latency, forms the convolution windows that will be dispatched to the *PE*, accompanied by the *WindowValid* signal. The *Reconfigurable PE* will contain T_N identical CE blocks each structured as illustrated in Figure 5.15(a). The T_N output streams thus produced are sent to the *Ofmaps store* module for accumulation. In this case, after the first T_M set has been processed, the next T_M *ifmaps* are read from the external memory, and the *PE* results are accumulated with those obtained in the previous step by a pixel-wise sum. It is important to observe that the *Ofmaps store* also receives the *CTRL* signals *CTRL*, though which the IP core establishes if it has to continue with the accumulation or the last step is occurred. In the latter case, the *Ofmaps store* module transfers the final results to the *ReLU* module that applies the homonymous activation function. Finally, within the *ReLU* block, results are quantized to 8-bits, packed within a 64-bit word, taking into account an eventual pooling mode, and transferred to the external memory by the write channel of DMA_0. It is worth highlighting that the read channel of DMA_0, as well as DMA_1, can access the external memory to read *ifmaps* and weights before the *ReLU* core has accomplished its task, thus allowing next transfer times to be hidden.

5.4.2 Results

Table 5.4 summarizes implementation results. The whole embedded system depicted in Figure 5.16 achieves a maximum running frequency of 195 MHz. Designs referenced as competitors implement the strategies described in Section 5.4 to support different convolutional layers. For purposes of comparison, the efficiency metric DE (GOPs/s/DSPs) is used. It can be easily observed that the proposed embedded system exhibits a DE up to ×2.64 higher than the other accelerators. However, a brief discussion about other implementations is mandatory. As the accelerator in [157] does not exploit the kernel-level parallelism, relatively low performances are reached. Similarly, the FPGA resources are not fully exploited in the HLS-based design proposed in [159]. The architectures in [155] and [156] achieves a throughput higher than this work, but they also show lower DEs due to the implementation of multiple CLP within the same CE, which strongly impacts also on the logic resources occupancy and bandwidth requirements. The complete embedded system presented in [158] makes also use of a considerable number of LUTs, probably related to the logic required to switch from 3×3 to 5×5 kernel mode. The design proposed in [52] achieves a DE relatively close to [155]; however, it is fair pointing out that the throughput of 187.8 GOPs/s declared in [52] represents a peak performance reachable only when the accelerator has to perform CONV layers with 3×3-sized kernels. Furthermore, both [52] and [155] refer to a standalone implementation, thus respective resource counts don't take into account auxiliary modules needed in modern heterogeneous embedded SoCs for implementing the communication among PL, PS and external memory.

Table 5.4 Characterization of the proposed heterogeneous embedded system with reconfigurable computational patterns and comparison with prior works.

	Platform	LUTs	FFs	BRAMs [Mbits]	DSPs	f_{max} [MHz]	GOPs	DE	Power (W)	GB/s
New	XCZU9EG	49022	85340	13	1048	195	350.4	0.33	4.8	6.09
[52][1]	XC7Z045	182616	127653	17	780	150	187.8	0.24	13.2	-
[155][1]	Virtex-7	133854	161411	19.5	3494	170	909.7	0.26	7.2	19.5
[156]	Virtex-7	317846	266252	16.25	2843	137	593.5	0.2	-	12.4
[157]	Virtex-7	78318	96929	12.5	1034	150	129.7	0.125	18	4.2
[159]	XC7Z045	52464	87440	12.8	369	150	79.63	0.2	4.2	3.8
[158]	XC7Z045	106036	97585	6.75	874	150	181.4	0.2	8.6	2.34

[1] Standalone designs.

In order to establish which solution achieves the best traded-off between performances, resources requirements and power consumption, the Figure of Merit (FoM) defined in (19) is introduced.

$$FoM = \frac{100 \times DE}{kbits \times Power} \tag{19}$$

In (19), the amount of kbits accounts FFs, BRAMs and LUTs. The single contributions are obtained counting each flip-flop as one bit; each Mbit as 1024 kbits for the BRAMs; and each *Ninput* LUT as 2^{Ninput} input bits. Results, illustrated in Figure 5.17, highlight that the system accommodating the novel reconfigurable accelerator achieves the best compromise, with a FoM up to 90% higher than prior works. Unfortunately, [156] has been excluded from this comparison since no information is available on its power consumption. However, the resources occupancy suggests a non-negligible power consumption that, joint to a DE of just 0.2, would leads anyway to a FoM value lower than this work.

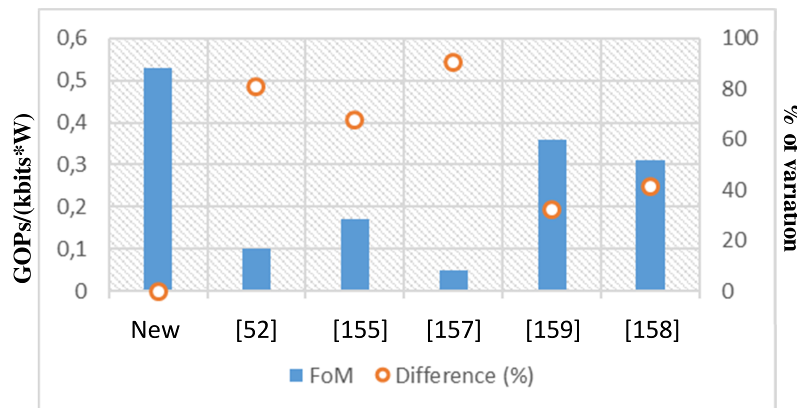


Figure 5.17 State-of-the-art comparison for performances, power and resources occupancy.

The embedded system illustrated in Figure 5.16 has been tested by the execution of VGG-16 [132] and VGG-S [135] models. Since VGG-16 adopts only 3×3 kernels, the accelerator always works with T_N=8 and T_M=8, completing all the convolutional layers within 140.2 ms. In such a case, the achieved frame rate, considering only the 13 CONVs of the network, is 14.3%, 65.7% and 34% higher than [52], [159] and [158] respectively, but 65.4% lower than [156]. This is because the layer-folding strategy adopted in [156] permits a pipeline mode between cascaded folders, which leads to significant hardware requirements, as highlighted in Table 5.4. When the proposed design is used to accelerate the VGG-S model, having different kernel sizes across the layers, the execution time for CONVs is 16.7 ms, which corresponds to a processing rate of about 60 frames per second.

5.5 SUMMARY

In an edge computing perspective, implementing the inference of convolutional neural networks within an IoT node is a hard challenge. The main issue is related to the selected processing platform, which would satisfy all the constraints, in terms of power, performances and sizes, imposed by typical IoT applications. Although heterogeneous

FPGA SoCs have been recently recognized as a promising solution, and considerable efforts have been invested in this research topic, state-of-the-art implementations are still inefficient to be integrated within smart visual IoT nodes. In this chapter, a novel SIMD hardware architecture has been presented for accelerating the inference of reduced precision large-scale convolutional neural networks by doubling the number of MAC operations performed by the single DSP slice. The SIMD architecture has been accommodated within a complete heterogeneous embedded system, implementing an effective hardware/software partitioning. Two implementations of the proposed embedded system have been characterized for the Xilinx Zynq-7000 SoC family. Such prototypes outperform state-of-the-art implementations based on the same embedded platforms. The architecture here presented exhibits a resources efficiency higher than competitive prior works, thus representing a good candidate for the realization of resource-constrained high-performance intelligent systems. The proposed strategy allows the accelerator to be easily configured for various kernels sizes and parallelism levels, and allows reaching for the VGG-16 model a frame rate of 2.65fps and 11.8fps on the XC7Z020 and XC7Z045 device, respectively.

