DISSERTAÇÃO DE MESTRADO

A MANYCORE VISION PROCESSOR ARCHITECTURE FOR EMBEDDED APPLICATIONS

Bruno Almeida da Silva

Brasília, Março de 2021

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO

A MANYCORE VISION PROCESSOR ARCHITECTURE FOR EMBEDDED APPLICATIONS

Bruno Almeida da Silva

Dissertação de Mestrado submetida ao Departamento de Engenharia Mecânica como requisito parcial para obtenção do grau de Mestre em Sistemas Mecatrônicos

Banca Examinadora

FICHA CATALOGRÁFICA

DA SILVA, BRUNO ALMEIDA
A manycore vision processor architecture for embedded applications [Distrito Federal] 2021.
xvi, 82 p., 210 x 297 mm (ENM/FT/UnB, Mestre, Engenharia Mecânica, 2021).
Dissertação de Mestrado - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Mecânica

1. Multiprocessor System-on-Chip	2. Network-on-Chip
3. Image Processing	4. Computer Vision
I. ENM/FT/UnB	II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

DA SILVA, B. A. (2021). *A manycore vision processor architecture for embedded applications*. Dissertação de Mestrado, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 82 p.

CESSÃO DE DIREITOS

AUTOR: Bruno Almeida da Silva
TÍTULO: A manycore vision processor architecture for embedded applications.
GRAU: Mestre em Sistemas Mecatrônicos   ANO: 2021

Bruno Almeida da Silva
Depto. de Engenharia Mecânica (ENM) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

Agradecimentos

Agradeço a Deus e Nossa Senhora, pela minha vida e pelo sustento providente em especial durante o período deste mestrado. Aos meus pais, pelo apoio incondicional e sustento material durante quase toda vida, que um dia eu possa retribuir parte do esforço que empregaram na minha criação e educação. À minha noiva, Andressa, pela companhia constante e suporte durante os anos da graduação e mestrado. Ao Arthur Floriano, pela amizade e conversas sobre a vida profissional. Aos amigos da CCUnB, pelas boas conversas e acolhida durante minha chegada na universidade. Ao Jones, pela orientação neste trabalho e também na minha carreira profissional. Ao Arthur Lima, por ter pego no pesado junto comigo pra alcançarmos alguns dos resultados desta dissertação. O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Bruno Almeida da Silva

RESUMO

Sistemas de Processamento de Imagem e Visão Computacional agora estão cada vez mais difundidos na tecnologia, permitindo aplicações para Sistemas Ciberfísicos, a Internet das Coisas, Realidade Aumentada, e Indústria 4.0. Essas aplicações trazem consigo a necessidade de Smart Cameras para processamento de imagens e vídeos localmente e em tempo real. No entanto, soluções comerciais de câmeras não são capazes de lidar com a grande quantidade de dados que precisa ser processada em curtos períodos de tempo nesses tipos de aplicação. Neste trabalho, mostramos o design e implementação de uma arquitetura multi-núcleo para ser usada nessas Smart Cameras, podendo ser classificada como um processador de visão. Com exploração massiva de paralelismo e características específicas para aplicações em Visão Computacional, nossa arquitetura é composta de Elementos de Processamento distribuídos e memórias conectadas via uma Rede Intra-chip. A arquitetura foi implementada como um FPGA overlay, focando na otimização do uso de hardware. A arquitetura parametrizada foi caracterizada por sua ocupação do hardware, frequência máxima de operação, taxa de processamento de quadros e consumo de energia. Configurações diferentes, de um até quatrocentos Elementos de Processamento, foram implementadas e comparadas com diversos trabalhos da literatura. Uma cadeia completa de Processamento de Imagens e Visão Computacional foi implementada para validar a arquitetura proposta, incluindo um sistema de aquisição de imagens com câmera. Estimou-se o consumo de energia da nossa arquitetura usando componentes próprios para medição de energia já presentes no kit FPGA. Os resultados mostram que a arquitetura proposta foi bem-sucedida em aliar programabilidade com performance, sendo uma alternativa adequada para futuras Smart Cameras. Sendo este trabalho uma prova de conceito totalmente funcional, muitas melhorias podem ser feitas, sendo sugeridas como trabalhos futuros.

Palavras-chave: Multiprocessadores em chip, Redes intra-chip, Processamento de Imagens, Visão Computacional

ABSTRACT

Real-Time Image Processing and Computer Vision systems are now in the mainstream of technologies enabling applications for Cyber-Physical Systems, Internet of Things, Augmented Reality, and Industry 4.0. These applications bring the need for Smart Camera for local real-time processing of images and videos. However, the massive amount of data to be processed within short deadlines cannot be handled by most commercial cameras. In this work, we show the design and implementation of a many-core vision processor architecture to be used in Smart Cameras. With massive parallelism exploration and application-specific characteristics, our architecture is composed of distributed Processing Elements and Memories connected through a Network-on-Chip. The architecture was implemented as an FPGA overlay, focusing on optimized hardware utilization. The parameterized architecture was characterized by its hardware occupation, maximum operating frequency, processing frame rate, and power consumption. Different configurations ranging from one to four hundred Processing Elements were implemented and compared to several works from the literature. A complete Image Processing and Computer Vision processing chain is implemented to validate the proposed architecture, including a camera acquisition scheme. This work also measures the power consumption of our architecture using built-in power monitoring components. The results show that the proposed architecture successfully allies programmability and performance, being a suitable alternative for future Smart Cameras. As this work is a functional proof of concept, there are many possible improvements suggested for future works.

Keywords: Multiprocessor System-on-Chip, Network-on-Chip, Image Processing, Computer Vision

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ADC	Analog-to-Digital Converter
APU	Application Processing Unit
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
CED	Canny Edge Detector
CMOS	Complementary Metal–Oxide–Semiconductor
CUDA	Compute Unified Device Architecture
CV	Computer Vision
DMA	Direct Memory Access
DMA	Direct Memory Access Controller
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EDA	Electronic Design Automation
FB	Frame Buffer
FF	Flip-flop
FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPI	General Purpose Interface
GPU	Graphics Processing Unit
GPP	General Purpose Port
HCD	Harris Corner Detector
HDMI	High-Definition Multimedia Interface
HLS	High-Level Synthesis
HPC	High Performance Computing
HP FPD	High-Performance Full Power Domain
HW/SW	Hardware/Software
IP	Image Processing
IP/CV	Image Processing and Computer Vision
JSON	JavaScript Object Notation

LUT	Look Up Table
LUTRAM	Look Up Table Random Access Memory
MCVP	Manycore Vision Processor
MCVP-AT	Manycore Vision Processor Automation Tool
MPI	Message Passing Interface
MPSoC	Multiprocessor System-on-chip
NoC	Network-on-chip
NUMA	Non-Uniform Memory Access
OS	Operating System
PE	Processing Element
PI	Pixel Interface
PL	Programmable Logic
PM	Pixel Memory
PMOD	Peripheral Module interface
PS	Programming Systems
PU	Processing Unit
QVGA	Quarter Video Graphics Array
RF	Register File
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single Input Multiple Data
SoC	System-on-chip
TCP/IP	Transmission Control Protocol and Internet Protocol
TLM	Transaction-Level Modeling
URAM	Ultra Random Access Memory
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration

1 INTRODUCTION

This chapter presents an introduction concerning the main motivations, contributions and the organization of this work.

1.1 MOTIVATION

The emergence of new trends in technology, like the Internet of Things and Industry 4.0, pulled out several applications based on Image Processing and Computer Vision (IP/CV) techniques. Cyber-Physical Systems, Augmented Reality, and Autonomous Machines, among others, all have applications supported by the extensive use of cameras.

These data-intensive tasks like Image Processing (IP) became popular, with the development of computers and its constant improvement, for Computer Graphics, Image Editing, Computer Vision, and Robotics (1)(2). Hence, new hardware designs are important to fulfill these applications requirements. This work pays special attention to Image Processing and Computer Vision (IP/CV) applications.

Human perception is capable of perceiving the 3D world around. How it looks, which objects, things, and persons are there. Even tell the difference among details like flowers, the aspect of petals, and its patterns, for example. Computer Vision (CV) uses mathematical models to recognize aspects of a scene: create a 3D reconstruction using hundreds of photos, track a person's movement, find people in photos, and more. This field has multiple difficulties because it deals with an inverse problem: recover unknown properties with insufficient information to specify a solution to model the visual world (3).

Even though Vision is a complex job, a wide variety of applications use CV nowadays (3):

- Optical character recognition (OCR): automatic numbers and letters recognition;
- Machine inspection: use stereo vision (a combination of two or more cameras to get more scene information) to measure tolerances and detect defects;
- Retail: to recognize automated products checkout;
- 3D model building: automated construction of 3D models from aerial photographs;
- Automotive safety: detecting unexpected objects in automobiles;
- Motion capture: use markers from multiple cameras to capture actors for computer animation;
- Surveillance: monitor for intrudes, analyze highway traffic, and detect social disturb.

Most of the conventional cameras are designed for the acquisition and transmission of images and videos. These cameras are not able to support complete applications running under real-time constraints. For these reasons, there is a need for devices capable of acquiring and processing images and videos efficiently under specific time constraints. Such devices are usually called Smart Cameras.

The design of such Smart Cameras is not a new topic, as seminal works have proposed the concepts of Focal-Plane and Near-Sensor Image Processing, on which highly-optimized systems would be able to integrate acquisition and processing on a single chip (4), like shown in Figure 1.1. Several works have been developed based on these concepts, spanning different areas, like photonics, VLSI technology, parallel computing, among others(5)(6)(7)(8). In this scenario, acquisition scheme plays an important role to optimize performance of Smart Cameras, because better pixel distribution allows more parallelism exploration for these devices.

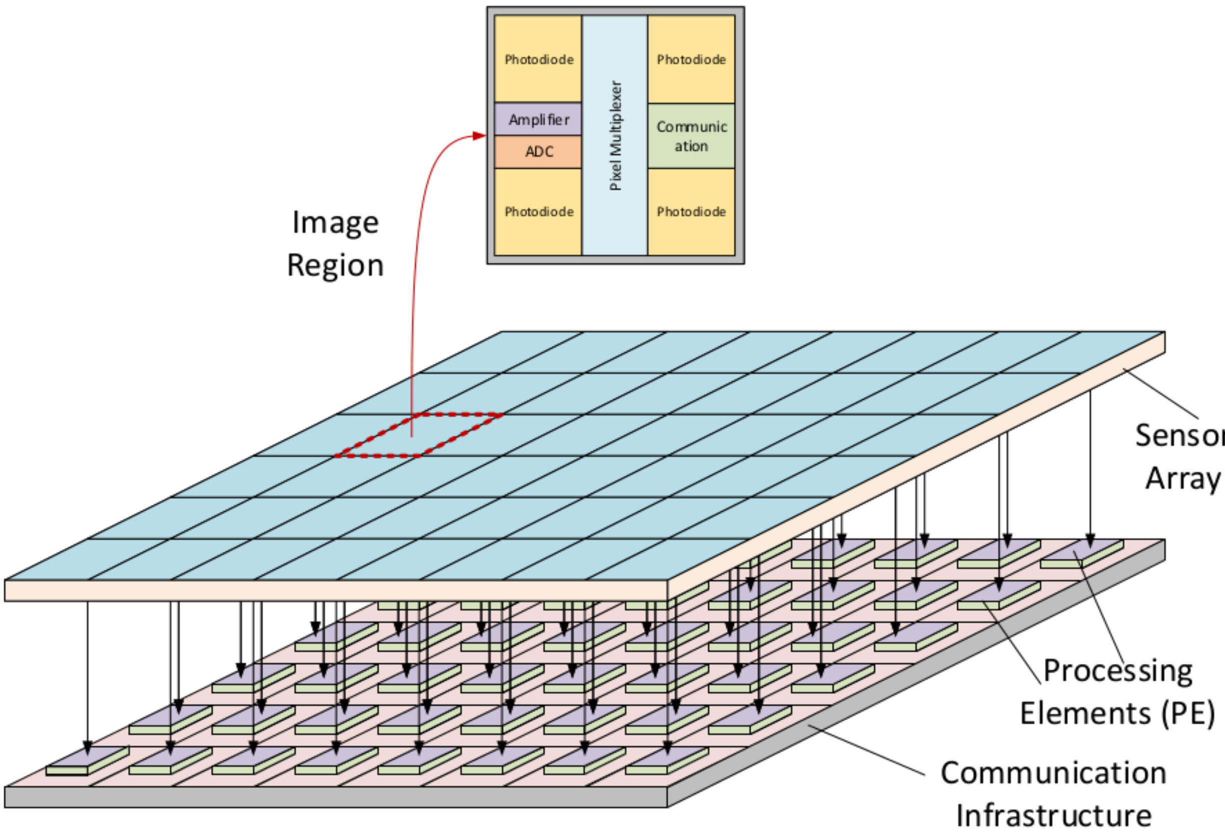

Figure 1.1: The focal plane Vision Processor concept (4)(5)(9).

There are different ways to distribute pixels from multiple CMOS sensors. Figure 1.2 shows different stream architectures. Single Stream uses one ADC to convert all pixels to digital signals. This is the most simple arrangement but the least efficient. On the other side, One Pixel per stream has better throughput but is the most expensive. It requires one ADC per Pixel. Parallel Columns and Region-based are flexible choices as they combine both previous schemes. As discussed in (6), Region-based architecture provides better scaling integration for highly parallel architectures in comparison with the others.

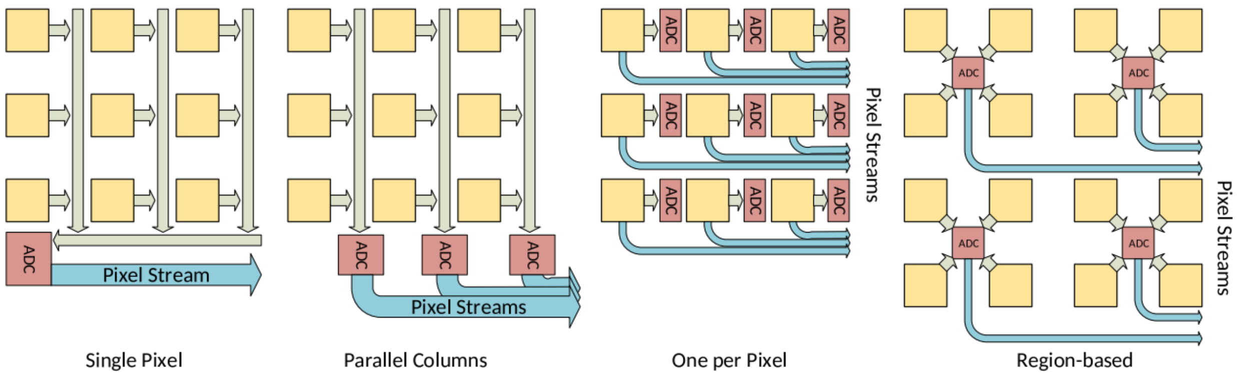

Figure 1.2: CMOS pixel distribution: from single to multiple streams (6).

Historically, mainly due to costs and technological constraints in the VLSI design, the camera organization is usually based on a single-pixel stream from the sensor to the processing systems. Images are inherently parallel, and the single-stream is a bottleneck for the whole system, preventing the exploration of the natural pixel parallelism. Some recent studies show that the utilization of multiple pixel streams can improve the overall performance of vision processors, offering an attractive balance among cost and performance (10)(6).

Allied to the acquisition scheme, there was a need for efficient data communication infrastructure to process multiple pixel streams. Not only the acquisition scheme of Focal-Plane processors demanded better communication architecture, but also most of the VLSI design were facing challenges. At that time, instead of measuring IC progress by improvements in speed and size, energy efficiency started to predominate, leading to a new design paradigm (11). Therefore, the concept of Network-on-Chip (NoC) emerged because of this new trend and advances in SoC designs, demanding a higher level of IP integration and granting a fully distributed communication pattern with almost no global control needed (12).

Networks-on-chip were designed similarly to off-chip networks commonly available nowadays. It has fewer layers than a conventional TCP/IP stack, for example. According to (12), the micro-network layers are composed by,

- *Data link layer*: the physical layer responsible for exchange packages in certain protocols. It represents the protocol used at the hardware level and is very important to ensure reliable data transmission;
- *Network layer*: connects the source from its destination, routing between different network nodes if necessary. Multiple algorithms and methods can be used as real or design-time tools to improve multiple parameters (13);
- *Transport layer*: break messages into packages at the top-level accordingly to the implementation. It is important because packet size represents a significant bottleneck in network design.

Figure 1.3 pictures a simple 2-D mesh Network composed of 9 Intellectual Properties (IPs).

Each IP is connected to the others by its own router (R). Routers can exchange all kinds of messages, depending on the application and design decisions. Furthermore, IPs can be selected to fulfill different tasks for Homogeneous or Heterogeneous computing systems. NoCs have become popular because of their flexibility and scalability. Buses are possible alternatives to NoCs, but their implementation for highly parallel systems become complex as they need an arbiter to control multiple masters.

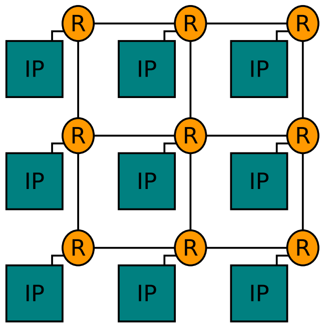

Figure 1.3: 2-D mesh grid Network-on-chip diagram.

There are several different approaches in both hardware and software for the processing part of these new systems. In software, the typical approach is to explore data caching to avoid expensive memory accesses, as well as redundant mathematical operations. Hardware architectures span a broad diversity: ASICs, direct FPGA implementations, Application-Specific Instruction Set Processors (ASIPs), Very-Large Instruction Word (VLIW) processors, Digital Signal Processors (DSPs), Graphics Processing Units (GPUs), Multi-Processor System-on-Chip (MPSoCs), and so on. Each approach is an attempt to explore data parallelism, higher processing frequencies, pipelines, or complex instructions. The last one in special, can be combined with a multiple stream acquisition scheme and NoCs to create a highly scaling architecture for IP/CV applications (6).

In this new VLSI design paradigm, FPGAs became useful tools to prototype computing systems because enable the construction of accelerators to boost some tasks, instead of using general purpose processors. They contain a large number of logic blocks that can be connected by a matrix of switches (14), for example, flip-flops, look-up tables, I/O banks, DSPs, Block Memories, and others depending on device family and vendor. FPGAs also represent a cheap option compared to ASIC design. They offer more performance for application specific tasks and similar flexilibity of general-purpose CPUs.