In addition, a novel design, exploiting reconfigurable computational patterns to support inference of convolutional neural networks having different kernel sizes across layers, has been introduced. Results in terms of hardware requirements and GOPs/s have demonstrated the efficiency of the proposed architecture. For a fair comparison with relevant prior works, a figure of merit (FoM) has been calculated taking into account performances, power and area. The proposed design exhibits a FoM up to 90% higher than competitors, thus confirming its suitability for edge computing IoT applications.

6 CONCLUSIONS

The Internet of Things revolution is driving the development of novel technologies for collecting and processing a massive amount of data. In Chapter 2, we surveyed requirements and challenges that characterize modern IoT sensor nodes. These can be resumed in the following key points:

- in the context of visual sensing, where data captured by camera sensors provide more information in comparison to mono-dimensional counterparts, the problem of network congestions is already a reality. To reduce the traffic of raw data, new techniques are currently explored for embedding computation capabilities within the node. However, due to the stringent constraints, in terms of both energy and performances, the design process of smart visual IoT devices requires a significant effort;
- the choice of the most appropriately computing platform strictly depends on several factors, such as speed performance, energy efficiency, size, cost and application flexibility. Fully application-specific integrated circuits provide more competitive energy and performance features compared to traditional CPU- and GPU-based counterparts. Unfortunately, the fabrication cost of these chips increases with every technologic node, making them less convenient when specialized hardware designed for advanced nodes is the target. Moreover, pure ASIC solutions lack of flexibility, which is another relevant requirement in the IoT perspective;
- as deep learning is becoming a mainstream candidate to solve problems related to the large amount of data traffic, designing efficient architectures able to accomplish all the constraints above discussed is getting more and more interesting. However, recent deep learning algorithms come with high (and growing) computational costs, which strongly impact on both energy and performance. In such a perspective, innovative processing techniques become the main way forward for bringing intelligence close to IoT sensors.

The results presented in this dissertation demonstrate that modern FPGA-based heterogeneous SoCs, exploited in conjunction with novel designs and computing techniques, are a powerful solution to enable visual IoT at the edge. We have proposed four different examples of heterogeneous systems for various computer vision applications, ranging from stereo-vision and integral images computation (Chapter 3), to connected component analysis (Chapter 4) and convolutional neural networks (Chapter 5). In all the

presented cases, speed performances and/or energy efficiency are considerably improved with respect to state-of-the-art solutions. From a system level point of view, it is worth noting that the heterogeneous implementations above described are area-effective and they adopt a low number of I/O pins (mainly used for communication with possible external cameras, as in the case of stereo-vision). Such an outcome suggests that FPGA SoCs cheapest than those exploited in this dissertation could be employed, introducing further benefits in terms of both power consumption and physical integration, which are key requirements in portable IoT systems. Indeed, even though hardware characterizations described in previous chapters refer to Xilinx Zynq-based SoCs, the methodologies proposed in this thesis, being platform-independent, can be easily extended to other FPGA families, such as those provided by Intel [160]-[161]. Moreover, different heterogeneous architectures, even based on standard-cell ASICs, can be explored for implementation of low-power and high-performance IoT nodes based on the design line guides here proposed. In this case, implementation of specific functionalities, such as the communication between the SoC and the external memory, would require dedicated solutions to be developed.

Apart the technical results, a fundamental aspect when exploiting heterogeneous platforms is related to the flexibility-efficiency trade-off. In such a context, partitioning software and hardware tasks becomes the most critical step in the design process. Depending on the specific applications, different strategies can be adopted, as we stated in previous chapters. In some cases, the flexibility arises by just exploiting a general-purpose processor to runtime reconfigure the hardware accelerators, thus ensuring similar speed and energy performances to be kept in different application scenarios. In other cases (i.e. the proposed SIMD CNN accelerator), the general-purpose processor is also used as computing unit, coupled with hardware accelerators, to execute specific tasks. In both situations, we have demonstrated that the heterogeneity introduced in the proposed systems is extremely advantageous for energy efficiency. Obtained results suggest that contributions of this dissertation is significant with respect to extend the utilization of such a class of heterogeneous platforms in all application fields where high-performance and low-power smart sensor nodes are involved.

6.1 FUTURE PERSPECTIVES

We believe there is still room for improvements building upon the results presented in this thesis. As an example, in Chapter 5 we presented a solution for accelerating inference of deep CNNs where a reduced precision (i.e. 8 bits fixed-point) is exploited. Such a technique provides appreciable results in terms of both energy efficiency and speed performances, with a bounded loss in classification accuracy. Recent works [162]-[164]

focus their attention on approximate computing as emerging trend to achieve even growing improvements when applied to computationally intensive algorithms, such as CNNs. As a related research direction, and considering results obtained in this thesis, exploring innovative computing techniques suitable for integration within heterogeneous FPGAs could be promising.

APPENDIX A

Table A.1 Parameters and acronyms used to explain the novel stereo vision algorithm.

Symbol	Name	Description
$R_{(x,y)}$	Reference pixel	The generic pixel located at the x-th row and y-th column within the image used as the reference for the disparity computation
d	Current disparity	The disparity value currently considered to perform computations
$C_{(x,y+d)}$	Candidate pixel	The generic pixel located at the x-th row and $(y+d)$-th column within the image used as the candidate for the disparity computation
$dmin$	Minimum disparity	The minimum value of the disparity d at which the candidate pixel $C_{(x,y+d)}$ matching with $R_{(x,y)}$ can be located
$dmax$	Maximum disparity	The maximum value of the disparity d at which the candidate pixel $C_{(x,y+d)}$ matching with $R_{(x,y)}$ can be located
dr	Disparity range	The number of possible disparity values ($dr=dmax-dmin+1$)
$MC_{(x,y,d)}$	Matching cost	The measure of how much it costs considering the pixels $R_{(x,y)}$ and $C_{(x,y+d)}$ matching, that is corresponding to projections of the same 3D point
s	Aggregation radius	The radius of the aggregation windows formed around the locations (x,y) and $(x,y+d)$ within the reference and the candidate images, respectively
$W \times W$	Aggregation window size	The number of elements within the generic aggregation window ($W=2 \times s+1$).
s_T	Support radius	The radius of the support windows formed around the locations (x,y) to perform transformations
$W_T \times W_T$	Support window size	The number of pixels within the generic support window ($W_T=2 \times s_T+1$)
$P_{(x+r,y+r')}$	Neighbouring pixel	The generic pixel within the support window centred at $R_{(x,y)}$
$Q_{(x+r,y+r'+d)}$	Neighbouring pixel	The generic pixel within the support window centred at $C_{(x,y+d)}$
Δc	Absolute value of the pixel intensity difference	The absolute value of the intensity difference between the neighbouring pixel and the central pixel of the generic support window (i.e. $\Delta c = \lvert P_{(x+r,y+r')} - R_{(x,y)} \rvert$ for reference pixels; whereas $\Delta c = \lvert Q_{(x+r,y+r'+d)} - C_{(x,y+d)} \rvert$ for candidate pixels)
ACT	-	Adaptive Census Transformation
$WCVR_{(x,y)}$	-	The weighted census vector computed for the support window centred at $R_{(x,y)}$
$WCVC_{(x,y+d)}$	-	The weighted census vector computed for the support window centred at $C_{(x,y+d)}$
$AVR_{(x,y)}$	-	The additional vector computed processing the support window centred at $R_{(x,y)}$
$AVC_{(x,y+d)}$	-	The additional vector computed processing the support window centred at $C_{(x,y+d)}$
$w(P,R)$	Support weight	The support provided by the neighbouring pixel P to the central pixel R within the generic support window of the reference image
$w(Q,C)$	Support weight	The support provided by the neighbouring pixel Q to the central pixel C within the generic support window of the candidate image
$D_{(x,y,d)}$	Dissimilarity	The measure of how much the reference and candidate pixels $R_{(x,y)}$ and $C_{(x,y+d)}$ differ
$\delta_{(x,y)}$	Output disparity	The disparity d in the evaluated disparity range for which the minimum dissimilarity is obtained
h	-	The number of partitions in which the disparity range is split
Pl	Parallelism level	The number of disparity values associated to each partition
Nonocc	Non occluded regions	Regions of pixels corresponding to the projections of 3D points visible from both cameras (i.e. pixels for which a valid disparity value certainly exists)
Disc	Discontinuity regions	Regions within the acquired images corresponding to disparity discontinuities
All	All regions	Regions including the overall images (the borders excepted)
ground truth	-	The image collecting the true disparity value for each pixel in the reference image
error rate	-	The percentage of computed disparity values that differ from the true values
fps	Frame rate	The number of disparity map calculated within 1 second
Calibration Parameters		
RL	-	Rotation Matrix of the left camera
fcL	-	Focal length vector of the left camera
ccL	-	Principal point vector of the left camera
kcL	-	Distortion vector of the left camera
KL	-	Perspective projection matrix of the left camera
KKL	-	Left Camera Matrix
ML	-	Alignment Matrix of the left camera $ML = RL^{T} \times KKL^{-1}$
RR	-	Rotation Matrix of the right camera
fcR	-	Focal length vector of the right camera
ccR	-	Principal point vector of the right camera
kcR	-	Distortion vector of the right camera
KR	-	Perspective projection matrix of the left camera
KKR	-	Right Camera Matrix
MR	-	Alignment Matrix of the right camera $ML = RL^{T} \times KKL^{-1}$

PUBLICATIONS

2020

- **Fanny Spagnolo**, Stefania Perri, Fabio Frustaci, Pasquale Corsonello

  Reconfigurable Convolution Architecture for Heterogeneous Systems-on-Chips. Accepted for oral presentation to the 9th Mediterranean Conference on Embedded Computing (MECO 2020).

- **Fanny Spagnolo**, Stefania Perri, Pasquale Corsonello

  Design of a Real-Time Face Detection Architecture for Heterogeneous Systems-on-Chips. Accepted for publication in Integration, the VLSI Journal.

2019

- Stefania Perri, **Fanny Spagnolo**, Pasquale Corsonello

  A Parallel Connected Component Labeling Architecture for Heterogeneous Systems-on-Chip. Electronics 2020, vol. 9(2), no. 292.

- **Fanny Spagnolo**, Stefania Perri, Fabio Frustaci, Pasquale Corsonello

  Energy-efficient architecture for CNNs inference on heterogeneous FPGAs. Journal of Low Power Electronics and Applications, vol. 10(1), issue 1, 2020.

- Stefania Perri, **Fanny Spagnolo**, Fabio Frustaci, Pasquale Corsonello

  Parallel architecture of power-of-two multipliers for FPGAs. IET Circuits, Devices & Systems, vol. 14, issue 3, pp. 381–389, 2020.

- **Fanny Spagnolo**, Pasquale Corsonello, Stefania Perri

  Efficient Architecture for Integral Image Computation on Heterogeneous FPGAs. In Proceedings of the 15th IEEE Conference on PhD Research in Microelectronics and Electronics, PRIME 2019. (Gold Leaf Award)

- **Fanny Spagnolo**, Stefania Perri, Pasquale Corsonello

  An Efficient Hardware-Oriented Single-Pass Approach for Connected Component Analysis. Sensors 2019, vol. 19, issue 14, no. 3055, pp. 1-14.

2018

- **Fanny Spagnolo**, Stefania Perri, Fabio Frustaci, Pasquale Corsonello

  Designing fast convolution engines for deep learning applications. In Proceedings of the 25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018.

- **Fanny Spagnolo**, Stefania Perri, Fabio Frustaci, Pasquale Corsonello

  Connected Component Analysis for Traffic Sign Recognition Embedded Processing Systems. In Proceedings of the 25th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2018.

- Stefania Perri, Fabio Frustaci, **Fanny Spagnolo**, Pasquale Corsonello

  Stereo Vision Architecture for Heterogeneous Systems-on-Chip. Journal of Real-Time Image Processing, Springer, May 2018. DOI: https://doi.org/10.1007/s11554-018-0782-z

- Stefania Perri, Fabio Frustaci, **Fanny Spagnolo**, Pasquale Corsonello

  Design of Real-Time FPGA-based Embedded System for Stereo Vision. In Proceedings of the IEEE International Symposium on Circuits & Systems, ISCAS 2018.

- **Fanny Spagnolo**, Stefania Perri, Fabio Frustaci, Pasquale Corsonello

  An Efficient Connected Component Labeling Architecture for Embedded Systems. Journal of Low Power Electronics and Applications vol. 8(7), issue 1, 2018.

2017

- **Fanny Spagnolo**, Stefania Perri, Pasquale Corsonello

  Evaluating Heterogeneous Architecture based on Zynq AP SOC for Real-Time Video Processing. In Proceedings of the International Conference on Advances in Circuits, Electronics and Micro-electronics, CENICS 2017. (Best Paper Award)

REFERENCES

[1] L. Atzori, A. Iera and G. Morabito, "The Internet of Things: A Survey", in *Computer Networks*, Elsevier, vol. 54, issue 15, pp. 2787-2805, Oct. 2010.

[2] M. Ali Feki, F. Kawsar, M. Boussard and L. Trappeniers, "The Internet of Things: The Next Technological Revolution, in *Computer*, vol. 46, pp. 24-25, Feb. 2013.

[3] Gartner [Online]. Available: http://www.gartner.com/newsroom/id/2684616

[4] S. Biookaghazadeh, M. Zhao, and F. Ren, "Are FPGAs Suitable for Edge Computing?", Workshop on Hot Topics in Edge Computing (HotEdge 18), Boston, MA, USA, July 2018.