Xilinx devices play a special role in FPGA design, particularly the Zynq family. It connects FPGA fabric and hardware processors in the same chip. (15)(16) explore multiple Programming Systems (PS) and Programmable Logic (PL) interfaces in Zynq-7000 family chips. Their high throughput, achieving GB/s magnitude from PS-PL transferences, is advantageous to process data-intensive tasks in the FPGA fabric and control-intensive routines in the processor (16).

Most of these FPGAs IP/CV implementations explore optimizations at pixel-level operations to obtain better performance (17)(18)(19). Usually, in this kind of application, the algorithms work in pixel level, transforming a given image in another. The IP/CV operations can be classified in three types, as depicted in Figure 1.4, which is referred as *Pixel level parallelism* along this work(6)(20)(21):

- Pixel operation: gets a single pixel as input and generates a single output pixel;
- Neighbourhood operation: gets a regular group or region of input pixels to produce a single output pixel;
- Global operation: gets a whole image as input to create a single output pixel;
- Non-Regular operation: gets a set of input pixels that shares a common property to produce a single output pixel.

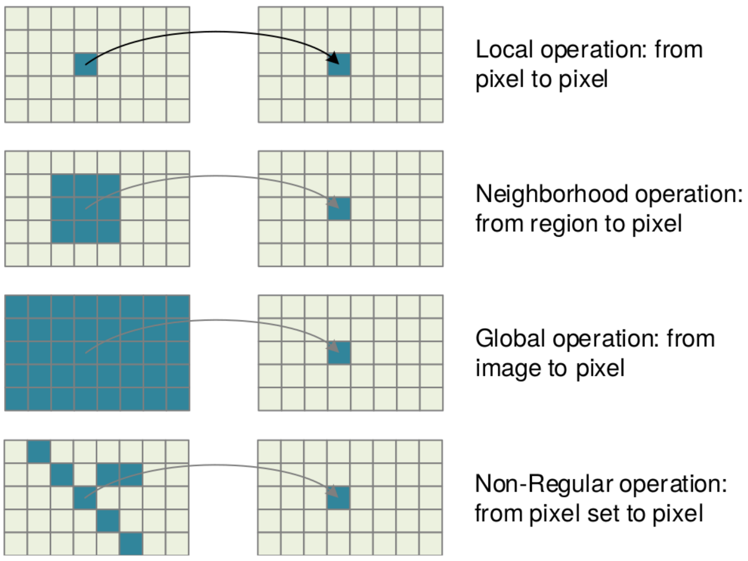
Local operation: from pixel to pixel
Neighborhood operation: from region to pixel
Global operation: from image to pixel
Non-Regular operation: from pixel set to pixel

Figure 1.4: Pixel level IP/CV operations (6).

1.2 CONTRIBUTION

Considering the Smart Camera context and FPGA IP/CV architectures, and based on the findings of (21), that built a SystemC/TLM EDA tool to explore multiple design choices for IP/CV Vision Processors. This work prototypes a Many-core Vision Processor in FPGA as a proof-of-concept, and also evaluates multiple advantages of overlay architectures that combine the PL's performance enhancement to the flexibility and robustness of PS in a Zynq UltraScale+ device.

To integrate acquisition and processing, in this work, we build an architecture able to receive multiple pixel streams and process them in parallel using an MPSoC (Multiprocessor System-on-chip) based on a Network-on-Chip (NoC). All design decisions are based on IP/CV application-specific needs, from the Processing Elements instruction set to the NoC flit composition.

The Many-Core Vision Processor (MCVP) architecture, developed in this work, is responsible for distributing the picture to the tiles and send them to a visualization scheme. Pixel distribution uses a *Region-based* scheme, as depicted in Figure 1.5. Each region has its own Processing Element inside the Tiles, which are responsible for one image region and share information with the others through a NoC. The system encompasses a complete IP/CV pipeline, allowing a realistic scenario to evaluate its performance.

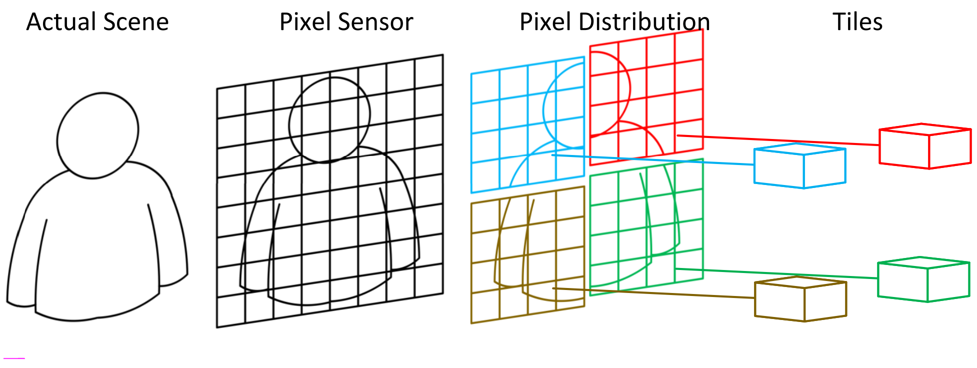

Figure 1.5: The proposed scheme of pixel distribution between Tiles.

The contribution of this work to the state-of-the-art is a programmable, NoC-based, highly scalable and IP/CV targeted Many-core architecture for FPGAs. The results show that multiple-level parallelism exploration improves the system's capabilities and enables more flexible designs.

1.3 ORGANIZATION

The rest of this work is organized as follows:

- Chapter 2 depicts a literature review with the main drawbacks found and the proposals to solve them;
- Chapter 3 explains the HW/SW organization and the main design decisions. The system is depicted and described at a higher level;
- Chapter 4 shows the implementation details of each hardware block. The processor, memory, and network are explained;
- Chapter 5 discloses architectural aspects and implementation details of the toolset with a parameterizable VHDL generator and an assembler designed to scale up the architecture;
- Chapter 6 describes the IP/CV applications, shows the results and comparison with related works. Power consumption, FPGA resources, and timing performance are analyzed for multiple algorithms;
- Chapter 7 presents the conclusion and future works.

2 RELATED WORK

In the literature, there are plenty of works towards accelerating/optimizing IP/CV algorithms, covering soft-related aspects, like algorithm rewriting (22), to hardware-related ones, like GPU systems (23). This review covers recent approaches related to embedded IP/CV systems and HW/SW architectures that implement common image processing tasks, focusing on the Harris Detector algorithm, used in Chapters 6.1 and 6.

2.1 IP/CV SYSTEMS

As IP/CV techniques present massive data and operation parallelism, efficient parallelism exploration is potentially a solution to optimize the performance of such techniques. VLIW processors can perform several operations in parallel, offering a suitable approach. In (24), the authors merge a scalable VLIW architecture with OpenCL parallelization capabilities to build a NoC-based multiprocessor for medical applications on FPGAs. They prioritize better layout and resource usage to model the network, in a data oriented approach, but lack performance for IP/CV applications compared with similar works. Similar to our work, they slice the image in multiple segments, demanding more from the network though.

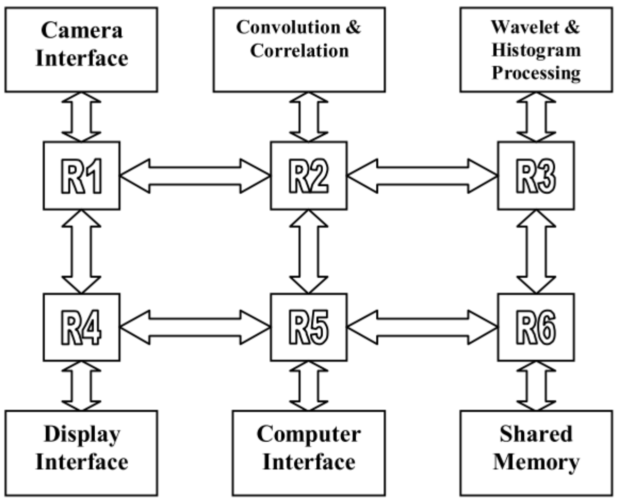

Figure 2.1: Six-port Torus NoC for Image Processing (25).

The development of time-critical systems requires attention to all aspects of the HW/SW architecture. The data transmission among processors represents one of the bottlenecks in any parallel architecture. (26) implements an Image Processing four-port NoC architecture in Virtex II family FPGA, capable of store memory and display results at the same time. (25) presents an improved six-port Torus topology (Figure 2.1) architecture based on (26), each one with different functions: from acquisition to display interfaces, passing through processing units (PUs). Each core interface is composed of FIFOs instead of Finite States Machines (FSMs), different from our work. Even they present an early stage prototype, there is no parallelism exploration with multitasking or image slicing. The authors separates the task in different sub-blocks and passes the whole image data through a NoC instead, which simplyfies the architecture but reduces timing performance.

(27) proposes a NoC for image processing algorithms, Figure 2.2. It is based on a ”token-ring” approach, using one circular unidirectional network to transport commands and results data via an asynchronous network. (26) (25) (27) work explore the scalability of NoC communication, and each PU is responsible for specific parts of the desired IP/CV algorithm, differing mostly by the six-port Torus and token-ring network types. This approach favors local optimization of each PU; however, it does not explore the operation parallelism of the IP/CV data, which leads to the need for much higher operating frequencies. Our architecture organization also explores the NoC scalability features, but besides, by the exploration of pixel-level parallelism, we can operate at lower frequencies, which could represent smaller power consumption. For further discussion about power, read Subsection 6.4.3.

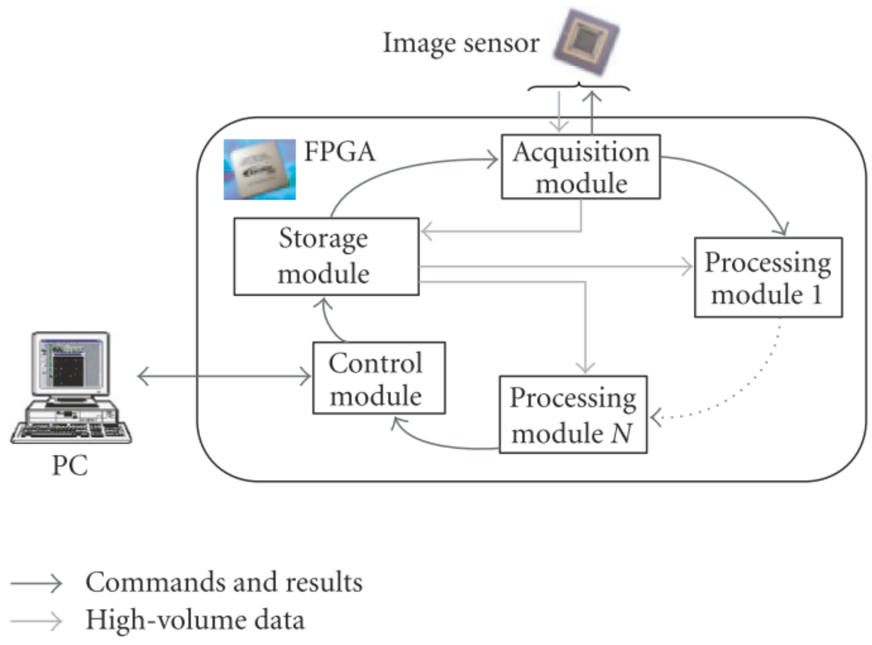

Figure 2.2: Proposed Token-Ring NoC architecture for image analysis algorithms (27).

(28) presents a multi-processor architecture based on a Spidergon NoC topology, see Figure 2.3. It is a heterogeneous processing system, where each tile has its specialization: memory, General-Purpose Processor (GPP), motion estimation, among others. Each tile can reach another one through the NoC, and several different IP/CV algorithms can be implemented using or not all the tiles. There is some task-level parallelism, since the processing tiles can work in parallel, and high throughput is reached. That work could explore the pixel-level parallelism; however, the authors did not consider that possibility.

A NoC-based MPSoC is explored for some IP/CV algorithms in (29), as shown in Figure

2.4. The authors use MPI (Message Passing Interface) to parallelize the algorithms and distribute the threads among the cores. It enables multiprocessing with high-level abstraction in tightly constrained devices and, similar to our work, divides the image in slices and distribute them inside processor's memories. Our approach is also based on an NoC-based MPSoC, however, we have all cores executing the same program in parallel, but over different data sets (pixel regions), instead of the MPI model. We also use a FPGA based architecture instead of an ASIC like (29) does.

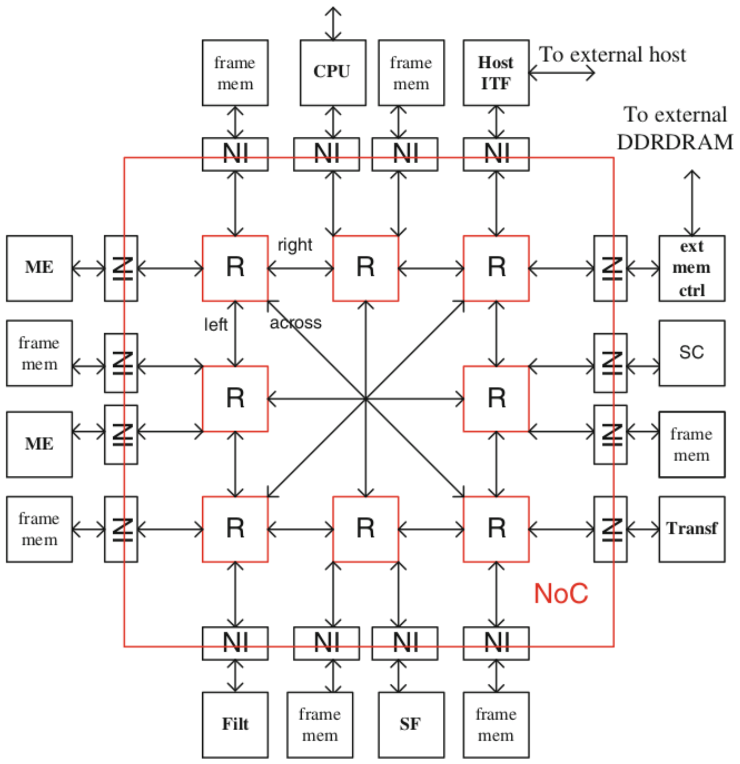

Figure 2.3: Block diagram of the Spidergon NoC (28).

Another relevant architecture type for IP/CV implementations is the GPUs. A GPU is commonly able to explore vector operations efficiently. Several IP/CV algorithms can be expressed in such a way that the compilers are effective in addressing the instruction-level parallelism. In (23) (30), a soft-GPU architecture is presented. Similarly to our approach, the authors developed an FPGA overlay exploring the platform features (DSP blocks, distributed memory, logic blocks, and interconnects) to optimize the architecture design, showing the feasibility of FPGA overlays as an end-user platform. It develops an FGPA GPU (called FGPU) architecture with Compute Units, a basic processing block of the FGPU architecture with 8 custom processing elements to perform SIMD instructions. It implements IP algorithms like Sobel, Compass Edge Detector, and fitler algorithms for benchmarks. Different of our work, the authors develop a general purpose Graphics Processing Unit for FPGAs, focusing on high performance gain and power consumption, instead of an IP/CV specific solution.

The NVidia Tegra Tx1 is a heterogeneous architecture composed of an embedded GPU architecture with 256 CUDA cores and a quad-core ARM processor. In (31), that platform is used to develop a smart industrial camera able to perform real-time object recognition. The authors show that an embedded GPU can efficiently explore the parallelism in IP/CV algorithms and provide high throughput and flexibility.

A general-purpose pixel distributor for parallel processing in FPGAs is depicted in (32). The authors address the importance of such architecture for real-time image processing and the demand for an efficient parallel distribution system to reduce required memory. Stream processing is the primary approach of their work, with specific processing units directly implemented in FPGA, and also exploring the pixel-level parallelism. However, despite stream processors, tiny processing elements based on RISC architectures are used, favoring the flexibility for any IP/CV application.

Some authors explore the FPGA dynamic reconfiguration to improve online flexibility (33). However, the reconfiguration time is way too time-consuming to be used in real-time applications. In our architecture, full flexibility is provided by the use of soft-programmable processing elements.

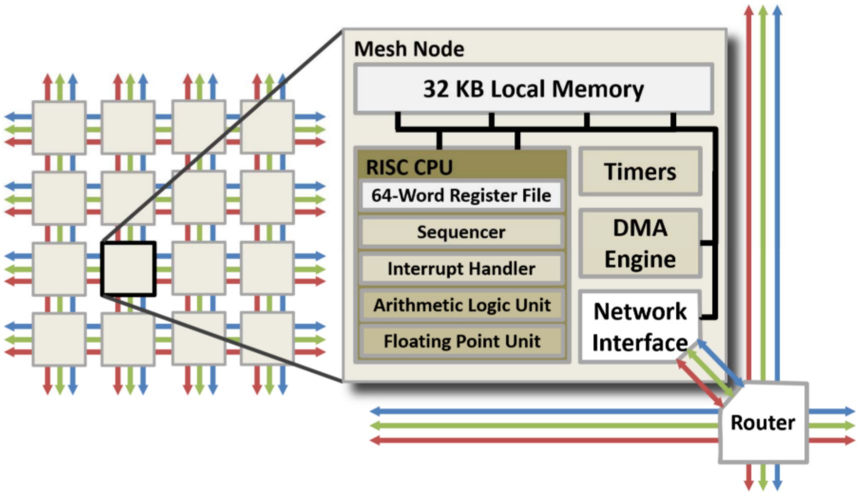

Figure 2.4: 16 core Epiphany III architecture (29).

(6) (9) (34) propose a methodology for the design and programming of next-generation many-core vision processors. The authors suggest new design architectures that optimize multi-cost functions: memory and resource usage, communication cost, power consumption, and hardware speed. Their work computes microarchitectures for the IP/CV applications that explore pixel operation parallelism using multiple iterations and SystemC/TLM models to make better design decision for this kind of applications. Besides, they analyse multiple parallelism aspects in many-core vision processors based on algorithm characteristics, pixel-level, and multiple stages of design space. They explore various subsystems in the architecture to extract better parallelism, develop a simulation tool using SystemC/TLM 2.0, and shows possible architectural choices to improve this kind of processing system. Also, execute the Canny Edge Detector as a reference algorithm, computes the number of operations and memory accesses necessary to the system. The authors build the architecture for 16x16 pixels per tile prototype in FPGA, showing that its viable

compared with other state of the art Canny Edge Detector (CED) implementations. In conclusion, they obtain a many-core architecture based on simulations in SystemC/TLM 2.0 and other design space exploration techniques. Our work concerns with practical implementation issues based on their results to build a viable many-core vision processor on FPGAs based on their findings.