[5] A. HajiRassouliha, A.J. Taberner, M.P. Nash and P.M.F. Nielsen, "Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms", in *Signal Processing: Image Communication*, Elsevier, vol. 68, pp. 101-119, 2018.

[6] M. Capra, R. Peloso, G. Masera, M.R. Roch and M. Martina, "Edge Computing: A Survey On the Hardware Requirements in the Internet of Things World", in *Future Internet* (2019), vol. 11, no. 100, Apr. 2019.

[7] V.S. Chua, J.Z. Esquivel, A.S. Paul, T. Techathamnukool, C.F. Ferjardo, N. Jain, O. Tickoo and R. Iyer, "Visual IoT: Ultra-Low-Power Processing Architectures and Implications", in *IEEE Micro*, vol. 37, issue 6, pp. 52-61, Nov. 2017.

[8] Y. Yang, H. Zhang, D. Yuan, D. Sun, G. Li, R. Ranjan and M. Sun, "Hierarchical extreme learning machine based image denoising network for visual Internet of Things", in *Applied Soft Computing Journal*, Elsevier, vol. 74 (2019), pp. 747-759.

[9] M.O. Ojo, S. Giordano, G. Procissi and I.N. Seitanidis, "A Review of Low-End, Middle-End, and High-End IoT Devices", in *IEEE Access*, vol. 6 (2018), pp. 70528-70554, Dec. 2018.

[10] F. Spagnolo, S. Perri, F. Frustaci, P. Corsonello, "Energy-efficient architecture for CNNs inference on heterogeneous FPGAs", submitted to *Journal of Low Power Electronics and Applications*.

[11] M. Rusci, D. Rossi, E. Farella and L. Benini, "A sub-mW IoT-endnode for always-on visual monitoring and smart triggering", in *IEEE Internet of Things Journal*, vol. 4, issue 5, pp. 1284-1295, Oct. 2017.

[12] M. Tausif, A. Jain, E. Khan and M. Hasan, "Low Memory Architectures of Fractional Wavelet Filter for low-cost Visual Sensors and Wearable Devices", in *IEEE Sensors Journal*, to be published, July 2019.

[13] G. Lacey, G. Taylor and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future" [Online]. Available: https://arxiv.org/abs/1602.04283.

[14] F. Spagnolo, S. Perri and Pasquale Corsonello, "Evaluating Heterogeneous Architectures based on Zynq AP SoC for Real-Time Video Processing", Proc. of CENICS 2017, Rome, Italy, Sept. 2017.

[15] S. Perri, F. Frustaci, F. Spagnolo and P. Corsonello, "Stereo Vision Architecture for Heterogeneous Systems-on-Chip", in *Journal of Real-Time Image Processing*, 2018.

[16] S. Perri, F. Frustaci, F. Spagnolo and P. Corsonello, "Design of Real-Time FPGA-Based Embedded System for Stereo Vision", Proc. of IEEE ISCAS 2018, Firenze, Italy, May 2018.

[17] F. Spagnolo, P. Corsonello and S. Perri, "Efficient Architecture for Integral Image Computation on Heterogeneous FPGAs", Proc. of IEEE PRIME 2019, Lausanne, Switzerland, July 2019.

[18] O. Veksler, "Fast Variable Window for Stereo Correspondence using Integral Images", Proc. of IEEE CVPR 2003, Madison, WI, USA, June 2003.

[19] F. Spagnolo, S. Perri, F. Frustaci and P. Corsonello, "An Efficient Connected Component Labeling Architecture for Embedded Systems", in *Journal of Low Power Electronics and Applications*, vol. 8(7), issue 1, 2018.

[20] F. Spagnolo, S. Perri, F. Frustaci and P. Corsonello, "Connected Component Analysis for Traffic Sign Recognition Embedded Processing Systems", Proc. of IEEE ICECS 2018, Bordeaux, France, December 2018.

[21] F. Spagnolo, S. Perri and P. Corsonello, "An Efficient Hardware-Oriented Single-Pass Approach for Connected Component Analysis", in *Sensors* 2019, vol. 19, issue 14, no. 3055, pp. 1-14.

[22] F. Spagnolo, S. Perri, F. Frustaci and P. Corsonello, "Designing Fast Convolutional Engine for Deep Learning Applications", Proc. of IEEE ICECS 2018, Bordeaux, France, December 2018.

[23] B. Blanco-Filgueira, D. Garcìa-Lesta, M. Fernàndez-Sanjurjo, V.M. Brea and P. Lòpez, "Deep Learning-Based Multiple Object Visual Tracking on Embedded System for IoT and Mobile Edge Computing Applications", in *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 5423-5431, June 2019.

[24] V. Sze, Y. Chen, T. Yang and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey", Proc. of IEEE, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[25] H. de Groot, "IoT and the cloud: A hacked personality and an empty battery head-ache or an intuitive environment to make our lives easier?", Proc. of IEEE S3S 2015, Rohnert Park, CA, USA, Oct. 2015.

[26] M. Alioto, "IoT: Bird's Eye View, Megatrends and Perspectives", in *Enabling the Internet of Things*, Springer Book, pp. 1-45, Jan. 2017.

[27] D.G. Costa, "Visual Sensors Hardware Platforms: A Review", in IEEE Sensors Journal, Nov. 2019. DOI: 10.1109/JSEN.2019.2952447.

[28] ARM Cortex-M Series [Online]. Available: https://www.arm.com/products/processors/cortex-m.

[29] STM32 32-bit ARM Cortex MCUs [Online]. Available: http://www.st.com.

[30] A. Rodrìguez, A. Navarro, R. Asenjo, F. Corbera, R. Gran, D. Suàrez and J. Nunez-Yanez, "Exploring heterogeneous scheduling for edge computing with CPU and FPGA MPSoCs", in *Journal of Systems Architecture*, vol. 98 (2019), pp. 27-40, June 2019.

[31] C. Hao, X. Zhang, Y. Li, S. Huang, K. Rupnow, W. Hwu and D. Chen, "FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge", Proc. of DAC 2019, Las Vegas, Nevada, USA, June 2019.

[32] D. Chen, J. Cong, S. Gurumani, E. Hwu, K. Rupnow and Z. Zhang, "Platform choices and design demands for IoT platforms: cost, power, and performance tradeoff", in *IET Cyber-Physical Systems: Theory and Applications*, vol. 1, issue 1, pp. 70-77, 2016.

[33] Altera User Customizable ARM-base SoC [Online]. Available: https://www.altera.com/content/dam/alterawww/global/en_US/pdfs/literature/br/br-soc-fpga.pdf

[34] Xilinx Zynq All Programmable SoC [Online]. Available: http://www.xilinx.com/products/silicondevices/soc/zynq-7000.html

[35] P. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun and E. Culurciello, "NeuFlow: Dataflow Vision Processing System-on-a-Chip", Proc. of IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS), Boise, USA, Aug. 2012.

[36] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi and L. Benini, "A Heterogeneous Multicore System on Chip for Energy-Efficient Brain Inspired Computing", in *IEEE Trans. on Circuits and Systems-II: Express Brief*, vol. 65, no. 8, pp. 1094-1098, Aug. 2018.

[37] M.D. Valdes Pena, J.J. Rodriguez-Andina and M. Manic, "The Internet of Things: The Role of Reconfigurable Platforms", in *IEEE Industrial Electronic Magazine*, vol. 11, issue 3, pp. 6-19, Set. 2017.

[38] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang and W. Shi, "Edge Computing for Autonomous Driving: Opportunities and Challenges", *Proc. of the IEEE*, vol. 107, no. 8, pp. 1697-1716, 2019.

[39] M. Urbina, T. Acosta, J. Làzaro, A. Astarloa and U. Bidarte, "Smart Sensor: SoC Architecture for the Industrial Internet of Things", in *IEEE Internet of Things Journal*, vol. 6, no. 4, pp. 6567-6577, Aug. 2019.
[40] M. Birem and F. Berry, "DreamCam: A modular FPGA-based smart camera architecture", in *Journal of Systems Architecture*, vol. 60 (2014), pp. 519-527, Jan. 2014.
[41] M. Benetti, M. Gottardi, T. Mayr and R. Passerone, "A Low-Power Vision System with Adaptive Background Substraction and Image Segmentation for Unusual Event Detection", in *IEEE Trans. On Circuits and Systems-I*, vol. 65, no. 11, Nov. 2018.
[42] O. Lahdenoja, T. Säntti, J.K. Poikonen, M. Laiho, A. Paasio, J. Pekkarinen and A. Salminen. "Embedded processing methods for online visual analysis of laser welding", in *Journal of Real-Time Image Processing*, vol. 16 (2019), pp. 1099-1116.
[43] P. Jokic, S. Emery, L. Benini, "BinaryEye: A 20 kfps Streaming Camera System on FPGA with Real-Time On-Device Image Recognition Using Binary Neural Networks", Proc. of IEEE SIES 2018, Graz, Austria, June 2018.
[44] T. Kryjak, M. Komorkiewicz and M. Gorgon, "Real-time hardware-software embedded vision system for smart camera implemented in Zynq SoC", in *Journal of Real-Time Image Processing*, vol. 15 (2018), pp. 123-159.
[45] D. Bhowmik, P. Garcia, A. Wallace, R. Stewart and G. Michaelson, "Power Efficient Dataflow Design for a Heterogeneous Smart Camera Architecture", Proc. of IEEE DASIP 2017, Dresden, Germany, Sept. 2017.
[46] M. Tausif, A. Jain, E. Khan and M. Hasan, "Low Memory Architectures of Fractional Wavelet Filter for low-cost Visual Sensors and Wearable Devices", in *IEEE Sensors Journal*, July 2019. To be published.
[47] T. Wang, C. Wang, X. Zhou and H. Chen, "A Survey of FPGA Based Deep Learning Accelerators: Challenges and Opportunities" [Online]. Available: https://arxiv.org/abs/1901.04988.
[48] X. Feng, Y. Jiang, X. Yang, M. Du and X. Li, "Computer vision algorithms and hardware implementations: A Survey", in *Integration the VLSI Journal*, July 2019. Article in press. Available: https://doi.org/10.1016/j.vlsi.2019.07.005.
[49] S. Mittal, "A survey of FPGA-based accelerators for convolutional neural networks", in *Neural Computing and Applications*, Spinger, pp. 1-31, Sept. 2018.
[50] M.P. Véstias, "A Survey of Convolutional Neural Networks on Edge with Reconfigurable Computing", in *Algorithms* 2019, vol. 12, no. 154, July 2019.
[51] V. Gokhale, J. Jin, A. Dundar, B. Martini and E. Culurciello, "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks", Proc. of IEEE CVPR 2014, Columbus, OH, USA, June 2014, pp. 696-701.
[52] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang and H. Yang, "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA", in *IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no.1, pp. 35-47, Jan. 2018.
[53] S.I. Venieris and C. Bouganis, "fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs", in *IEEE Trans. On Neural Networks and Learning Systems*, vol. 30, no. 2, pp. 326-342, July 2018.
[54] X. Chen and Z. Yu, "A Flexible and Energy-Efficient Convolutional Neural Network Acceleration with Dedicated ISA and Accelerator", in *IEEE Trans. on VLSI Systems*, vol. 26, no. 7, pp. 1408-1412, March 2018.
[55] P. Dudek and S.J Carey, "General-purpose 128×128 SIMD processor array with integrated image sensor", in *Electronics Letters*, vol. 42, issue 12 (2006), pp. 678–679.
[56] M. Benetti, M. Gottardi and Z. Smilansky, "A 80μW 30 fps 104×104 all-nMOS pixels CMOS imager wit 7-bit PWM ADC for robust detection of relatively intensity change", Proc of ESSCIRC, Bucarest, Romania, Sept. 2013.
[57] Omnivision sensors Datasheet [Online]. Available: https://www.voti.nl/docs/OV7670.pdf.
[58] Kovilta sensors with sense [Online]. Available: http://www.kovilta.fi/.
[59] F. Cheng and X. Chen, "Integration of 3D stereo vision measurements in industrial robot applications", Proc. of 2008 IAJC-IJME International Conference, Nashville, 2008.