(35) presents a high-level synthesis tool to facilitate time-to-market Heterogeneous MPSoCs design. The hardware architecture combines Microblaze softcore flexibility with HLS practicality to implement multiple designs for different project constraints. Despite the MPSoC implementation in FPGA using the on-chip Network, our approach focuses on lower-level aspects instead of the programming model and high-level synthesis. The choice from (35) reduces design effort but has performance reduction compared to more specialized architectures like ours for IP/CV applications, see Subsection 6.3.3 for details.

(36) sets up a method to program heterogeneous MPSoCs using the Xilinx SDSoC framework and other open-source tools. The application profiles automatic instrumentation of the code to the designer, who takes better decisions if necessary. This approach differs from our work because of its focus on a high-level tool to ease rapid prototyping in Many-core systems. We provide design choices for Homogeneous MPSoCs in Overlay Architectures implementing the same task for all cores.

Reference	PE Type	Communication	Programmable?	Main feature
(24)	VLIW	NoC	yes	FPGA overlay architecture to explore PS-PL
(26), (25)	Heterogeneous	NoC	no	Early NoC proposal for IP/CV applications
(27)	Heterogeneous	NoC	no	Multiple Processing Units in asynchronous NoC
(29)	RISC	NoC	yes	Proprietary MPSoC and use of MPI model
(23)	Soft-GPU	Bus	yes	FPGA overlay resources optimization, use of soft-GPUs
(31)	GPU	Bus	yes	Achieve real-time object recognition with GPU/CPU combination
(6)	RISC	NoC	yes	Spatial parallelism exploration achieves high performance scaling
(35)	RISC	NoC	yes	HLS synthesis tool for a heterogeneous system built for FPGAs
(36)	Heterogeneous	Bus	no	Method to explore HW/SW design choices with few user interaction
Our work	RISC	NoC	yes	Overlay architecture that explores spatial parallelism with region-based processing

Table 2.1: IP/CV reference comparison.

Table 2.1 shows a comparison of different architectures cited in this section with our work in terms of Processing Element type, communication structure, programmability, and main features. Most of the work use NoC for communication, and have programmable devices. This shows the interest of recent IP/CV related architecture with programmability and high-level design, using OpenCL, MPI and Vivado HLS in many of them as a possible way to improve performance and time to market.

2.2 SPECIFIC HARRIS DETECTOR IMPLEMENTATIONS

Several approaches for FPGA direct implementation of IP/CV algorithms can be found in the literature, and most of them are based on stream processing to reach high operating frequencies, as seen in (37) for example. That implements a pipelined Harris Corner Detector architecture to be used in mobile robots for colored stereo image processing. Then, it divides the algorithm into three distinct parts: Cornerness Measures (CM), Sum of Gradient Products (SOGP), and Non-Maximal Suppression (NMS). Like our work, it uses Block RAMs as frame buffers to store frame data, instead of the conventional line buffer approach. The architecture is tested in a Virtex-5 series FPGA and achieve high throughput. However, it only works with HCD, not allowing other algorithms or programmability. (37) builds a HCD specific architecture with no flexibility to implement different algorithms.

(38) implements the Harris & Stephens corner detector in a Zynq-7000 FPGA. It proposes a co-design using the PS for PL control and memory management. The logic design, which is written in Vivado HLS, receives and processes the camera stream with fixed-point arithmetic. The post-processing image is sent to an output HDMI port, forming a complete IP pipeline. The resulting architecture is able to double the performance in comparison with similar work at the publication time (37) (39). Different from our work, it builds an application-specific design using HLS and OpenCV to reduce design-time complexity but lacks flexibility.

(40) builds a real-time pipelined Harris Coner architecture in Zynq-7000, see Figure 2.5. It uses line buffers methodology to acquire, process, and display results. The work also presents the whole CMOS sensor setup and image pre-processing chain and tests it using MATLAB tools. The architecture is capable of performing the pipeline with 154 fps in VGA resolution. Different from our approach, this paper deals with application-specific problems of the Harris Corner algorithm. It implements a common line buffer system and is not programmable.

(18) proposes two pipelined architectures to compute the Harris Corner algorithm. One of them uses one line buffer in the algorithm, while the other uses two in distinct parts. The work utilizes FIFO line buffers and tests the application in Altera FPGAs. Different from our work approach, their work does not develop a flexible design, which focused mostly on FPGA resources trade-offs and not on programmability.

(17) presents an FPGA architecture targeted for real-time and multi-resolution Edge and Corners Detection applications. It compares the proposed FPGA architecture with CPU and GPU-based systems in terms of performance and power consumption for the HCD. It proposes a complete IP pipeline, from acquisition to visualization with an external monitor. Also, it builds the

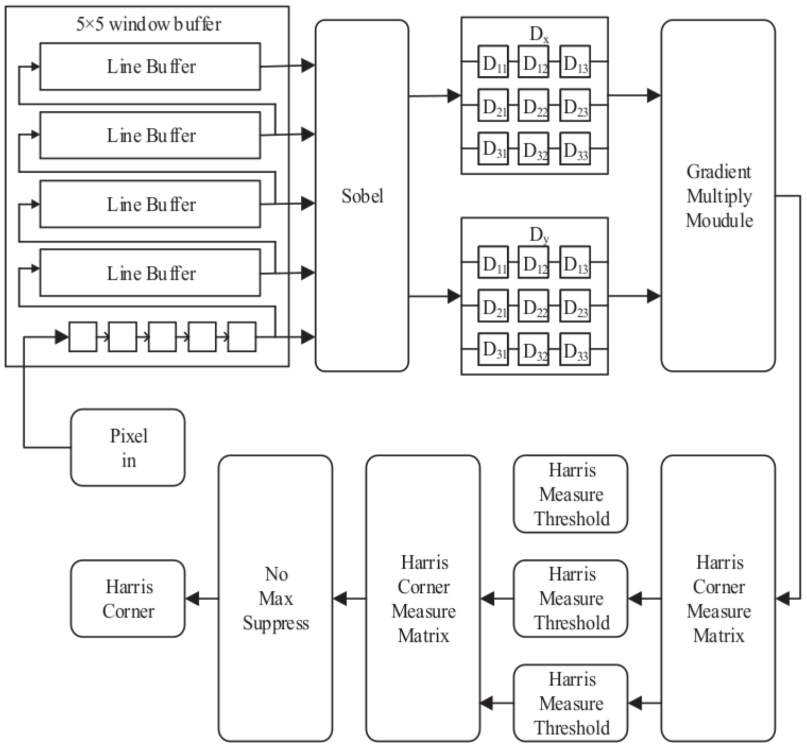

Figure 2.5: Complete Hardware implementation of HCD (40).

architecture on what it calls Neighborhood Extraction (NE) block, which provides a sliding window with a fixed dimension to the next processing block. The NE is made of register arrays that store pixels to be processed and line buffers to store the actual and two or more store pixels in the block. This structure is based on a Coordinate Counter to provide the correct output selection based on the window position. It is also able to outperform CPU and GPU for timing and power consumption for both Harris and Canny detectors. Different from the approach of our work, it uses a fixed design specific to some detectors.

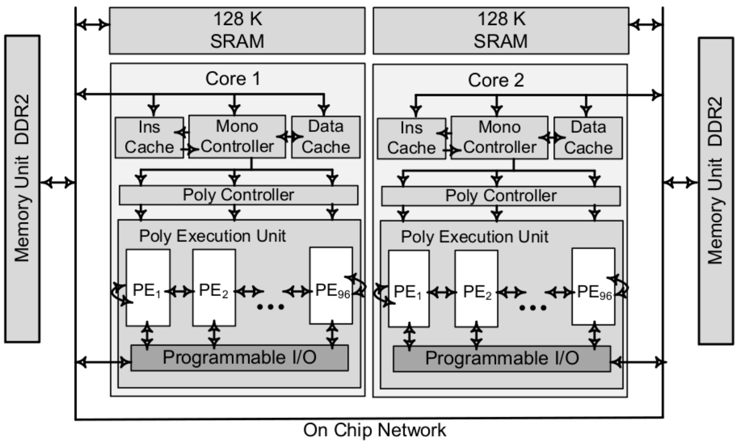

Figure 2.6: Dual core CSX Chip architecture (41).

(41) implements a real-time Harris Corner Detector implementation in a low-power SIMD architecture named ClearSpeed CSX700. There are two cores, which have a RISC based processor

coupled to what is called Poly Execution Unit: a highly parallel SIMD architecture consisting of 96 processing elements, refer to Figure 2.6. The work divides the image in two parts, one for each core, and these subimages are also splitted in 96 rows to use all available cores. The approach can perform 142 fps for HD resolution, but different from our work, it shows an optimized implementation of the HCD for the CSX700 chip.

(42) builds a mobile targeted HCD with a Tegra-based GPU. It implements the algorithm in GPU-enabled functions of OpenCV and ArrayFire in Jetson TK1 and TX1 and only OpenCV for Tegra X2. Also, the authors compare the results with MATLAB implementations for precision checking. They evaluate multiple chessboards like images with multiple corners and HD, Full HD, and 4k UHD. The device achieves real-time processing and up to 51.32 fps in 4k for Tegra X2. While their work can achieve high performance, it is built on a fixed GPU architecture that usually consumes more power than typical CPU applications (30).

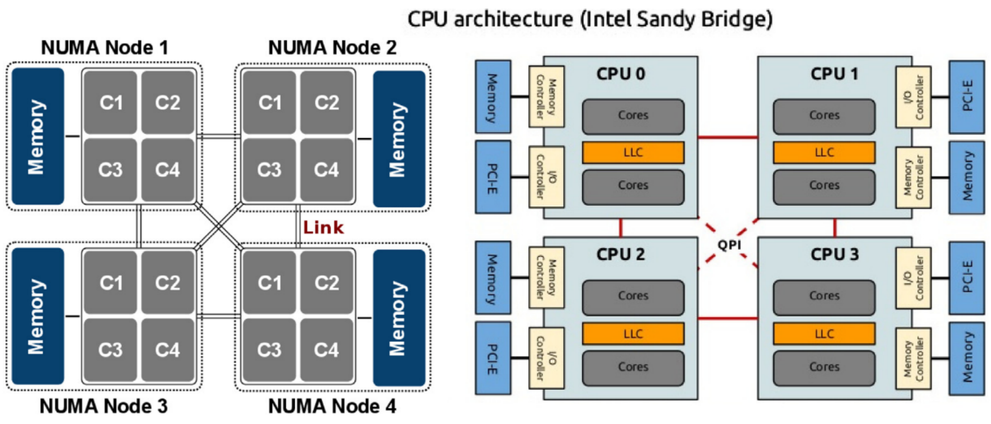

Figure 2.7: NUMA configuration with 4 cores (43).

(43) uses a Non-Uniform Memory Access (NUMA) architecture to perform the HCD with a serial combination of 3x3 derivatives and Gaussian filters, as shown in Figure 2.7. Optimizes the algorithm with stencil computation, an algorithm class that maps a neighborhood of inputs on a specific output. It utilizes two Intel Xeon processors to validate the proposed method. Similar to our work, it performs memory aware parallelization, but does not concern with HW architectural aspects and power-constrained systems, which is important for embedded applications. Instead, (43) cares of software architectural aspects for High Performance Computing (HPC) and high end computing systems.

Table 2.2 compares the HCD implementations with our work in terms of architecture type, programmability and main features. It shows a predominance of non-programmable FPGA architectures for the Harris Detector. Most of these FPGA architectures use common Image Processing structures, like line buffers. Non-FPGA implementations make use of software optimizations to improve performance, as a differential.

Based on the presented analysis of the literature, our work designs an architecture to explore the intrinsic spatial pixel parallelism using a *Region-Based* acquisition scheme (Fig. 1.5). As seen, there is a lack of works that explore SW for specialized IP/CV HW to improve applications

Reference	Architecture	Programmable?	Main feature
(37)	FPGA	no	Colored stereo HCD pipeline in a limited device
(38)	FPGA	no	HLS tool to optimize performance
(40)	FPGA	no	Complete pipeline implementation tested with MATLAB
(18)	FPGA	no	Hybrid line buffers to reduce resource usage
(17)	FPGA	no	Neighborhood Extration block and CPU/GPU comparison
(41)	SIMD-ASIC	yes	Explore parallelism using multiple PEs available
(42)	Embedded-GPU	yes	High-level libraries to simplify the implementation
(43)	CPU	yes	Explore non-uniform memory with stencil computation
Our work	MPSoC-FPGA	yes	Overlay architecture with region-based processing

Table 2.2: Comparison of different HCD implementations present in this work.

performance. Also, few works explore the use of manycores to enchance parallelism exploration while keeping programmability. Combining these three aspects, we propose to build a FPGA manycore architecture optimized for IP/CV applications. Every Tile of our architecture is responsible for one image region and is composed of a Processing Element, a Router, and a Pixel Memory. To address communication issues, a Network-on-Chip handles data transmission among the PEs using application-specific message packets. All PEs run the same binary code using Special Registers (Sec. 4.2) to delimit the internal image of each Tile, also referred to as Tile's sub-image. Image parameters are static and defined during the design-time process.

The next chapter explains in-depth the HW/SW architecture developed for the FPGA overlay and the complete IP/CV pipeline organization.

3 THE HW/SW PLATFORM

This chapter explains the IP/CV pipeline implemented in this work, as depicted in Figure 3.1. We selected a ZYNQ Ultrascale+ device (ZCU104 development kit from Xilinx), a state-of-art SoC (7). It has a camera with an acquisition IP block, which stores pixel data in an acquisition frame buffer (FB). After that, the Many-Core Vision Processor reads the frame from the acquisition FB, processes it, and stores the final frame in the visualization frame buffer. Then, using the AXI Stream interface, the final frame goes to an AXI DMA IP that saves the data in ARM's DRAM. This approach reduces the ARM processor load because it is only responsible for each IP control but does not deal directly with the data transfer.

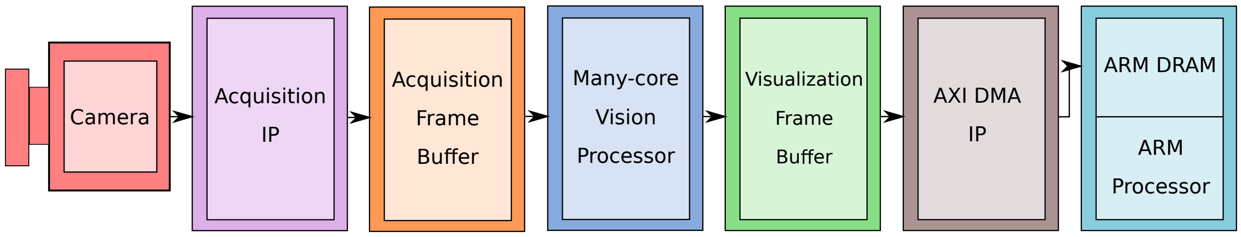

Figure 3.1: The many-core architecture block diagram.

The manycore architecture proposed in our work is designed thinking on multiple pixel streams being sent in parallel to each manycore tile. There is no comercial image sensor that provide Region-based pixel streams. Due to its limitation in sensors manufacturing, we chose to make a proof of concept using available and affordable image sensors. An FPGA is used as the base platform since this work strives to validate the architectural concepts.

The ZYNQ Chip receives the pixel stream in the Acquisition Frame Buffer, which is responsible for pixel distribution among the manycore tiles. After the image processing, the output image is stored in the Visualization Frame Buffer, which is then read out by an ARM processor, integrated into the SoC for algorithm validation.

Next sections break down each block of Fig. 3.1, except for the MCVP that is treated separetly in Chapter 4. But before, a quick explanation about design parameters adopted along the work.

3.1 DESIGN PARAMETERS

A general IP/CV processing chain usually has multiple steps, where the initial image is acquired, pre-processed and manipulated to obtain the desired final image for a specific application, as shown in Figure 3.2. Some parameters are important no matter the algorithm or architecture in use for these IP/CV applications. First, the common division of image size by width and height, usually defined in pixels. They are important because give a raw estimative of the computational cost for a given chain. For example, considering a Harris Detector in stardard-definition compared to an implementation using 4K resolution. Of course, the last one demands much more resources, time and power.

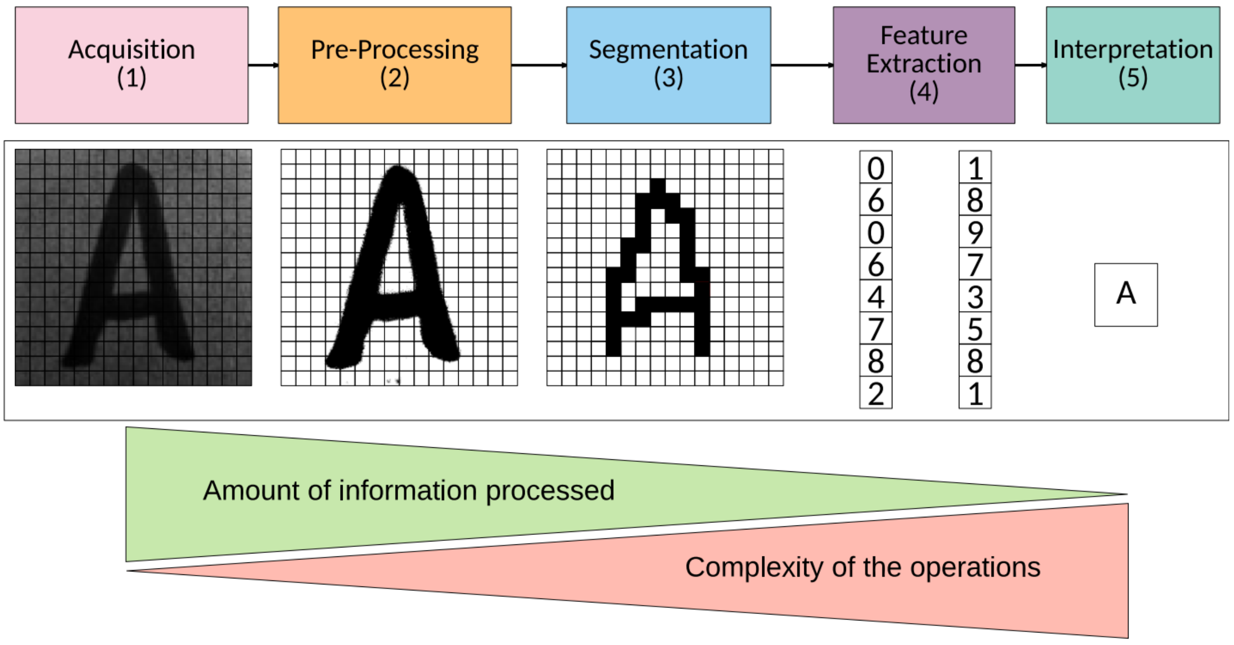

Figure 3.2: General IP/CV processing chain and its complexity (4) (6).

Second, and Figure 3.2 even depicts it, are the different steps of a processing chain. In this example, Acquisition is the first step of the application, Pre-processing the second, Segmentation is the third, Feature Extraction the fourth and Interpretation the fifth and the last one. The amount of information processed usually decreases for higher steps, while the complexity of their operations tends to increase. For general IP/CV chains, we defined the first step as the inital, the last step as the final, and intermediary steps as the others (steps two, three and four in Figure 3.2).

Consider a simple application, with only a pre-processing routine to reduce some noise from an acquired image and display it somewhere. Besides, in another application, consider the same previous pre-processed image this time used to complete a Canny Edge Detector (44), leading to segmentation and extraction steps. The last one has more phases to complete the IP/CV chain, while the initial has fewer step. Thus, we conclude that processing chain steps are important parameters when designing IP/CV applications.

Third, suppose that someone wants to implement a stereo vision visual odometry algorithm to estimate the distance a robot has traveled (2). Figure 3.3 shows the example of feature correspondence in highway pictures, from a stereo pair from current time (top), compared with a stereo pair from previous time (bottom). Numbers in red are frame parameters for this application, endorcing the need of this parameter to address different frames in multiple image applications. This application needs two pair of stereo images, one pair of the previous time step (frames three and four) other pair from the current one (frames one and two) to match features correspondence between them. Therefore, there is a need for frame parameters in algorithms that require multiple images at the same step.

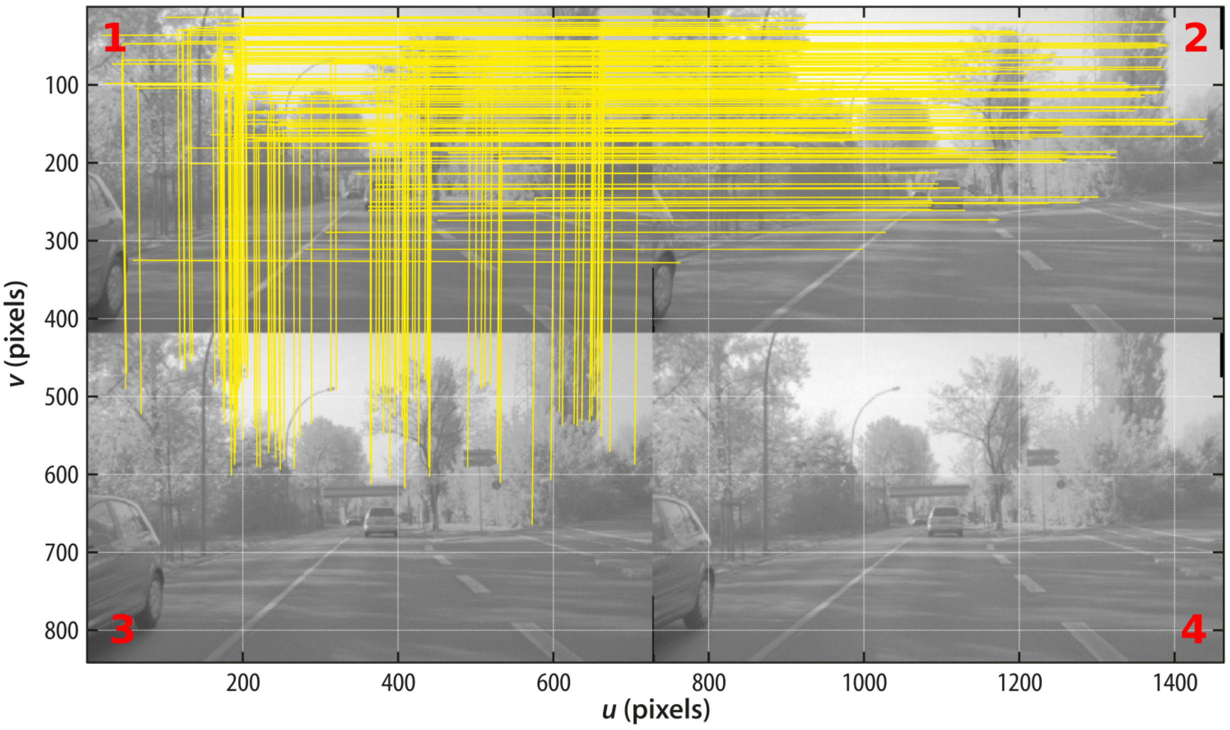

Figure 3.3: Feature correspondence for visual odometry. Stereo pair for current time at the top. Stereo pair for previous time at the bottom. Numbers in red are frame parameters, based on (2) with modifications.

Finally, a fourth parameter type is the data type and size that concerns to a lower level processing but has impact over computing architectures in general. Consider a 9 bits signed integer corner detector implementation compared to a float 64 one. The first has better power and performance but lacks precision compared to the last. Finally, we can summarize our architecture design parameters as following:

- **Image Width** in pixels and in bits;
- **Image Height** in pixel and in bits;
- **Step number** of processing steps in the IP/CV chain;
- **Frame number** for algorithms that need multiple images in the same step;
- **Pixel data size** in bits, defined as 9 (8 bits to store pixel intensity and 1 for the signal) along our work.

3.2 CAMERA

This work uses the OmniVision OV7670 CMOS sensor (45), with specs shown in Table 3.1. The camera is set to QVGA resolution, 320x240 pixels, can achieve 60 frames per second in current setting. It outputs pixels in RGB444 encoding, which is therefore converted to grayscale in a hardware IP.

Table 3.1: OmniVision OV7670 specifications (45).

Parameter	Value
Resolution	VGA, CIF and below
Optical format	1/6”
Pixel size	3.6 x 3.6 μ m
Frame rate	30 fps @ VGA
Output formats	YUV
	RGB
	RGB Raw data
Colored	yes
Power Consumption	Active: 60 mW
Package	24-pin CSP2

The 24 pins are connected to the ZCU104 board through two PMOD interfaces, following pinouts from Figure 3.4, using the LVCMOS33 I/O standard with primary power supply voltage (V_{CCO}) of $3.3V$ (46) for both input and ouput pins.

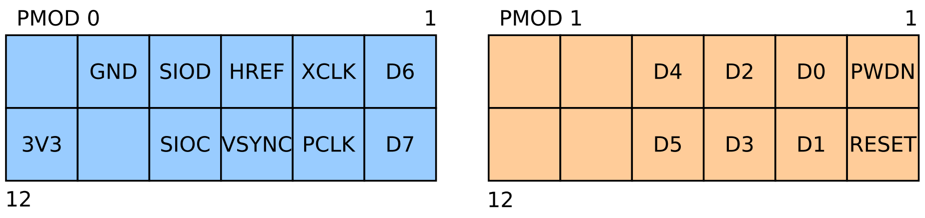

Figure 3.4: Camera pinout in ZCU’s PMOD interfaces.

3.3 ACQUISITION IP

Figure 3.5 shows what is called the Acquisition IP, which is a group of different modules: to control the CMOS sensor, receive pixel data, and to store it in the Acquisition Frame Buffer. This scheme uses an open-source project (47) as the main reference with modifications.

Five modules, i.e. AXI Camera Control IP, Debounce, OV7670 Capture, RGB444 to Grayscale and OV7670 Controller, compose the IP, either for control or data sampling. OV7670 Controller module is responsible for the CMOS initialization. Pins SIOC and SIOD sends data and clock to an i2c like proprietary interface known as SCCB. The controller sets the image size, output data format (RGB444 in this case), prescaler, contrast, gamma, UV auto adjust, image orientation, color conversion, VSYNC/HREF setups, and other configurations. The RESET pin is responsible for the camera reset. PWDN is an active-high pin to select power down or normal mode. XCLK is the clock source responsible for synchronize control signals between the OV7670 Controller and the camera.

The SW pin is a hard-switch to enable the Debounce module and hard-reset the Controller, if necessary. The Debouce block is a custom feature that enables external control over the acquisition scheme, resetting its settings when necessary. PCLK is the Pixel clock source to synchronize pixel data with the Capture module. VSYNC is the vertical sync pin, which flags the end of a frame. Similarly, HREF indicates when an image line ends. These two pins give information about each pixel coordinate relative to the final picture. Finally, the DATA pin carries on pixel data in 8 bits for processing in the OV7670 Capture module.

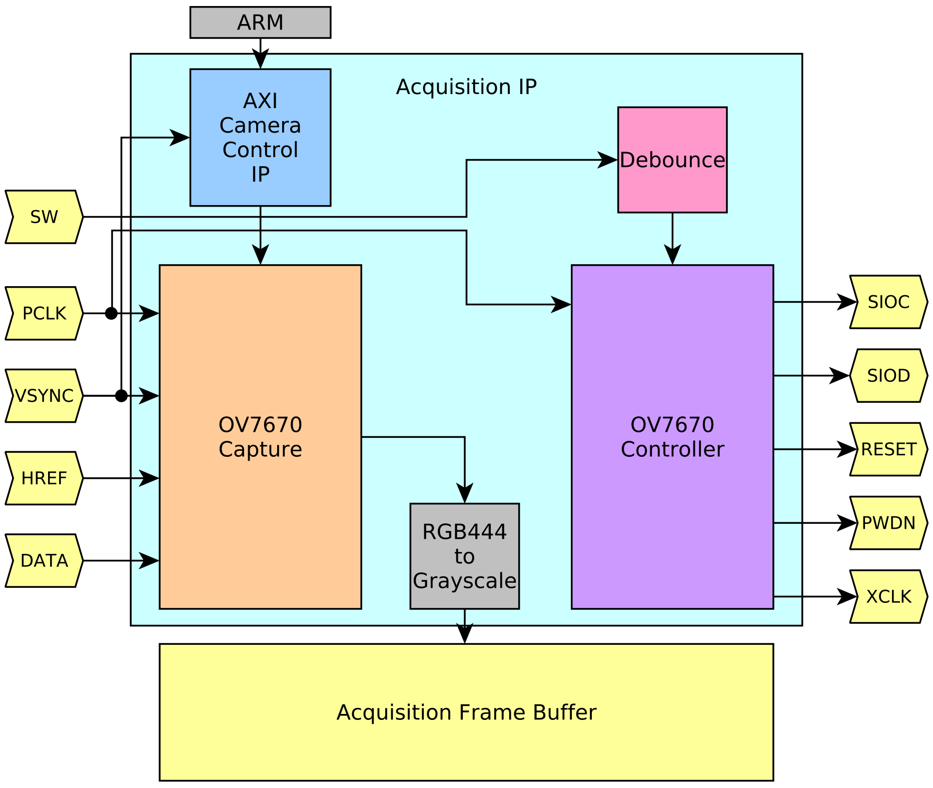

Figure 3.5: Acquisiton IP block diagram.

The OV7670 Capture gets RGB444 data from the camera and sends it to the RGB444 to Grayscale module. It converts from RGB format to Grayscale accordingly to the equation,

$$gray = 5.1r + 10.2g + 1.74b \tag{3.1}$$

where r, g, and b are intensity values of red, green, and blue lights measured by the CMOS sensor, respectively. Their coefficients give appropriate weights for more important colors in the visible spectrum. These values are get applying a commonly used RGB444 to Grayscale equation (48), with approximations for hardware implementation. Also, the values were normalized to a 255 maximum intensity level because they have four bits and are limited to 16 (2 to the 4th power).

The pixel grayscale value is finally stored in the Acquisition Frame Buffer, refer to Section 3.4. Hence, the hardware architecture of the Acquisition IP is almost covered, missing the last module named AXI Camera Control IP. This entity is different from the others because it has integration with the ARM processor and was implemented with the HW/SW integration in mind. This IP has two functions: stop frame acquisition in the Capture module and inform the ARM processor at the end of a frame using the VSYNC signal. Appendix A has the register space description for the Acquisition IP.

3.4 ACQUISITION FRAME BUFFER

Acquisition Frame Buffer uses true dual-port block RAMs with two write ports in read-first mode (49). It is responsible for interfacing the Acquisition IP with the Many-core Vision Processor. There is no need for any AXI port to control the module because it is simply the Dual-port BRAM. The FB has three parameters modifiable in design time:

- **Address Size**: defines the bit length of the address field;
- **Memory Size**: sets total pixels storage capacity;
- **Pixel Size**: assigns pixel bit length, from design parameters in Section 3.1.

3.5 MANY-CORE VISION PROCESSOR

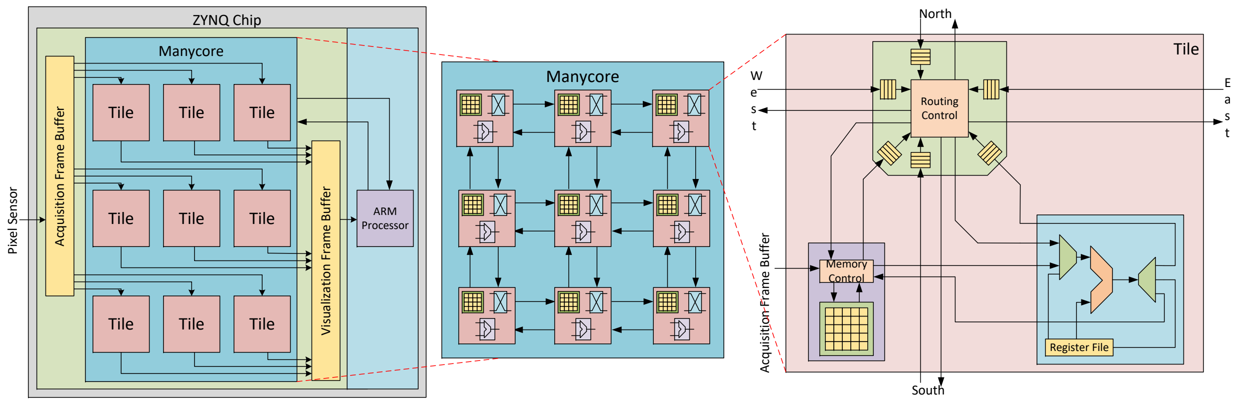

Figure 3.6: The many-core hardware architecture designed in our work (7).

This section describes the big-picture of the Many-core Vision Processor architecture and its HW/SW integration, as shown in Figure 3.6. Tile's internal structure is described in Chapter 4. Figure 3.7 depicts the Vision processor architecture. It is made of basic units, called Tiles, that process, store pixel values in memory, and communicates with the other Tiles using a router. Multiple tiles form a 2D-mesh Network-on-Chip, which transmits pixel data and control messages, forming a homogeneous many-core processing system. We decided to run the same binary code in all Tiles to explore natural pixel parallelism, and because we identified that it fitted beter in a

Region-based image division. What differs from each other are pixel parameters inside special registers, accessible at programming time (see Section 4.2 for more details). These values are stored in hardware, available after implementation, static for a specific architecture and image size, and read-only during the run-time. We envision to create an application-time parameterizable manycore to address the network and image sizes customization for future works.

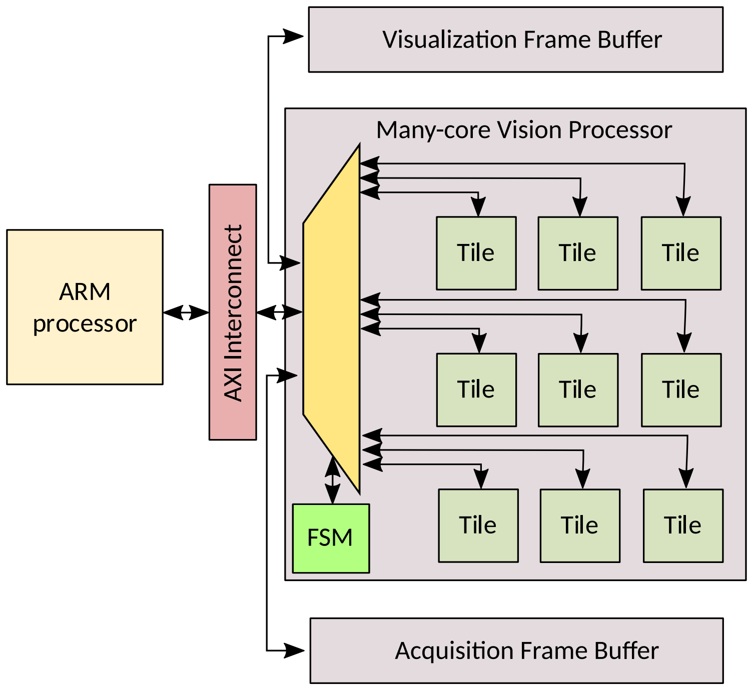

Figure 3.7: Pixel Interface that links the Many-core architecture, the Frame Buffers and the AXI4-Lite interface. It is represented by arrows.

The ARM processor acts like a master from the manycore perspective. It controls the IP operation modes: writes programs in Tile's Processing Elements, transfers pixels from Acquisition FB to Visualization FB, and reads useful data from benchmarking registers when needed (explained bellow). Because of the ARM processor importance to the manycore, we divided their interface in two: the Pixel Interface, responsible for pixel transfers, and the General Purpose Interface, in charge of Processing Elements programming and another useful data transmission. We aim to simplify system understanding with this distinction, but the reader should keep in mind that both use the same AXI interface and similar ARM firmware to control them.

Pixel Interface (PI) architecture appears in Figure 3.7, which has the function to transmit pixel information to and from the Many-core and is a subset of the complete many-core from Figure 3.6. Note that the Acquisition Frame Buffer interfaces with all tiles. There is a multiplexer inside the MCVP logic that selects the correct tile with the desired pixel coordinate (x,y) of the Acquisition FB. This is made by the Many-core's Finite State Machine (FSM) and controlled by the ARM processor. For example, for a 3x3 NoC (like the one in Figure 3.7) and 192x192 image, each tile stores 64x64 pixels. Consider that the FSM wants a pixel from (2,2) coordinate. Then, it is going to multiplex its input to connect the tile from top-left (responsible for this region) with the FB and write the pixel inside the correct Pixel Memory.

The same goes for Visualization Frame Buffer, but instead of writing pixels from FB in all Pixel Memories, the FSM reads the final image and writes it in Visualization FB. An internal demultiplexer, similar to the one used in Acquisition Frame Buffer, selects the desired tile, and communicates with its PM. This is made using pixel coordinates (x,y) as reference. In both cases, a Finite State Machine (FSM) controls the multiplexer. First, it reads the image from Acquisition FB and stores inside the corresponding Pixel Memory. After the Many-core finishes to process and generates the final image, the FSM transfers the final pixels to the Visualization FB.