[60] L. Nalpantidis and A. Gasteratos, "Stereo Vision for robotic applications in the presence of non-ideal lighting", in *Image and Vision Computing*, vol. 28, issue 6, pp. 940-951, June 2010.
[61] W. Yu and B. Xu, "A portable stereo vision system for whole body surface imaging", in *Image and Vision Computing*, vol. 28, issue 4, pp. 605-613, Apr. 2010.
[62] O. Faugeras, "Three dimensional computer vision: a geometric viewpoint", MIT Press, Cambridge (1993).
[63] S. Madeo, R. Pelliccia, C. Salvadori, J.M. Del Rincon and J.C. Nebel, "An optimized stereo vision implementation for embedded systems: application to RGB and infra-red images", in *Journal of Real-Time Image Processing*, vol. 12, issue 4, pp. 725–746, 2016.
[64] S. Perri, P. Corsonello and G. Cocorullo, "Adaptive census transform: a novel hardware-oriented stereovision algorithm", in *Computer Vision and Image Understanding*, vol. 117, issue 1, pp. 29-41, 2013.
[65] G. Cocorullo, P. Corsonello, F. Frustaci and S. Perri, "An efficient hardware-oriented stereo matching algorithm", in *Microprocessors and Microsystems*, vol. 46, issue 10, pp. 21-33, 2016.
[66] M. Perez-Patricio and A. Aguilar-Gonzalez, "FPGA implementation of an efficient similarity-based adaptive window algorithm for real-time stereo matching", in *Journal of Real-Time Image Processing*, vol. 16, issue 2, pp. 271–287, 2019.
[67] C. Ttofis, C. Kyrkou and T. Theocharides, "A Low-Cost Real-Time embedded stereo vision system for accurate disparity estimation based on guided image filtering", in *IEEE Trans. On Computers*, vol. 64, no. 9, pp. 2678-2693, 2016.
[68] V.D. Nguyen, D.D. Nguyen, T.T. Nguyen, V.Q. Dinh and J.W. Jeon, "Support local pattern and its application to disparity improvement and texture classification", in *IEEE Trans. On Circuits and Systems for Video Technology*, vol. 24, no. 2, pp. 263-276, 2014.
[69] D.W. Yang, L.C. Chu, C.W. Chen, J. Wang and M.D. Shieh, "Depth-reliability-based Stereo Matching algorithm and its VLSI architecture design", in *IEEE Trans. On Circuits and Systems for Video Technology*, vol. 25, no. 6, pp. 1038-1050, 2015.
[70] M. Perez-Patricio, A. Aguilar-Gonzalez, M. Arias-Estrada, H.R. Hernandez-de Leon, J.L. Camas-Anzueto and J.A. Jesùs Osuna-Coutiño, "An FPGA stereo matching unit based on fuzzy logic", in *Microprocessors and Microsystems*, vol. 42, pp. 87-99, 2016.
[71] A. Aguilar-Gonzalez and M. Arias-Estrada, "An FPGA stereo matching processor based on the sum of hamming distances", Proc. of ARC 2016, Rio de Janeiro, Brazil, March 2016.
[72] P.M. Santos, J.C. Ferreira and J.S. Matos, "Scalable hardware architecture for disparity map complutation and object location I real-time", in *Journal of Real-Time Image Processing*, vol. 11, issue 3, pp. 473–485, 2016.
[73] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms", in International Journal on Computer Vision, vol. 47, issues 1-3, pp. 7–42, 2002.
[74] B. Tippetts, D.J. Lee, K. Lillywhite, and J. Archibald, "Review of stereo vision algorithms and their suitability for resource-limited systems", in *Journal of Real-Time Image Processing*, vol. 11, issue 1, pp. 5-25, 2016.
[75] L. Puglia, M. Vigliar and G. Raiconi, "Real-Time low-power FPGA architecture for stereo vision", in *IEEE Trans. On Circuits and Systems II-Express Brief*, vol. 64, no. 11, pp. 1307-1311, 2017.
[76] L. Li, X. Yu, S. Zhang, X. Zhao and L. Zhang, "3D cost aggregation with multiple minimum spanning trees for stereo matching", in *Applied Optics*, vol. 56, Issue 12, pp. 3411-3420, 2017.
[77] Camera Calibration Toolbox for Matlab [Online]. Available: http://www.vision.caltech.edu/bougu etj/calib doc/.
[78] P. Zicari, S. Perri, P. Corsonello and G. Cocorullo, "Low-cost FPGA stereo vision system for real time disparity maps calculation", in *Microprocessors and Microsystems*, vol. 36, issue 4, pp. 281-288, 2012.
[79] L. Di Stefano, M. Marchionni, S. Mattoccia and G. Neri, "A fast area-based stereo matching algorithm", in *Image and Vision Computing*, vol. 22, issue 12, pp. 983-1005, 2004.

[80] AMBA 4 AXI4, AXI4-Lite, and AXI4-Stream Protocol Assertions User Guide [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html

[81] Vivado High Level Synthesis [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[82] Xilinx Zynq-7000 External Memory Interfaces". [Online]. Available: https://www.xilinx.com/products/technology/memory.html#externalMemory.

[83] Middlebury Stereo Vision page [Online]. Available: http://vision.middlebury.edu/stereo.

[84] F.C. Crow, "Summed-Area Tables for Texture Mapping", in *Computer Graphics*, vol. 18, no. 3, pp. 207-212, 1984.

[85] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features", in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition* (CVPR 2001), Kauai, HI, USA, Dec. 2001, pp. I511-I518.

[86] H. Bay, A. Ess, T. Tuytelaars and L. Gool, "Speeded-Up Robust Features (SURF)", in *Computer Vision and Image Understanding*, vol. 110 (2008), pp. 346-359.

[87] M. Grabner, H. Grabner and H. Bischof, "Fast Approximated SIFT", Lecture notes in Computer Science Proc. ACCV 2006, Hyderabad, India, Jan. 2006.

[88] S. Ehsan, A.F. Clark, N. ur Rehman and K.D. McDonald-Maier, "Integral Images: Efficient Algorithms for Their Computation and Storage in Resource-Constrained Embedded Vision Systems", in *Sensors* 2015, vol. 15, July 2015.

[89] B. Kisačanin, "Integral Image Optimizations for Embedded Vision Applications", Proc. of the IEEE Southwest Symposium on Image Analysis and Interpretation, Santa Fe, NM, USA, March 2008.

[90] P. Ouyang, S. Yin, Y. Zhang, L. Liu and S. Wei, "A Fast Integral Image Computing Hardware Architecture with High Power and Area Efficiency", *IEEE Trans. on Circuits and Systems-II: Express Brief*, vol. 62, no.1, pp. 75-79, Jan. 2015.

[91] O.G. Valenzuela-López, J.L. Tecpanecatl-Xihuitl and R.M. Aguilar-Ponce, "A Novel Low Latency Integral Image Architecture", Proc. of IEEE ROPEC2017, Ixtapa, Mexico, Nov. 2017.

[92] 7 Series DSP48E1 Slice User Guide UG479 (v1.10) March 27, 2018 [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[93] Intel Digital Signal Processing [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/programmable/digital-signal-processing/overview.html

[94] K. Zhang, J. Lu and G. Lafruit, "Cross-based local stereo matching using orthogonal integral images", in *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 19, no. 7, pp. 1073-1079, Apr. 2009.

[95] V.Q. Dinh, V.D. Nguyen and J.W. Jeon, "Robust Matching Cost Function for Stereo Correspondence Using Matching by Tone Mapping and Adaptive Orthogonal Integral Image", in IEEE Trans. on Image Processing, vol. 24, no. 12, pp. 5416-5431, Dec. 2015.

[96] S. Zhu and L. Yan, "Local stereo matching algorithm with efficient matching cost and adaptive guided image filter", in *The Visual Computer*, vol. 33, issue 9, pp. 1087-1102, Sep. 2017.

[97] M.J. Klaiber, D.G. Bailey and Y.O. Baroud, "A Resource-Efficient Hardware Architecture for Connected Component Analysis", in IEEE Trans. on Circuits and Systems for Video Technology, vol. 26, no. 7, pp. 1334–1349, July 2016.

[98] P. Sutheebanjard, W. Premchaiswadi, "Efficient scan mask techniques for connected components labeling algorithm", in *EURASIP Journal on Image and Video Processing*, vol. 2011:14, Dec. 2011.

[99] L. He, Y. Chao and K. Suzuki, "An efficient first-scan method for label-equivalence-based labeling algorithms", in *Pattern Recognition Letters*, vol. 31, pp. 28-35, 2010.

[100] W. Farhat, H. Faiedg, C. Souani and K. Basbes, "Real-Time Embedded System for Traffic Sign Recognition based on Zedboard", in *Journal of Real-Time Image Processing*, pp. 1-11, 2017.