Figure 3.8 pictures the second interface, with an internal bus, defined as the General Purpose Interface (GPI). The ARM processor programs the Processing Elements using it. Also, GPI is responsible for signal control (enabling/disabling memory interfaces), monitor task execution, and access benchmarking counters. These counters belong to the tile's internal structure and measure: the Many-core execution time, Network overhead time, and Pixel Memories overhead time. They yield information about memory access and network use. The GPI works like a JTAG interface of common processors, in terms of usability.

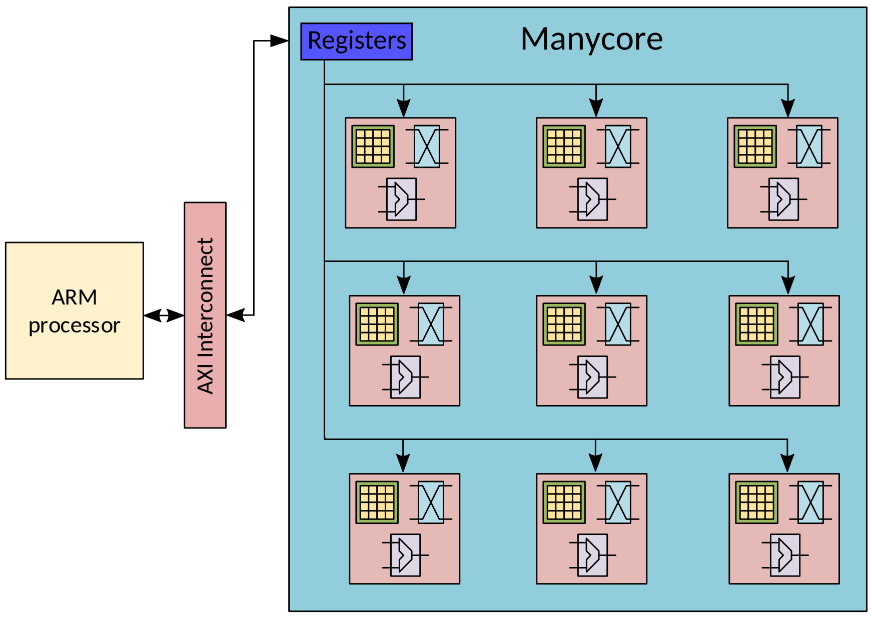

Figure 3.8: Internal bus to interface with the ARM processor. Defined as General Purpose Interface (GPI).

The ARM processor is used to configure and program the tiles, as well as for monitoring and debugging. It communicates to the FPGA fabric through a single AXI4-Lite interface, with 56 configuration words, to transfer data to all tiles. This number of words is fixed and does not change with the number of tiles. With this interface, it would be possible to support up to 65,535 ($2^{16} - 1$) tiles because of the address signal size (see Subsection A.2, SCR.MODE field) to select each tile individually. The address of value 0 is reserved for use when the interface is idle, that is the reason for the minus one in the total supported tiles. Through this interface, the ARM processor has access to pixel memories (PMs), instruction memories (IMs), and special-purpose registers for control and debug purposes. PMs and IMs are explained in Chapter 4.

There are some parameters in the MCVP IP to define important variables in design time, including the design parameters from Section 3.1, there are also two related with AXI interface:

- **Axi Address Width**: AXI4-Lite interface address size in bits;
- **Axi Data Width**: AXI4-Lite interface data field size in bits;

The Appendix A explains the Many-core Vision Processor's registers. It makes more clear the limitations and function of this IP. It is relevant to highlight that both the ARM processor and the AXI bus do not take part in image processing algorithms. The ARM processor access the manycore, as well as all other IPs that use AXI4 interfaces (not including the Stream), as memory mapped devices with specific base address and offsets to reach all registers. Hence, these registers are nothing more than memory addresses from a hardware perspective, but we name them here to ease explanation like usually appears in microcontrollers, IPs and peripherals in general (50) (51).

3.6 VISUALIZATION FRAME BUFFER

Visualization Frame Buffer controls pixel transfer from the Many-core Vision Processor to the ARM processor, after the MCVP ends the processing of a frame. It has one AXI4-Lite interface to control and read relevant data from Visualization FB. Besides, a Memory interface, made of a true dual-port block RAM with two write ports in read-first mode (49), stores pixels written by the MCVP. This IP deals with two interfaces to link the manycore with the ARM processor: a memory interface from the MCVP, and an AXI4 Stream interface to communicate with the AXI DMA IP.

The Memory interface has memory address, data, control, and status signals. It connects directly to the Pixel Interface, where a special control signal, called Stream Enable (SE), starts the FSM. SE keeps low most of the time but gets a high logic level as soon as the MCVP ends the algorithm processing. All Processing Elements need to finish the task to trigger the SE signal. With this arrangement, there is no need for ARM intervention, even to initialize and start the Visualization Frame Buffer.

An AXI4 Stream interface sends data without ARM direct intervention. This approach reduces the processor load but demands a new IP to convert the stream to memory-mapped transfer (AXI DMA IP) and a Finite State Machine to read pixels from the Block RAM and send it through AXI Stream interface. Figure 3.9 depicts a diagram of the Frame Buffer.

The FSM also has a memory interface to connect to the other side of the Block Memory. It is similar to the Pixel Interface's FSM, in which signals scan all memory pixels, stores it one at a time in an internal register, and sends the data to the Stream interface. The FB has multiple design time parameters that allow different image resolutions, including all explained in Section 3.1 and the following:

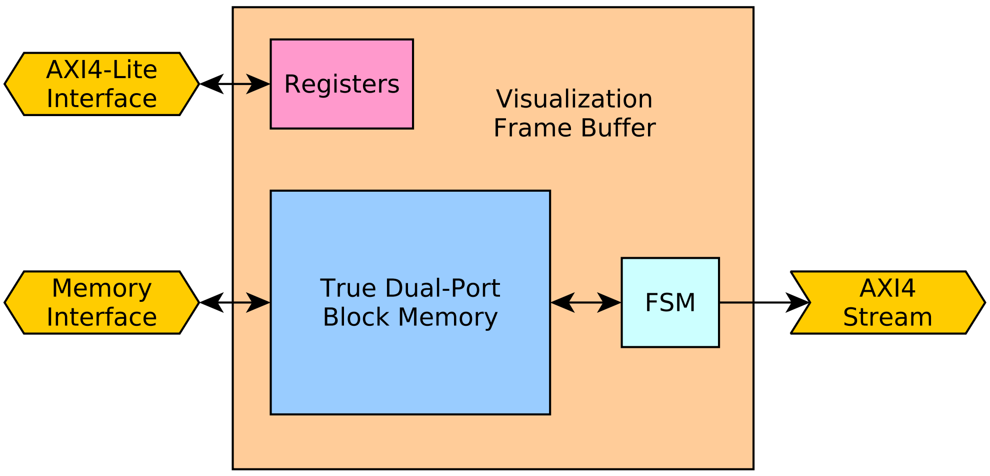

Figure 3.9: Visualization Frame Buffer diagram.

- **Axi Stream Start Count**: number of clock cycles to delay the first stream data transmission, the default value is 32 cycles;

A simple AXI4-Lite deals with the HW/SW integration. It has only one register, but it can be upgraded if necessary in the future. Currently, this interface has one register to store a flag from the FSM, named Stream Done (SD). This signal tells the processor that Visualization FB finished transfering the image, and it is available in DRAM. Assuming the AXI overhead being insignificant relative to the total time necessary to send the whole image, there is no significant delay between SD and when DMA finishes its task. The Appendix A describes the Visualization FB register space in detail.

3.7 AXI DIRECT MEMORY ACCESS IP

This work uses the AXI Direct Memory Access (DMA) IP version 7.1 (50) to write inside ARM's DRAM. Figure 3.10 illustrates the integration between AXI DMA and the ARM processor. DMA operates in Scatter Gather Mode, where *descriptors* changes the behavior of the DMA, setting DRAM addresses and the number of bytes to transfer. These entities act like an instruction memory for the DMA, controlling its functioning to behave as the designer wants. For this reason, the diagram in Fig. 3.10 needs the Block Memory and BRAM generators. The ARM processor writes these descriptors in Block Memory at initialization time and sets DMA registers through an AXI slave interface, through the AXI Interconnect IP. The DMA runs in Scatter Gather mode with ARM interruption if necessary.

After programmed and initialized, the DMA starts to operate: interprets the descriptors in the Block Memory Generator IP, reads data from the AXI Stream input interface, and sends pixels to the ARM processor via an AXI High-Performance Full Power Domain (HP FPD)(52) slave port (red arrows path in Figure 3.10). The work uses this model to deal with the DMA to reduce ARM's load. Another way to use it is to interrupt the processor every time a DMA iteration is complete. Them, the IP needs to be reprogrammed and so forth. Using descriptors, there is no need to interrupt the processor and reprogram anything, although it is still possible to interrupt the processor if necessary. However, a disadvantage of this approach is to depend on Block Memories to store the descriptors, even not using much space. In cases where a board has memory constraints, the interruption-only method is better. Otherwise, Scatter Gather mode is usually the most common setting.

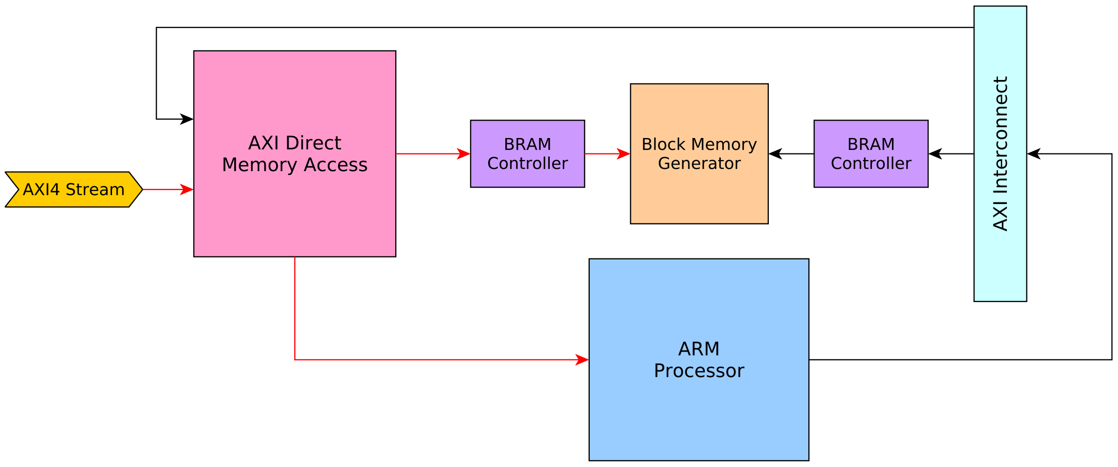

Figure 3.10: Diagram of AXI DMA and ARM processor connections, red arrows show data path during processing time.

In practice, this work combines the interruption with the power and scalability of Scatter Gather mode. The processor is interrupted every time the DMA ends an iteration. The descriptor is very simple, containing just one Buffer Descriptor (the minimum size of this entity), can be used to deal with more complex scenarios, sending data to different DRAM segments if necessary. In the implemented design, one memory segment is reserved for the image, but the architecture can handle even multichannel cases. This situation can deal with two distinct cameras for stereo computer vision, for example. Appendix B presents the programming sequence for the complete IP/CV processing chain architecture.

3.8 ARM PROCESSOR

This work uses a Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC with an ARM Cortex-A53 Based Application Processing Unit (APU) from the ZCU104 development kit, like shown in Figure 3.11. It has four ARM Cortex-A53 processors with floating point, advanced single instruction multiple data (SIMD), and cryptography extensions. Also, each A53 includes 32 KB of Data and Instruction Cache, timers and debuggers peripherals.

The ARM processor runs a bare-metal implementation, using only a single core from the four available. Two High-Performance Full Power Domain (HPM FPD)(52) AXI Master interfaces control the peripherals described in previous sub-sections. One High-Performance Full Power

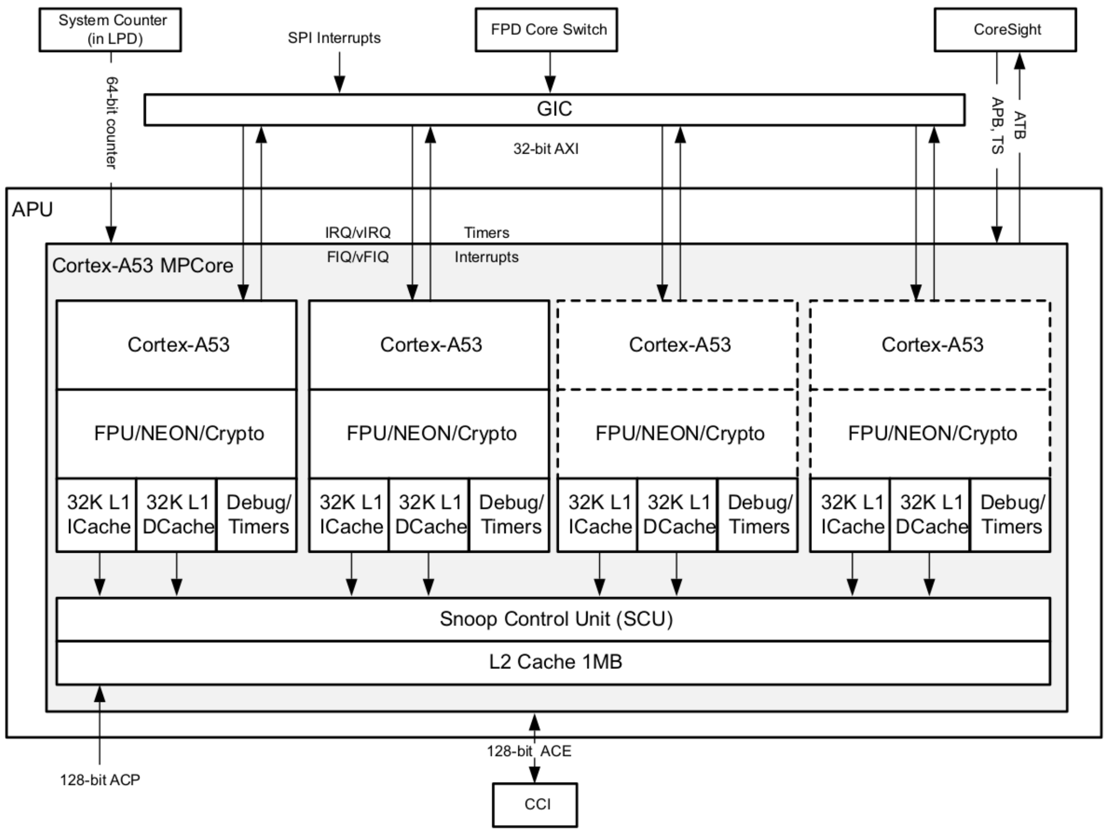

Figure 3.11: ARM Application Processing Unit (APU) block diagram (51).

Domain (HP FPD)(52) slave interface receives the data from DMA IP. The Coherency slave port has worse performance because the application transfer more data than the cache supports. Besides, this work does not deals with multiple cores in Programming System (PS) and does not require complex coherency functionality.

For a task done by an accelerator, which is the case of this work, and for data larger than the cache size, the HP slave interface is the better choice in terms of performance and power consumption(15). Also, for PS-PL data transfer, (16) shows that bare-metal has a better performance compared with OS using CPU.

The ARM has PL-PS interruption enabled to address DMA IP needs. Besides, the initialization process and other low-level tools are all dealt with by Xilinx proprietary tools: Vivado Design Suite to synthesize and implement the architecture, and Xilinx Software Development Kit to handle software and FPGA setup. Appendix C shows the complete IP/CV architecture built in this work in the Vivado Design Suite. All components of the hardware design, explained in this chapter, are depicted there.

4 TILE COMPOSITION

This chapter describes the implementation details and architectural aspects of the manycore vision processor's tile. Figure 4.1 shows the tile architecture composed mainly by PE, PM, and Router. The blocks have specific characteristics to improve performance and resource usage trade-off. The Pixel Memory is responsible for storing pixels of input/output images, as well as all intermediate images generated in an IP/CV chain, as explained in Section 3.1. The Processing Element performs arithmetic/logic operations over the pixels. The Router provides pixels to/from neighbor tiles, composing a NoC with the other routers.

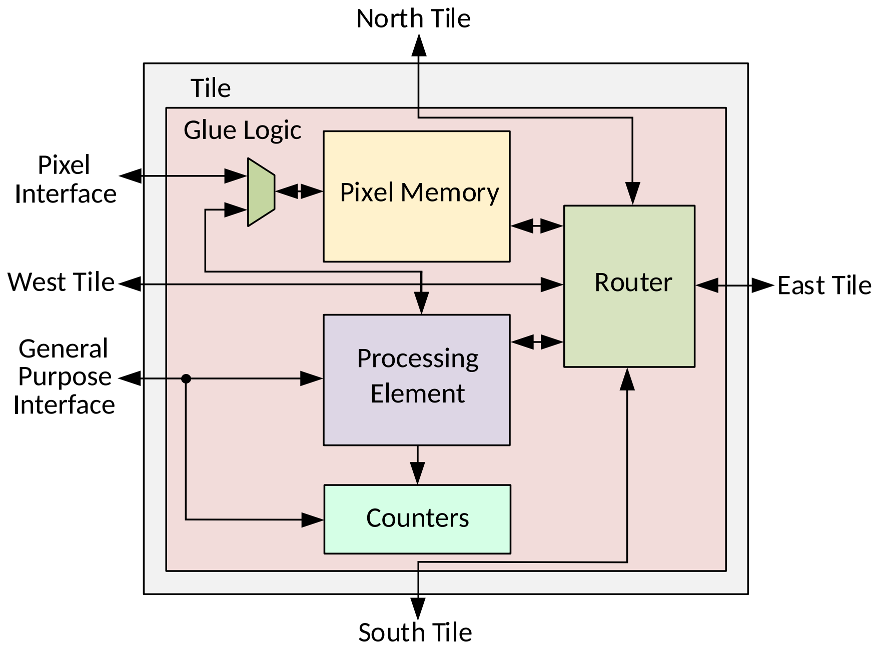

Figure 4.1: A single Tile's architecture.

In this work's architecture, each pixel can be addressed by a *5-tuple* with the following parameters:

- **value**: the pixel intensity (one or more words, considering grayscale or colored images);
- **x coordinate**: the pixel horizontal coordinate;
- **y coordinate**: the pixel vertical coordinate;
- **steps**: the pixel step index;
- **frames**: the pixel frame index.

The architecture uses these *pixel coordinates* to know which pixel is transferred, also shown in Section 3.1. The implementation details were refined to take advantage of FPGA building blocks (DSPs, Block Memories, Logic Blocks), creating an efficient overlay described in the next topics.