[101]R. Acevedo-Avila, M. Gonzalez-Mendoza, A. Garcia-Garcia, “A Linked List-based Algorithm for Blob Detection on Embedded Vision-based Sensors”, in *Sensors* 2016, vol. 16, issue 782, 2016.
[102]D.G. Bailey and C.T. Johnston, “Single Pass Connected Component Analysis”, Proc. of Image and Vision Computing New Zealand 2007, Hamilton, New Zealand, Dec. 2007, pp. 282-287.
[103] Y. Ito, and K. Nakano, “Low-Latency Connected Component Labeling Using an FPGA”, in *International Journal of Foundations of Computer Science*, vol. 21, no. 03, pp. 405-425, 2010.
[104] K. Appiah, A. Hunter, P. Dickinson and J. Owens, “A Run-Length based Connected Component Algorithm for FPGA Implementation”, Proc. of FTB 2008, Taipei, Taiwan, Dec. 2008, pp. 177-184.
[105] M.J. Klaiber, D.G. Bailey and S. Simon, “A single-cycle parallel multi-slice connected components analysis hardware architecture”, in *Journal of Real-Time Image Processing*, vol. 16, issue 4, pp. 1165-1175, Aug. 2019.
[106] C.Y. Lin, S.Y. Li and T.H. Tsai, “A scalable parallel hardware architecture for Connected Component Labeling”, Proc. of IEEE 17th International Conference on Image Processing, Honk Kong, China, Sept. 2010, pp. 3753-3756.
[107] C. Grana, D. Borghesani and R. Cucchiara, “Optimized Block-based Connected Component Labeling with Decision Trees”, in *IEEE Trans. on Image Processing*, vol. 19, no. 6, pp. 1596-1609, 2010.
[108] C. Grana, D. Borghesani, P. Santinelli and R. Cucchiara, “High-Performance Connected Component Labeling on FPGA”, Proc. of DEXA 2010, Bilbao, Spain, Sept. 2010.
[109] N. Ma, D.G. Bailey and C.T. Johnston, “Optimised single pass connected components analysis”, Proc. of the ICECE 2008, Taipei, Taiwan, Dec. 2008.
[110] J.W. Tang, N. Shaikh-Husin, U.U. Sheikh and M.N. Marsono, “A linked list run-length-based single-pass connected component analysis for real-time embedded hardware”, in *Journal of Real-Time Image Processing*, vol. 15, issue 1, pp. 197-215, June 2018.
[111] C. Zhao, G. Duan and N. Zheng, “A Hardware-Efficient Method for Extracting Static Information of Connected Component”, in *Journal of Signal Processing Systems*, vol. 88, issue 1, pp. 55-65, July 2017.
[112] A.W. Malik, B.Thörnberg, M. Imran and N. Lawal, “Hardware Architecture for Real-Time Computation of Image Component Feature Descriptors on a FPGA”, in *International Journal of Distributed Sensor Networks*, vol. 2014, pp. 1-14, 2014.
[113] J. Jeong, G. Lee, M. Lee and J. Kim, “A single-pass Connected Component Labeler without Label Merging Period”, in *Journal of Signal Processing Systems*, vol. 84, issue 2, pp. 211-223, Sept. 2015.
[114] M. Tekleyohannes, M. Sadri, M. Klein and M. Siegrist, “An Advanced Embedded Architecture for Connected Component Analysis in Industrial Applications”, Proc. of DATE 2017, Lausanne, Switzerland, 2017, pp. 734-735.
[115] V.S. Kumar, K. Irick, A.A. Maashri and N. Vijaykrishnan, “A Scalable Bandwidth Aware Architecture for Connected Component Labeling”, Proc. of 2010 IEEE Computer Society Annual Symposium on VLSI, Lixouri, Kefalonia, Greece, July 2010.
[116] A.W. Malik, B. Thörnberg, X. Cheng and N. Lawal, “Real-time Component Labelling with Centre of Gravity Calculation on FPGA”, Proc. of ICONS 2011, St. Maarten, The Netherlands Antilles, Jan. 2011, pp. 39-43.
[117] Y. He, T. Hu and D. Zeng, “Scan-flood Fill(SCAFF): an Efficient Automatic Precise Region Filling Algorithm for Complicated Regions”, Proc. of CVPR 2019, Long Beach, CA, USA, June 2019.
[118] F. Dìaz-del-Rio, P. Real and D. Onchis, “Labeling Color 2D Digital Images in Theoretical Near Logarithmic Time”, Lecture notes in CAIP 2017, pp. 391-402, Ystad, Sweden, Aug. 2017.
[119] Y.-A. Daraghmi and A.M. Hasasneh, “Accurate Real-Time Traffic Sign Recognition Based on the Connected Component Labeling and the Color Histogram Algorithms”, in International Journal of Signal Processing Systems, vol. 4, no. 5, pp. 417-421, Oct. 2016.

[120] S. Houben, J. Stallkamp, J. Salmen, M. Schlipsing and C. Igel, “Detection of Traffic Signs in Real-World Images: The German Traffic Sign Detection Benchmark”, Proc. of IJCNN 2013, Dallas, USA, 2013.
[121] H. Li, K. Ota and M. Dong, “Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing”, in *IEEE Network*, vol. 32, issue 1, pp. 96-101, Jan. 2018.
[122] A. Krizhevsky, I. Sutskever and G.E. Hinton, “ImageNet Classification with Deep Convolutional Neural Network”, Proc. of NIPS 2012, Lake Tahoe, Nevada, USA, Dec. 2012, pp. 1097-1105.
[123] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu and M.F. Chang, “A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things”, in *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 198-208, Jan. 2018.
[124] Y.-H. Chen, T. Krishna, J.S. Emer and V. Sze, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”, in *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127-138, Jan. 2017.
[125] A. Shawahna, S.M. Sait and A. El-Maleh, “FPGA-based Accelerators for Deep Learning Networks for Learning and Classification: A Review”, in *IEEE Access*, vol. 7, pp. 7823-7859, 2019.
[126] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Moralese, I.A. Lungu M.B. Milde, F. Corradi, A. Linares-Barranco, S.C. Liu and T. Delbruck, “NullHop- A Flexible Convolutional Neural Network Accelerator based on Sparse Representations of Feature Maps”, in *IEEE Trans. on Neural Network and Learning Systems*, vol. 30, no. 3, pp. 644-656, July 2018.
[127] Y. Ma, Y. Cao, S. Vrudhula and J. Seo, “Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA”, in *IEEE Trans. on VLSI Systems*, vol. 26, no. 7, pp. 1354-1367, July 2018.
[128] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo and L. Benini, “NEURAghe: Exploiting CPU-FPGA Synergies for Efficient and Flexible CNN Inference Acceleration on Zynq SoCs”, in *ACM Trans. on Reconfigurable Technologies and Systems*, vol. 11, issue 3, no. 18, Dec. 2018.
[129] G. Li, F. Li, T. Zhao and J. Cheng, “Block Convolution: Towards Memory-Efficient Inference of Large-Scale CNNs on FPGA”, Proc. IEEE DATE 2018, Dresden, Germany, March 2018, pp. 1163-1166.
[130] M. Alameh, A. Ibrahim, M. Valle and G. Moser, “DCNN for Tactile Sensory Data Classification based on Transfer Learning”, Proc. of IEEE PRIME 2019, Lausanne, Switzerland, July 2019.
[131] X. Jin, C. Xu, J. Feng, Y. Wei, J. Xiong and S. Yan, “Deep learning with S-shaped rectified linear activation units”, Proc. of AAAI’2016, Phoenix, Arizona, Febr. 2016, pp. 1737-1743.
[132] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition”, 2014. [Online]. Available: https://arxiv.org/abs/1409.1556.
[133] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition”, Proc. of IEEE CVPR 2016, Las Vegas, NV, USA, June 2016, pp. 770-779.
[134] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, A. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, “Going Deeper with Convolutions”, Proc. IEEE CVPR 2015, Boston, MA, USA, June 2015, pp. 1-9.
[135] K. Chatfield, K. Simonyan, A. Vedaldi and A. Zisserman, “Return of the Devil in the Details: Delving Deep into Convolutional Nets” [Online]. Available: https://arxiv.org/pdf/1405.3531.pdf.
[136] J. Cong and B. Xiao, “Minimizing computation in convolutional neural networks”, Lecture notes in Computer Science, ICANN2014, pp. 281-290, Springer, 2014.
[137] K. Abdelouahab, M. Pelcat, J. Sérot, C. Bourrasset and F. Berry, “Tatics to Directly Map CNN Graphs on Embedded FPGAs”, in *IEEE Embedded Systems Letters*, vol. 9, no. 4, Dec. 2017.
[138] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based Learning Applied to Document Recognition”, *Proc. of the IEEE*, 1998.