4.1 PIXEL MEMORY

The PM has to interface with the Router, the PE, and also interact with external components like the ARM processor or the input/output image buffers. The PMs were mapped to the BRAM blocks of the FPGA with a wrapper, to interface with the other components, as well as to deal with address computation from the *5-tuple*. Considering that UltraScale+ Block Memories limit the maximum memory interfaces to two (four data ports in total, two for read and write), this work organizes the PMs ports accordingly to Figure 4.2 using built-in BRAMs. The application of this kind of memory optimizes FPGA resource usage, improves time performance, and chip area efficiency.

There are multiple ways to access the PM according to Figure 4.1. The PE and Router need to get its pixel data, to process the algorithm and deliver pixels to other Tiles, respectively. Then, the Tile organization could connect the PM with PE and Router in three different ways:

1. The PM just connected to the PE. In this case, the Router needs to interrupt the PE to access pixel data. Its not a good choice because the processor would waste time with interruptions, and would need a complex interrupt system;
2. The PM just connected to the Router. This is a simpler solution because the PE would not need any interruption. But the Router centralizes pixel transmission in this approach, leading to network and PE data access delays;
3. The PM is connected to the PE and the Router. This is the faster approach, because the PE and Router have direct data access, without the need of any interruption, and not adding unnecessary delays, by the cost of designing a second interface.

Option 3 is the most interesting as the hardware already support a second memory interface, it does not increase resource usage significantly, and presents better performance. We chose it because of these reasons, and decided to validate our desing choice with option 2, the other possible alternative. Comparison results are shown in Subsection 6.2, but the dual-interface approach increases almost two times the manycore performance (for the same manycore size) with no significant rise in resource usage. Reserving one interface to the router guarantees better system's availability because the Network has considerable overhead compared with internal PE-PM communication (read Section 6.3.4 for details). The router also has an exclusive interface because it communicates with four neighbor tiles and the PE simultaneously and needs a faster connection with the PM to increase time performance. This way, the router does not need to compete for data with the PE.

Besides, shared PI and PE interface does not interfere in the processing time. The ARM program is aware of the manycore working cycle and does not transfer pixels while executing a task. The multiplexer logic and PE design gives priority to the Pixel Interface and prevents pixel losses in both inputs. However, the system provides support for run-time intervention if necessary, but the designer has to be aware that the PE does not have a halt functionality. So, writing to the PM through Pixel Interface while processing a task may cause undefined behavior.

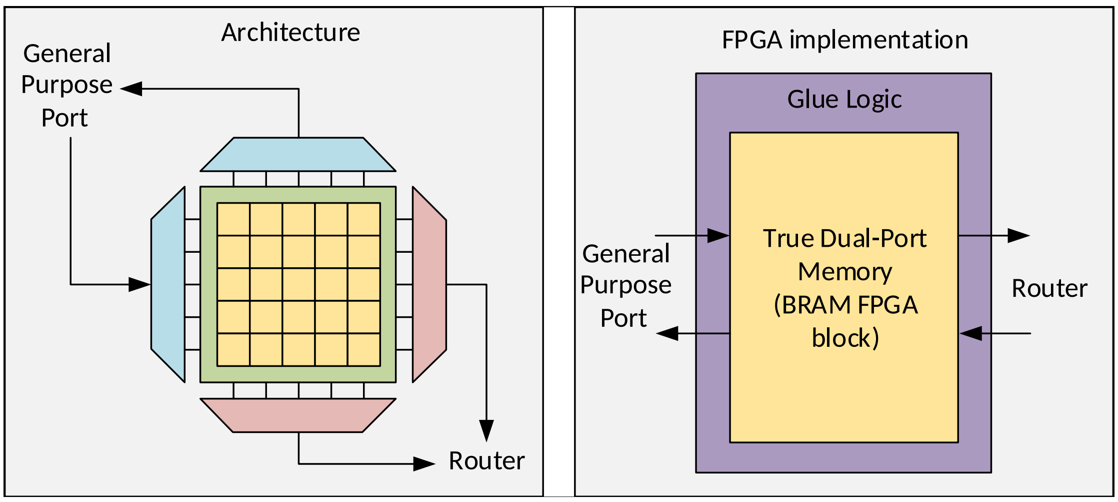

Figure 4.2: The Pixel Memory architecture.

The implementation uses VHDL code to generate true dual-port block RAMs with two write ports in read-first mode (49). The read/write operations are based on the *5-tuple*, with an address conversion to assure efficient memory utilization. First equation bellow converts the 5-tuple to a BRAM address, and the second one defines the Pixel Memory size, in number of addresses, as a dependence of some pixel parameters (21),

$$\begin{aligned} addr(x, y, n\_steps, n\_frames) =& subimg\_height * n\_steps * n\_frames * (x - x\_init) \\ & + n\_steps * n\_frames * (y - y\_init) \\ & + n\_frames * s + f, \end{aligned} \tag{4.1}$$

$$\begin{aligned} tile\_mem\_size =& n\_steps * n\_frames * subimg\_width \\ & * subimg\_height, \end{aligned} \tag{4.2}$$

where:

- $subimg\_width$: the total number of rows in tile image;
- $subimg\_height$: the total number of columns in tile image;
- (x_init, y_init): tile's initial image coordinates;
- n_frames: the total number of frames in the IP/CV algorithm;
- n_steps: the total number of steps in the IP/CV algorithm.

Address number is unique for a tile sub-image, and the region of the image stored in a tile depends on its place in the network (refer to Fig. 1.5).

Each Pixel Memory ideally stores a defined image region. In practice, there are some exceptions in border tiles, where it can have more storage than pixels in its sub-image. This happens because pixels can not be equally divided in regions depending on the manycore size. Besides, our automation tool has limitations and does not opitmize image slice for multiple architectures, which is not possible depending on the image resolution and MCVP size. But for simplicity, these exceeding addresses are not taken into account in the implemented algorithms. Besides, Vivado's block memory code generator cannot cut the exact BRAM slices depending on the architecture and image configurations due to its physical limitations, but it does not affect the implementations.

4.2 PROCESSING ELEMENT

Processing Elements must be able to perform simple computations like basic arithmetic and branch operations. Besides, they must be compliant with the application-specific needs of IP/CV algorithms. With that in mind, a minimalist RISC processor, with only 16 instructions, was designed. The PEs have access to the Register File and to the PM to use in tasks. Instruction Memories store programming code for execution.

The PE was developed focusing on: basic arithmetic operations necessary to most image processing algorithms, defined here as *R-type instructions*; branch and jump operations to control flow, named as *Branch* and *Jump instructions*, respectively; pixel data communication (*P-type*) and *Control instructions*. Those five constitute all *instruction formats* necessary for the PE to implement any type of IP/CV algorithm, turning the PE in a Turing complete machine.

For a given Register Size (RS), Figure 4.3 depicts the generalized version of the instruction formats. Fig. 4.4 is a subset of that image for RS equals to 64. Hence, Instruction size depends on RS and has the length in bits given by the following equation,

$$instruction\_size = \begin{cases} 32, & \text{if } RS < 6 \\ 5RS + 3, & \text{if } RS \geq 6, \end{cases} \tag{4.3}$$

for $RS \in \mathbb{N}$. For P-type instructions, the field at left before x pixel coordinate receives zeros for $RS < 6$ and does not exist for $RS \geq 6$. This happens because in the second case, the Pixel instruction delimits the size of all formats, and was chosen because the address field need a minimum length to work safely. By default, the address has 17 bits to ensure enough addressing in the pixel memory for multiple applications.

Figure 4.4 shows all instruction formats for a register file with 64 addresses used in our work. Note that pixel operations have to access five registers simultaneously, requiring 34-bits instruction length. *Branch* and *jump* instructions use 17-bits address field to guarantee enough Instruction Memory addressing. The PE does not have hardware for division and uses the shift-right operation as an approximate method instead, with a previous multiplication to ensure the correctness of the operation. Hence, instead of a hardware division, it performs a right-shift followed by

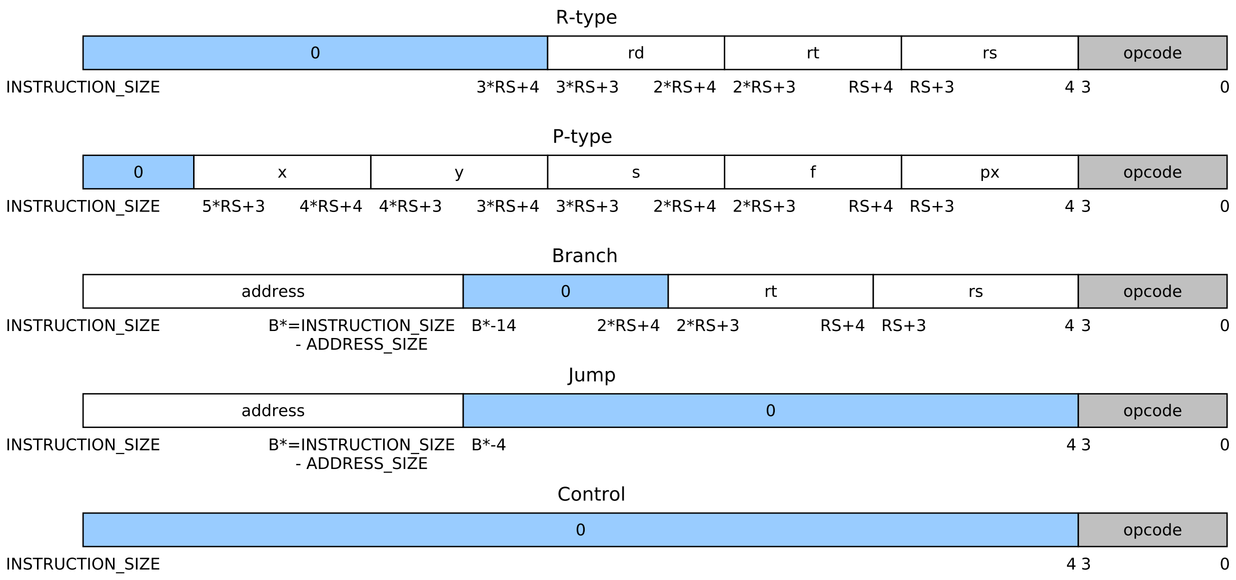

Figure 4.3: Processing Element generalized instruction formats.

a software multiplication.

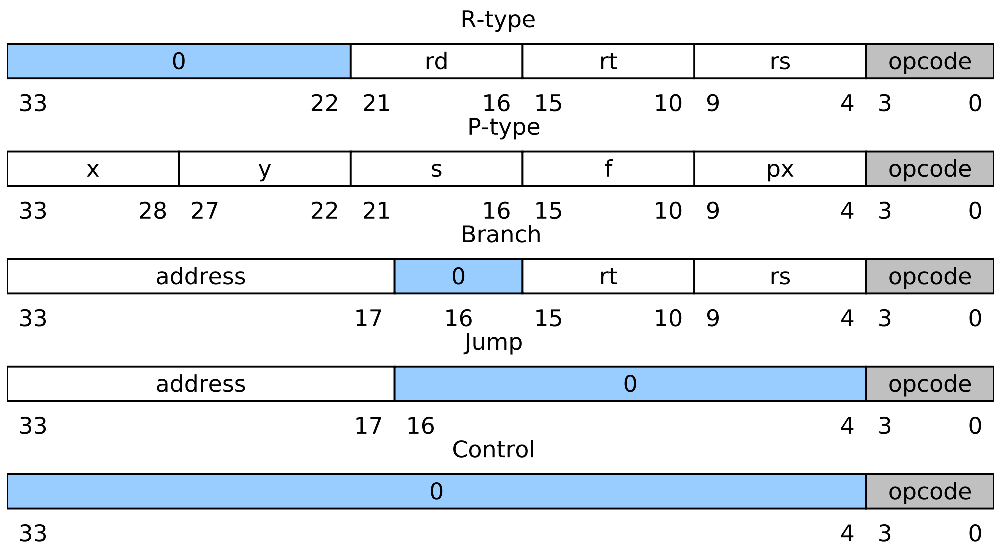

Figure 4.4: Processing Element instruction formats for a 64 addresses Register File.

With instruction formats explained, Table 4.1 shows the 16 operations implemented in our work divided by type. There are two Control, two Pixel types, eight arithmetic types, three branch, and one jump instructions. They can handle most IP/CV applications without hardware-level optimizations, but the purpose is to have a simple instruction set to show the scalability of our MCVP architecture.

On the other hand, the use of these limited sets combined with an assembly language allows complete control of the operations and fine-grained optimizations. This is not an advantage compared with modern compilers, but combined with our specialized HW enables much better performance, as shown in Chapter 6. Below is the explanation about operands necessary for each

Table 4.1: Instruction Set used in the Processing Element

Instruction	Type	Description
NOP	Control	Do nothing
GPX	P-type	Get a pixel value from address
SPX	P-type	Set a pixel value to address
ADD	R-type	Arithmetic addition of two integers
SUB	R-type	Arithmetic subtraction of two integers
MUL	R-type	Arithmetic multiplication of two integers
DIV *	R-type	Arithmetic division of two integers
AND1	R-type	Bitwise AND operation of two binary
OR1	R-type	Bitwise OR operation of two binary
XOR1	R-type	Bitwise XOR operation of two binary
NOT1	R-type	Bitwise NOT operation of two binary
BGT	Branch	Branch if bigger than
BST	Branch	Branch if less than
BEQ	Branch	Branch if equal to
JMP	Jump	Jump to
ENDPGR	Control	End Program

* in this version division was substituted by a shift-right hardware.

instruction and its result, which was inspired by the DLX instruction set (53), following the fields in Figure 4.3. The explanation has similar sequence as the RISC-V instruction list in (54), which serves as a reference for this part.

- **NOP** (Control Type): *No Operation.* Advances the PE's Program Counter by one instruction, consumes one clock cycle to perform it.
- **GPX** x, y, s, f, px (Pixel Type): *Get Pixel.* R[px] = sext(M[addr(R[x], R[y], R[s], R[f])]), where sext is the sign-extension, R the register file array and M the memory array.

  Stores a pixel in the register given by px and based on pixel parameters x, y, step, and frame stored in each register. Equation 4.1 transforms these parameters into effective Pixel Memory's address. The PE can read pixels from the local PM, using the PE-PM interface, or use the router to communicate with other image regions.
- **SPX** x, y, s, f, px (Pixel Type): *Set Pixel.* M[addr(R[x], R[y], R[s], R[f])] = R[px].

  Stores a pixel value in a Pixel Memory address given by addr function (Equation 4.1). The PE can write pixels in the local PM, using the PE-PM interface, or use the router to access other image regions.
- **ADD** rs, rt, rd (Arithmetic Type): *Add.* R[rd] = R[rs] + R[rt].

  Add register R[rs] to R[rt] and stores the value in register R[rd].
- **SUB** rs, rt, rd (Arithmetic Type): *Subtract.* R[rd] = R[rs] - R[rt].

  Subtract register R[rt] from R[rs] and stores the result in register R[rd].

- **MUL** rs, rt, rd (Arithmetic Type): *Multiply*. R[rd] = R[rs] $\times$ R[rt].

  Multiply register R[rs] by R[rt] and store the result in register R[rd].

- **DIV** rs, rt, rd (Arithmetic Type): *Divide*. R[rd] = R[rs] $>>_S$ R[rt].

  Divides R[rs] by multiples of two, applying a shift right arithmetic operation. Shifts R[rs] value to the right by R[rt] positions and stores the result in R[rd]. This operation copies R[rs]'s MSB to the empty bits after shift. This means that the divide operation performs a shift right one in the actual instruction set because it is easier to implement and scale in the FPGA. But theoretically, this instruction should perform a hardware division.

- **AND1** rs, rt, rd (Arithmetic Type): *AND*. R[rd] = R[rs] & R[rt].

  Calculates the bitwise AND operation between R[rs] and R[rt], stores the result in R[rd].

- **OR1** rs, rt, rd (Arithmetic Type): *OR*. R[rd] = R[rs] $|$ R[rt].

  Performs the bitwise OR operation between R[rs] and R[rt], stores the result in register R[rd].

- **XOR1** rs, rt, rd (Arithmetic Type): *Exclusive-OR*. R[rd] = R[rs] $\wedge$ R[rt].

  Calculates the bitwise Exclusive-OR operation between R[rs] and R[rt], stores the result in R[rd].

- **NOT1** rs, rd (Arithmetic Type): *NOT*. R[rd] = $\sim$ R[rs].

  Writes the compliment of R[rs] in register R[rd].

- **BGT** rs, rt, address (Branch): *Branch if Greater Than*. if (R[rs] ¿ R[rt]) pc = address.

  Program counter receives the address field value, which has to be multiple of 4 if register R[rs] is greater than R[rt].

- **BST** rs, rt, address (Branch): *Branch if Smaller Than*. if (R[rs] ¡ R[rt]) pc = address.

  Program counter receives the address field value, which has to be multiple of 4 if register R[rs] is smaller than R[rt].

- **BEQ** rs, rt, address (Branch): *Branch if Equal*. if (R[rs] == R[rt]) pc = address.

  Program counter receives the address field value, which has to be multiple of 4 if register R[rs] is equal to R[rt].

- **JMP** address (Jump): *Jump*. pc = address.

  Jump to address in the next clock cycle.

- **ENDPGR** (Control): *End Program*. ndn = '1'.

  Sets internal ndn signal to '1', gets '0' with the MCVP IP reset. It triggers the NDN flag in Status and Control Register to '1' for a given tile addressed in SCR's MODE signal if the Many-core is finished to write the final image in the Visualization Frame Buffer.

Table 4.2: Register File organization.

31 0	
R0/a0	Zero hardwired
R1/a1	One hardwired
R2/a2	Function argument, return value
R3/a3	Function argument, return value
...	Function argument, return value
R(RFS-9)/a(RFS-9)	Function argument, return value
R(RFS-8)/a(RFS-8)	Function argument, return value
R(RFS-7)/xi	Tile's x initial coordinate
R(RFS-6)/yi	Tile's y initial coordinate
R(RFS-5)/imw	Image width
R(RFS-4)/imh	Image height
R(RFS-3)/smw	Sub-image width
R(RFS-2)/smh	Sub-image height
R(RFS-1)/ssr	Step Sync Register

PE's Register File (RF) has two hardwired addresses to store 0 and 1 integer values for arithmetic operations. Six Special addresses store sub-image coordinates corresponding to the initial x and y Tile parameters, sub-image length, and total image size for each axis. Also, a Special Register address called Step Sync Register (SSR) saves the last s pixel coordinate to ensure that the Pixel Memory correctly reads and sends data to other tiles already processed from its PE.

It is important to emphasize that these six special registers are defined in design time and statically (hard-wired) implemented during logic synthesis and implementation steps. Run-time modifications acceptance is a possible improvement. Register File Size (RFS) depends on the Register Size value, and most of the addresses stores data for general purpose use, from R2 to R(RFS-8). Table 4.2 shows the Register File organization in terms of RFS, which is given by two to the power of RS ($RFS = 2^{RS}$).

The PE has a single-cycle microarchitecture, including its main datapath for R and P-type instructions. The Instruction Memory is implemented as a modified single-port RAM with read-first configuration (49). It can be configured as Block or Ultra RAM (the two types of distributed memory blocks in the SoC used) depending on design constraints and programmed through an External Interface by the ARM processor via the AXI bus.

The Register File stores pixel data from the Router or Pixel Memory transferences, while the Glue Logic selects the correct output interface depending on the desired pixel location. The PE interrupts clock cycles until the message arrival every time the RF requests pixel data. Figure 4.5 depicts a simple Processing Element diagram, showing the P-type and R-type instructions microarchitecture.

Since there is no compiler for the Processing Elements yet, all application algorithms were implemented directly in assembly language, as explained in Chapter 5.

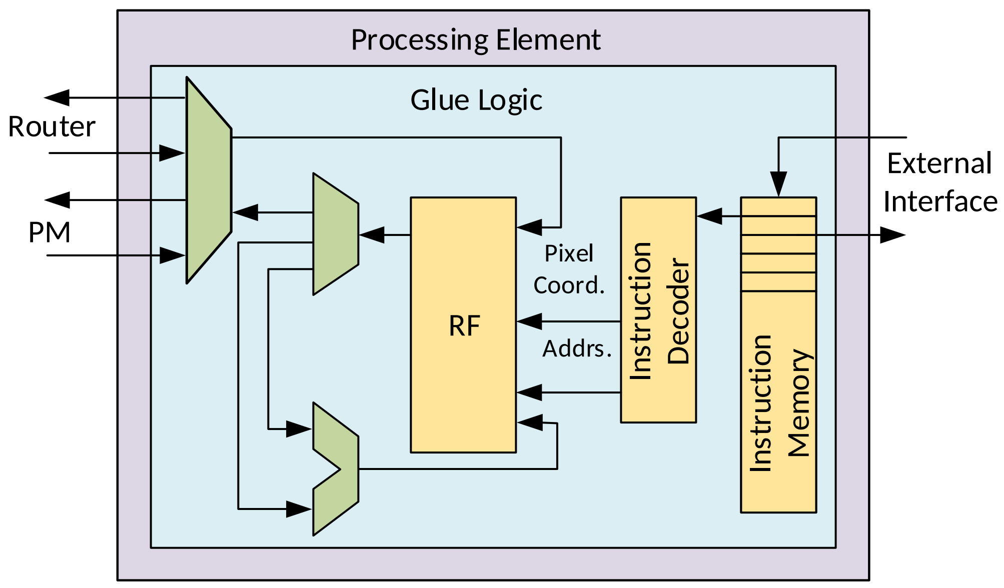

Figure 4.5: Processing Element microarchitecture for P-type and R-type instructions.

4.3 ROUTER

The Router is the component responsible for exchanging pixels among the tiles. Its interface consists of 6 ports, each one with one input and one output channel. All the channels are unidirectional connected to the neighboring tiles (N, S, E, W) and the local PE and PM. Between each tile pair, there are message buffers used to avoid network stalls. The size of these buffers is configurable and depends on the algorithms used and the overall architecture configuration (image resolution and number of tiles). Figure 4.6 shows router internal architecture, with its internal buffers, interfaces and control unit.

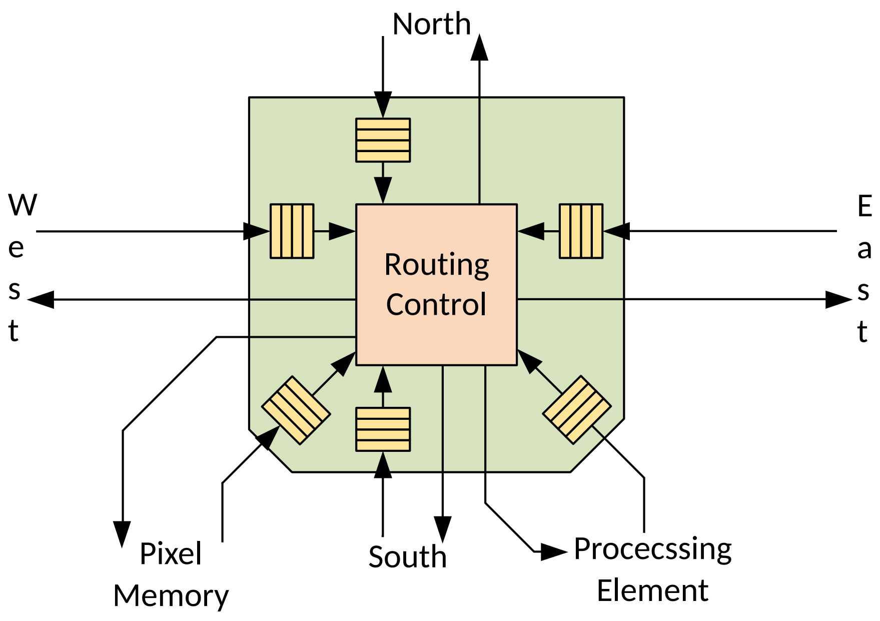

Figure 4.6: Router internal structure.

Despite the step sync messages deployed by the PE for controlling purposes, only a single kind of message can be traversed through the network. It contains the 5-tuple previously described, the origin coordinates, and a flag to indicate if a message is a request for a pixel or the answer for a request (forward/backward messages). Figure 4.7 illustrates a message and how it is routed through the manycore.

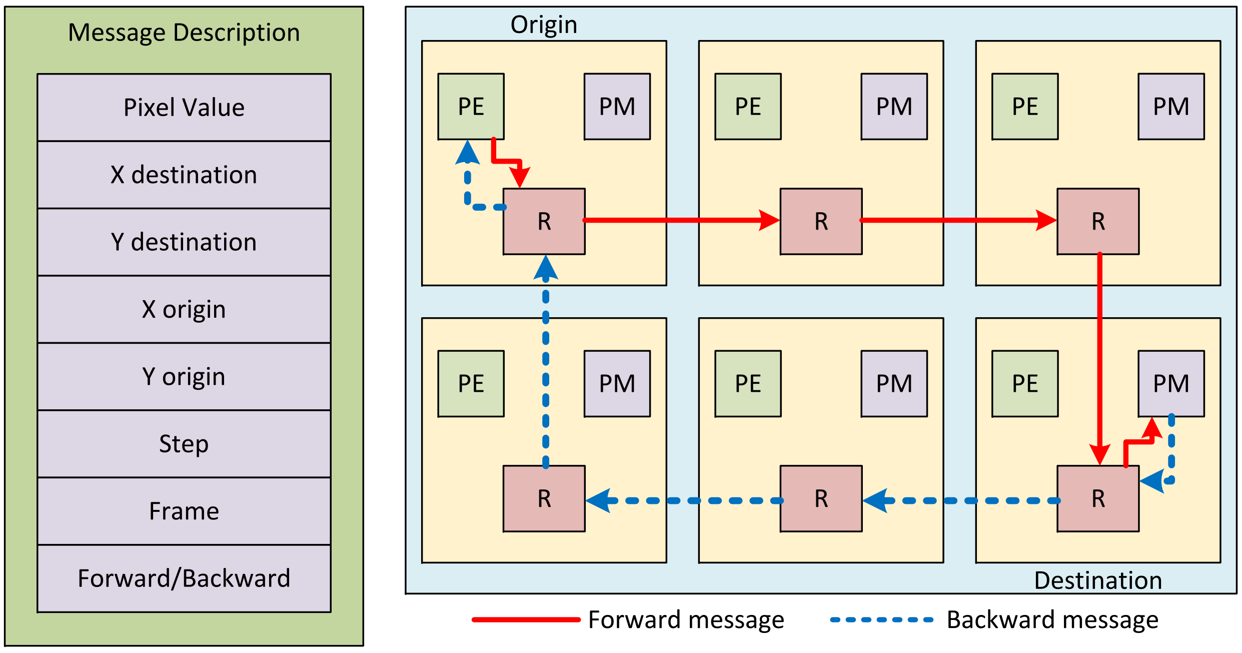

Figure 4.7: Message description and route.

For example, if a PE has to perform a Get Pixel instruction (GPX), which means that it needs a specific pixel at the RF to process. The PE wrapper verifies if the pixel belongs to its image region or another one. If it is a local pixel, the wrapper asks the local PM for the pixel value. In the case that a pixel is in another image region, the wrapper asks the local router. The router has an arbiter and, when it is the PE communication slot, the router decodes the destination and forwards the message to a neighbor router. Take the Figure 4.7 as a sample, the top left tile requests a pixels that belongs to the bottom right one. Because of this distribution, the forward massage passes through all routers in the red arrow route until it gets to its destination. After getting the desired pixel, the PM sends it to the nearest router, which passes the message (as shown in dashed blue arrow) to neighbor routers until the pixel arrives to its origin.

The router consists of three logic units that perform different tasks along with the message flow. The router has 6 input connections and 6 output connections. These connections are referred to as links. Considering the 2D-mesh topology, the links tie the router within the tile (PM and PE) and outside the tile (North, South, East, West). Figure 4.8 illustrates the router architecture implemented in this work.

The message path initiates by storing the entire message at once in the Buffer (FIFO), which has this name because of its changeable size in design time. At the end of the FIFO's queue, there is a unit named Decoder, that is responsible for deploying the routing algorithm. The Decoder will pull the necessary field to interpret the proper message direction, that is, origin coordinates, the destination coordinates, and the tail field that informs whether the message is going back (to the origin) or if the message is going forward (to the destination) and then the decoder issues a request to the Flow Control Unit (FCU) for assigning the message to its respective channel.

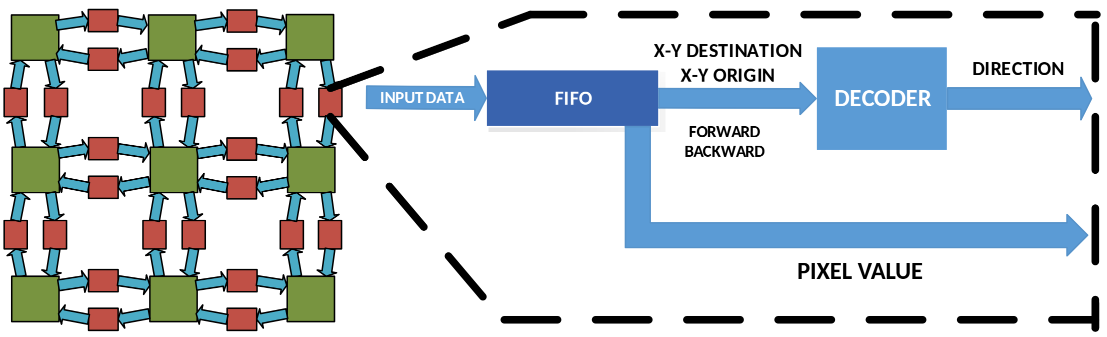

Figure 4.8: Router architecture.

A standard XY routing algorithm determines which output channel shall receive the message. It basically calculates the number of hops[1] to achieve the destination in each direction (XY), and then it allocates to the Proper output channel. There are some situations, in which the decoder will assign the message direction in observance of network congestion, given the history of the previous message assigned in the same situations. The combination of Buffer and Decoder is referred to as Input Channel.

```
Result: Message
if Output channel is idle then
    Analyse all messages;
    if Tied waiting time then
        Arbitrates to all messages that go out of the network;
    else
        Older messages;
    end
end
```

Algorithm 1: Injection controller arbiter.

The FCU is responsible to multiplex the input channels and assign to output channels. This is done by submitting the “ready” input channels of which their respective output direction has an output channel in idle state, to an arbiter. The arbiter priority is primary the longest message waiting, in case of a tie, there are 2 options implemented: prioritize message from PE and PM input channel or prioritize the messages from the neighboring tiles. Algorithm 1 is the algorithm adopted by the arbiter.

In the first case, prioritizing the PM and PE input channels will work as a form to expand the network load. Since the PEs are not able to issue more than one message at a time and waits in idle

[1]Number of links between origin and destination routers.

state until a pixel arrival for GPX instructions, there is a limitation in network load. Prioritizing in this case, the message injection will not have a significant impact and it will gather the messages inside the tile that can possibly lead to deadlocks. The second case was adopted for our work. It presents a better network load distribution.

5 AUTOMATION TOOL

This chapter explains the software responsible for the manycore architecture generation and other useful Vivado Project setups, called the Many-Core Vision Processor Architecture Automation Tool (MCVP-AT). The tool builds many-core vision processors for multiple 2-D mesh grid NoC topology and the binary to run in its Processing Elements. Besides, the software design and documentation were made thinking of future upgrades like assembly time optimizations, use of pseudo-instructions, memory aware mapping, and OpenVX (55) standard implementation.

It is written in ANSI C (also known as C89) to increase its compatibility and maintainability, as most embedded systems engineers work with C/C++. A previous assembler version written in python was discontinued due to software maintaining difficulties within the research members.

Figure 5.1 depicts a diagram of the Automation Tool organization. The program takes a JSON string file path as its single argument. It checks for the MCVP git project repository, downloads it if needed, assemble PE's binaries and build the many-core auxiliary files. These files are integrated with the MCVP Vivado project automatically, where the user needs to re-generate the bitstream (in the case of a new architecture) or compile and upload the ARM firmware in the ZCU102 board.

Next sections explain the application blocks in details.

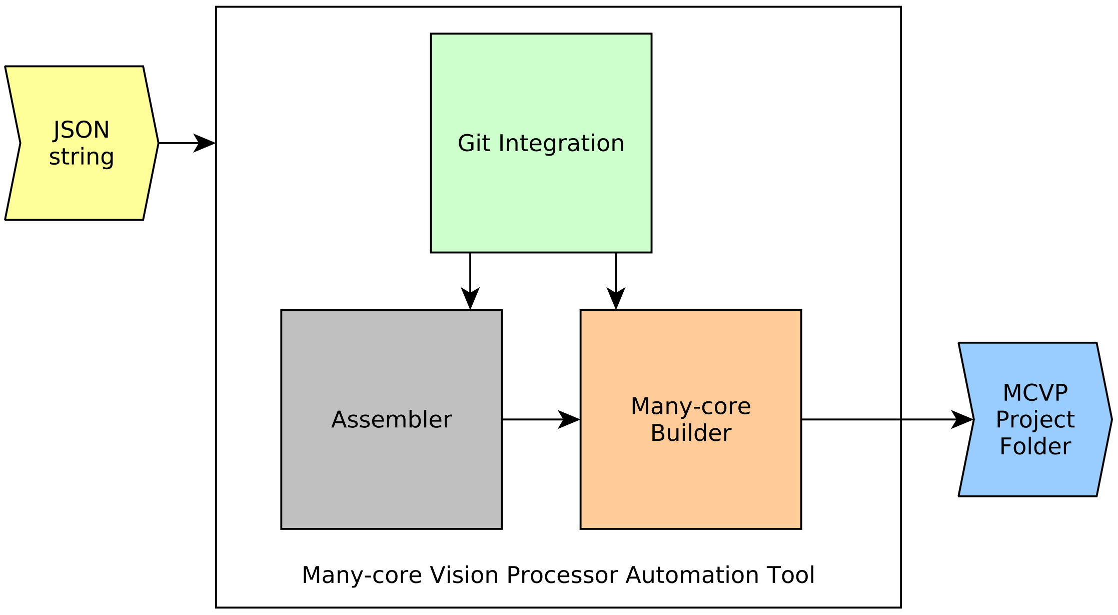

Figure 5.1: Automation Tool Diagram.

5.1 JSON STRING

JSON (JavaScript Object Notation) is a text syntax that eases integration between all programming languages. It provides a simple notation of expressing collections, a set of *key* and *values*, similar to a C `struct` (56). It is used as an input method for the automation tool in this work. Also, as a simple way of setting the program up without the need to write shell scripts or other wrapper tools to set parameters. A JSON simplifies future integration with other programs, with no need to modify the automation tool source code to use it with other applications. Below is an example of the JSON string to run the MCVP-AT which all fields are mandatory to run the program.

```
{
    "manycore":
    {
        "imageWidth"             : 256,
        "imageHeight"            : 256,
        "pixelBitLength"         : 9,
        "manycoreWidth"          : 40,
        "manycoreHeight"         : 40,
        "maximumStep"            : 6,
        "maximumFrame"           : 1,
        "instructionMemorySize" : 1024,
        "routerBufferLength"     : 2,
        "registerFileLength"     : 64,
        "ultraRamLimit"          : 96.0,
        "addressBitLength"       : 17,
        "registerFileBitSize"    : 5
    },
    "vivadoProject":
    {
        "gitRepoUrl"             : "https://gitlab.com/arctichopper/many-vp",
        "gitFolderPath"          : "/home/root/manycore_vision_processor",
        "projectName"            : "many_vp",
        "outputFolderPath"       : "/home/root/sbesc_2020/new/results/test"
    },
    "sdkApplicationProject":
    {
        "applicationName"        : "many_vp_app"
    },
    "assemblyCode":
    {
        "assemblyPath"           : "/home/root/smoothing_filter.asm",
        "binaryName"             : "smoothing_filter.bin",
        "binaryPath"             : "/home/root/fgcs_journal/smoothing_filter.bin"
    }
}
```

JSON value is a field that, as the name suggests, holds a value or information. It can be of type object, array, number, string, `true`, `false` or `null`. An object is a set of name/value pairs organized as a structure separated by curly brackets, where names are strings. A single comma delimits a value to the following name (56). `{ "one" : 1, "two" : 2 }` is an example of JSON object, where `"one"` and `"two"` are names for integers 1, 2, respectively.