[139] G.E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R.R. Salakhutdinov, “Improving neural networks by preventing coadaptation of feature detectors”. [Online]. Available: https://arxiv.org/abs/1207.0580.

[140] R. Andri, L. Cavigelli, D. Rossi and L. Benini, “Hyperdrive: A Systolically Scalable Binary-Weight CNN Inference Engine for mW IoT End-Nodes” [Online]. Available: https://arxiv.org/abs/1804.00623.

[141] T. Fujii, S. Sato, H. Nakahara and M. Motomura, “An FPGA Realization of a Deep Convolutional Neural Network Using a Threshold Neuron Pruning”, in *Applied Reconfigurable Computing*, pp. 268-280, Springer, 2017.

[142] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N.E. Jerger, R. Urtasun and A. Moshovos, “Proteus: Exploiting precision variability in deep neural networks”, in *Parallel Computing*, vol. 73(2018), pp. 40-51, Elsevier, 2018.

[143] P. Gysel, M. Motamedi and S. Ghiasi, “Hardware-oriented Approximation of Convolutional Neural Networks” [Online]. Available: https://arxiv.org/abs/1604.03168.

[144] A. Kouris, S.I. Venieris and C.S. Bouganis, “Cascade^CNN: Pushing the Performance Limits of Quantisation in Convolutional Neural Networks”, Proc. of IEEE FPL 2018, Dublin, Ireland, Dec. 2018.

[145] S. Lee, D. Kim, D. Nguyen and J. Lee, “Double MAC on a DSP: Boosting the Performance of Convolutional Neural Networks on FPGAs”, in *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 38, no.5, pp. 888-897, Apr. 2018.

[146] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou and X. Ji. “High-Performance FPGA-Based CNN Accelerator with Block-Floating-Point Arithmetic”, in *IEEE Trans. on VLSI Systems*, vol. 27, no. 8, pp. 1874-1885, Aug. 2019.

[147] D. Scherer, A. Müller and S. Behnke, “Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition”, Proc. ICANN 2010, Thessaloniki, Greece, Sept. 2010, pp. 92-101.

[148] M. Lin, Q. Chen, and S. Yan, “Network in Network”. [Online]. Available: https://arxiv.org/abs/1312.4400.

[149] S. Zhai, H. Wu, A. Kumar, Y. Cheng, Y. Lu, Z. Zhang and R. Feris, “S3Pool: Pooling with Stochastic Spatial Sampling”. [Online]. Available: https://arxiv.org/abs/1611.05138.

[150] Cortex-A9 NEON Media Processing Engine Technical Reference Manual, rev. r3p0. [Online]. Available: www.infocenter.arm.com.

[151] X. Chen, S. Xiao and Z. Yu, “A Reconfigurable Process Engine for Flexible Convolutional Neural Network Acceleration”, Proc. IEEE APSIPA ASC, Honolulu, Hawaii, USA, Nov. 2018, pp. 1402-1405.

[152] M. Hiardieck, M. Kumm, K. Möller and P. Zipf, “Reconfigurable Convolutional Kernels for Neural Networks on FPGAs”, Proc. ACM FPGA’19, Seaside, CA, USA, Febr. 2019.

[153] J. Qiu, J. Wang, S. Yao , K. Guo , B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang and H. Yang, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network”, Proc. ACM FPGA’16, Monterey, California, USA, Febr. 2016, pp. 26-35.

[154] J. Wang, J. Li and Z. Wang, “Efficient Hardware Architectures for Deep Convolutional Neural Network”, in *IEEE Trans. on Circuits and Systems I*, vol. 65, no. 6, pp. 1941-1953, Nov. 2017.

[155] Y. Shen, M. Ferdman and P. Milder, “Maximizing CNN Accelerator Efficiency Through Resource Partitioning”, Proc. ACM ISCA’17, Toronto, Canada, June 2017.

[156] J. Xu, Z. Liu, J. Jiangm Y. Dou and S. Li, “CaFPGA: An automatic generation model for CNN accelerator”, in *Microprocessors and Microsystems*, vol. 60, pp. 196-206, 2018.

[157] N. Shah, P. Chaudhari and K. Varghese, “Runtime Programmable and Memory Bandwidth Optimized FPGA-Based Coprocessor for Deep Convolutional Neural Network”, in *IEEE Trans. On Neural Networks and Learning Systems*, vol. 29, no. 12, pp. 5922-5934, Dec. 2018.

[158] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo and L. Benini, “A High-Efficiency Runtime Reconfigurable IP for CNN Acceleration on a Mid-Range All-Programmable SoC”, Proc. IEEE ReConFig, Cancun, Mexico, Nov. 2016.

[159] S.I. Venieris and C. Bouganis, “f-CNNX: A Toolflow for Mapping Multiple Convolutional Neural Networks on FPGAs”, Proc. IEEE FPL, Dublin, Ireland, Aug. 2018, pp. 381-388.

[160] Intel Arria 10 FPGAs. [Online]. Available: https://www.intel.it/content/www/it/it/products/programmable/fpga/arria-10.html

[161] Intel Cyclone V FPGAs. [Online]. Available: https://www.intel.it/content/www/it/it/products/programmable/fpga/cyclone-v.html

[162] J. Choi and S. Venkataramani, “Approximate Computing Techniques for Deep Neural Networks”, in *Approximate Circuits*, Springer, pp. 307-329.

[163] D. Esposito, A.G.M. Strollo, E. Napoli, D. De Caro and N. Petra, “Approximate Multipliers Based on New Approximate Compressors”, in *IEEE Trans. on Circuits and Systems I*, vol. 65, no. 12, pp. 4169-4182, Dec. 2018.

[164] Y. Gong, B. Liu, W. Ge and L. Shi, “ARA: Cross-Layer approximate computing framework based reconfigurable architecture for CNNs”, in *Microelectronics Journal*, vol. 87(2019), pp. 33-44.