It is possible to relate the MCVP-AT JSON string with these definitions. There are four main names in the string, concerning manycore settings, Vivado Project path and other information, the SDK Application Project name, and PE's Assembly Code location and binary name. Next is the definition of each field:

- `"manycore"`: name of the manycore parameters' object;
  - `"imageWidth"` (integer): the total number of image rows;
  - `"imageHeight"` (integer): the total number of image columns;
  - `"pixelBitLength"` (integer): number of bits in signals that carry pixel data;
  - `"manycoreWidth"` (integer): number of rows in the manycore mesh grid;
  - `"manycoreHeight"` (integer): number of columns in the manycore mesh grid;
  - `"maximumStep"` (integer): the total number of steps in the IP/CV algorithm;
  - `"maximumFrame"` (integer): the total number of frames in the IP/CV algorithm;
  - `"instructionMemorySize"` (integer): number of instructions that can be stored in Instruction Memory;
  - `"routerBufferLength"` (integer): router Elastic Buffer length, refer to Section 4.3 for more information;
  - `"registerFileLength"` (integer): number of addresses available in register file as explained in Section 4.2;
  - `"ultraRamLimit"` (float): maximum number of tiles using Ultra RAM in Instruction Memory;
  - `"addressBitLength"` (integer): bit length of address field in Branch and Jump instruction types (Figure 4.3);
  - `"registerFileBitSize"` (integer): number of bits in register file registers, for example, 5 means that all RF's addresses have $2^5 = 32$ bits;
- `"vivadoProject"`: name of the Vivado Project object;
  - `"getRepoUrl"` (string): the manycore vision processor git repository URL;
  - `"gitFolderPath"` (string): complete path to git repository;
  - `"projectName"` (string): Vivado Project name;
  - `"outputFolderPath"` (string): complete folder path to store Vivado simulation output;

- `"sdkApplicationProject"`: name of the Xilinx SDK application Project;
  - `"applicationName"` (string): Xilinx SDK application name that holds the firmware to run in ZCU's ARM;
- `"assemblyCode"`: Assembly code object name that refer to input/output files or paths to run the PEs assembler;
  - `"assemblyPath"` (string): complete path to the assembly source file;
  - `"binaryName"` (string): name of the output binary file;
  - `"binaryPath"` (string): complete path to the output binary string file.

This work uses an open-source JSON Parser written in ANSI C called cJSON (available at ⟨https://github.com/DaveGamble/cJSON⟩) to reduce efforts of interpreting the JSON file. The software claims to be ultra-lightweight, and indeed, has good performance and usability for a desktop application. Besides, cJSON is licensed under MIT, a short and permissive license that eases its integration with other software.

5.2 GIT INTEGRATION

The git integration module is a simple shell command verification to check if the Many-core repository is already downloaded in the current file system. If not, it downloads the data from ⟨https://gitlab.com/arctichopper/many-vp"⟩ using the following command,

```
git clone --recursive "getRepoUrl" "gitFolderPath".
```

5.3 ASSEMBLER

The assembler is related to the instruction formats from Section 4.2 and was designed for this works' ASIP architecture. It takes an Assembly source file and returns a binary string with the instructions, ready to be stored in Processing Elements. A custom assembly language was created to accomplish this task, implementing the same instructions of Table 4.1. For future improvements, new kinds o pseudo-instructions could also be implemented for code size reduction.

Assembly language follows the previously mentioned instructions rules and some additional functions to simplify coding, readability, and maintainability. First, single-line comments can be used with the character `;` like other assembly languages, it only accepts single line and does not have C-like comments (`//` or `/* */`).

Also, there is the concept of labels, any string at the beginning of a line terminated with a colon (`:`). These labels can be used to substitute instruction address field and drastically eases code writing. Automation tools automatically perform calculations to give proper label addressing during text parse and processing. Depending on the instruction and operands, the Processing Element's Program Counter branches to the first instruction after a given label.

The assembler has a feature to recognize special and regular arguments registers listed in Table 4.2. It is not case sensitive and accepts "_" to separate word in labels. Next is a sample assembly code chunk of a Non-maximum suppression algorithm implemented in this work.

```
1       add a0, a25, ssr     ; step sync register = 5
2
3       ; a55 and a56 will be used as tile delimiters
4       ; a55 - x delimiter
5       ; a56 - y delimiter
6       ; a53 and a54 will be used as tile initializers
7       ; a53 - x init
8       ; a54 - y init
9
10      add xi, smw, a5     ; a5 = xi + smw
11      sub a5, a0, a55
12      bst a5, imw, y_limiter_non_max
13      add imw, a0, a5
14      sub a5, a6, a55    ; subtract to assign x limit
15
16  y_limiter_non_max:
17      add yi, smh, a5     ; a5 = yi + smh
18      sub a5, a0, a56
19      bst a5, imh, x_init_non_max
20      add imh, a0, a5
21      sub a5, a6, a56    ; subtract to assign y limit
22
23  x_init_non_max:
24      add xi, a2, a53
25      beq xi, a0, y_init_non_max
26      sub xi, a0, a53
```

There is a use of the Step Sync Register (`ssr`) in the first line, which receives the value stored in Register File address 25 (`a25`). Besides, line 16 presents a label separated by underlines and finished by a comma. There is a branch type instruction in line 19 with the address field completed by a label from line 23. It means that if the value of `a5` is less than the value of Image Height Register (`imh`), the next instruction to be fetched is an addition from line 24. In this example, there are also many line comments to explain code behavior and improve readability.

5.4 MANY-CORE BUILDER

This part integrates directly with the git repository and overwrites four files. The first file, named ”manycore.vhd”, is responsible for setting up the manycore size in a 2-D mesh grid NoC architecture. The second one, named ”manycore_axi_wrapper.vhd”, is a VHDL wrapper responsible for instantiating the manycore, a memory, and AXI4-Lite interfaces. The third, named ”manycore_tb”, is a testbench that takes an assembly binary file, reads it, and stores multiple output images in the `"outputFolderPath"`. The last, named ”noc.h”, is a C header file containing architectural definitions for the firmware, like instruction properties, NoC, and image sizes.

For every execution of the MCVP Automation Tool, these files are updated depending on JSON string parameters. After that, the designer can directly update Vivado or SDK projects and test new features or applications.

5.5 MCVP PROJECT FOLDER

The project folder has a common Vivado project organization. There are folders for the SDK that stores the application firmware. Also, folders for hardware design sources and constraints. There is an IP repository folder in the root path with all custom IPs used in the design. Project organization allows complete reproduction of the results in different machines for the same Vivado 2019.1 suite version. The automation tool modifies files from `many_vp.sdk/many_vp_app` and `many_vp.srcs/sources_1` folders. Figure 5.2 shows the git repository source tree.

```
.
├── ip_repo
│   ├── CMOS_BRAM_1.0
│   ├── Image_Visualization_1.0
│   └── Visualization_Frame_Buffer_1.0
├── logs
└── many_vp
    ├── many_vp.sdk
    │   ├── hello_world
    │   └── many_vp_app
    └── many_vp.srcs
        ├── constrs_1
        └── sources_1
```

Figure 5.2: MCVP project source tree.

6 RESULTS AND ANALYSIS

This chapter describes the most significant results obtained using the Manycore Vision Processor architecture proposed and discussed in this work. It follows a logical path, from applications description, investigations with Vivado Simulation and pitfalls that changed design decisions, to final results using the complete IP/CV chain with acquisition and visualization scheme. All experiments are explained and addressed to distinguish significant features and possible improvements in the architecture.

It has the following organization: first, an explanation about implemented algorithms, next there are initial results that motivated further experimental analysis, following is a performance evaluation and comparison with the Manycore Vision Processor alone, and at last, there are results of the complete architecture using the camera and the ARM processor.

6.1 APPLICATION: HARRIS CORNER DETECTOR

The architecture proposed in this work aims to implement any type of IP/CV applications. In this context, we decided to show its flexibility with the implementation of an application suitable to explore different types of IP/CV algorithms.

Corner Detectors are used in a wide range of applications: object tracking, stereo image correspondence, reference for precise geometrical measurement, and camera calibration. While edge detectors estimate high gradients in one direction, a corner detector computes gradient in multiple directions (1). There are various corner detectors, but one of the most used is the Harris Corner Detector (HCD) (57), which was selected as our demo application.

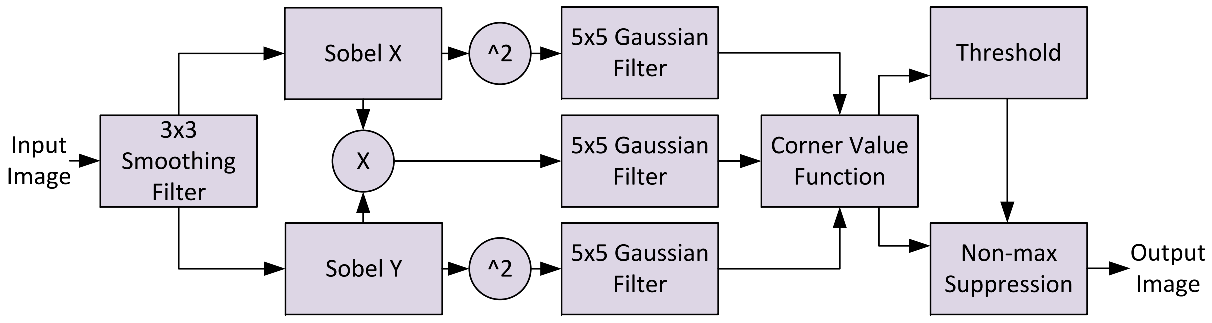

Figure 6.1: Harris Corner Detector processing chain.

Figure 6.1 shows the HCD processing chain based in common textbook implementations (1). The smoothing filter first reduces edge intensities throught a mean filter that passes low frequencies, it is also known as a box filter, given by the following equation,

$$H = \frac{1}{9}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}. \tag{6.1}$$

Next step is a Sobel X (I_x) and Y (I_y) estimation, responsible for calculate image gradients for x and y directions, respectively. It detects image edges according to these two convolutions,

$$I_x = I * H_x^P = I * \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \tag{6.2}$$

$$I_y = I * H_y^P = I * \frac{1}{8}\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}, \tag{6.3}$$

where the division by 8 normalizes gradient values from -255 to 255, which is pixel value limits acceptable in our implemented architectures. Then, with two gradient intermediary images, the algorithm computes the *structure matrix* M, given by the following equation,

$$M = \begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix} = \begin{pmatrix} A & C \\ C & B \end{pmatrix}. \tag{6.4}$$

Next, images A, B, and C are filtered by convolution with a 5x5 Gaussian filter given by the following mask,

$$H^G = \frac{1}{57}\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 1 & 3 & 5 & 3 & 1 \\ 2 & 5 & 9 & 5 & 2 \\ 1 & 3 & 5 & 3 & 1 \\ 0 & 1 & 2 & 1 & 0 \end{pmatrix}. \tag{6.5}$$

HCD defines the function $V = V(x, y)$,

$$V = AB - C^2 - \alpha\,(A + B)^2\,, \tag{6.6}$$

for each image pixel, called *corner value function*, where $\alpha \in [0.00, 1.00]$ is the detector sensitivity. With the image V computed, a threshold ($t_H \in [900, 10'000]$) is applied, generating a binary image with values to pixels that are higher than t_H. Finally, a non-max suppression routine removes non-maximum pixels in a 5x5 neighbourhood. The result is a binary image with only the detected corners.

The HCD algorithm is composed of several smaller blocks, most of them performing a convolution. To have different comparisons to the literature, we evaluated the performance of the complete HCD, the convolution with sizes 3x3, 5x5, 7x7, and the Sobel Edge Detector. The last one uses the edge stregth orientation equation,

$$E(u,v) = |I_x(u,v)| + |I_y(u,v)|\,, \tag{6.7}$$

to calculate, store and compute Sobel detector time.

6.2 INITIAL RESULTS

This section describes motivations for the MCVP architecture and the initial results that led to its final version. Four different architectures are implemented with a different number of Processing Elements: 1, 4, 9, and 16; in a 1x1, 2x2, 3x3, and 4x4 2-D mesh grid NoC, respectively. Different from Figure 4.1, there is no direct connection between PM and PE, where the only possible link is through a Router. Figure 6.2 shows the initial Tile implementation without PE-PM interface. The PE only has access to pixels through the Router.

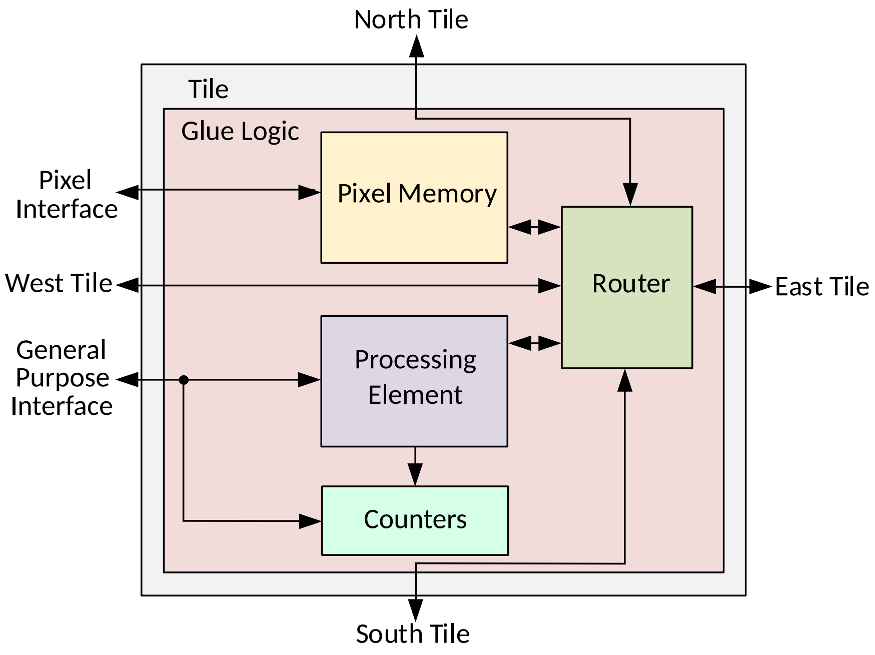

Figure 6.2: A initial Tile architecture without PE-PM interface. The PE only has access to pixels through the Router.

One core of the ARM Cortex processor is used as a standard embedded processor for performance comparisons with the previous four designs. It is targeted for real-time applications and was configured to run at $500MHz$. The processor stores the initial and final image data in FPGA's Block RAMs to emulate a processing chain with an external acquisition and displaying scheme. Figure 6.3 (left) depicts the Harris detector cycles per pixel results for these systems, considering a 256x256 image and 9 bits pixels. The ARM processor takes more than 9 thousand cycles to process a single pixel, most because of its non optimized memory access to the BRAMs (that

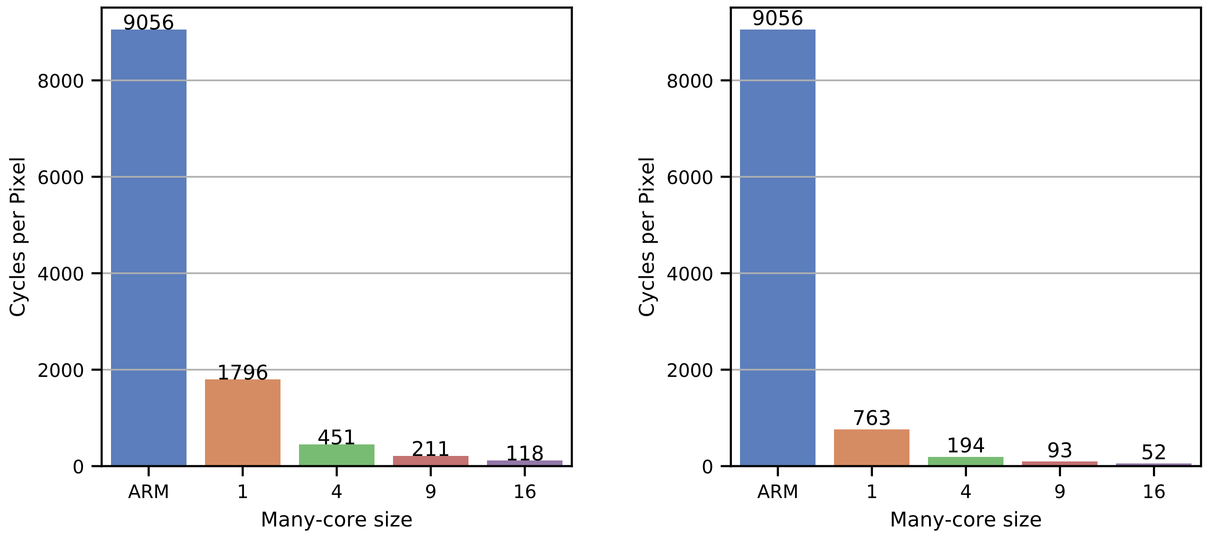

Figure 6.3: Comparison between only PM-Router only interface (left) with PE-PM/PM-Router interfaces (right) for the HCD at $100MHz$.

works at $100\ MHz$), also due to its lack of SIMD instructions or similar hardware optimizations (51). ARM results takes more than five times compared with other manycores, which decreases linearly with the architecture size. Figure 6.3 (right) shows the HCD cycle per pixel data for the MCVP final version, linking PM with PE directly. Again, the number of cycles decreases linearly with the manycore size increase, similar to previous results. But newer architectures present less than a half cycles to complete the same algorithm.

Table 6.1: Implementation report for a 256x256 image, 16 max. steps and 1 frame for previous Router only interface.

	Device	ARM	1 PE	4 PEs	9 PEs	16 PEs
Proc. Freq. (MHz)	-	500	100	100	100	100
PL Max. Freq. (MHz)	-	118.72	99.87	102.12	115.73	115.53
LUT	230400	9011	6752	22172	48840	89770
LUTRAM	101760	1870	2274	8806	19686	35019
FF	460800	10789	2298	11344	25257	62148
BRAM	312	256	256	256	261	256
DSP blocks	1728	0	4	16	36	64
On-Chip Power (W)	-	3.63	3.51	3.65	3.88	4.34

Table 6.1 has resource data for the Router only interface, while Table 6.2 with PE-PM interface for the same sizes. The newer version has considerable improvements in terms of memory blocks, including the use of Ultra RAM, which was not implemented previously. But, even with a new memory interface, it is clear that it has no significant impact on resource usage and could be suppressed with other memory optimizations. Hence, the PE-PM interface is an advantage with no significant resource increase compared with its performance gains.

Concerning scalability, one initial question to ask was if the MCVP could achieve better performance for a larger number of tiles. We made a series of Vivado simulations to evaluate system characteristics with the automation tool aid. Figure 6.4 shows time performance (in frames per second) behavioral simulation results for different core sizes with the PE-PM interface and implementing the HCD. There is a linear fps increase with manycore size, not showing signals of performance saturation. This means that the architecture can scale for at least 400 tiles in physical implementation, achieving more than 500 fps, according to Vivado Simulator. The automation tool can generate bigger sizes, and the simulator also is able to handle them. We chose to simulate a reasonable manycore size that could fit in bigger FPGAs, not intending to perform a complete analysis of parallelism limits of our architecture, but to demonstrate that it can have performance increase for bigger devices.

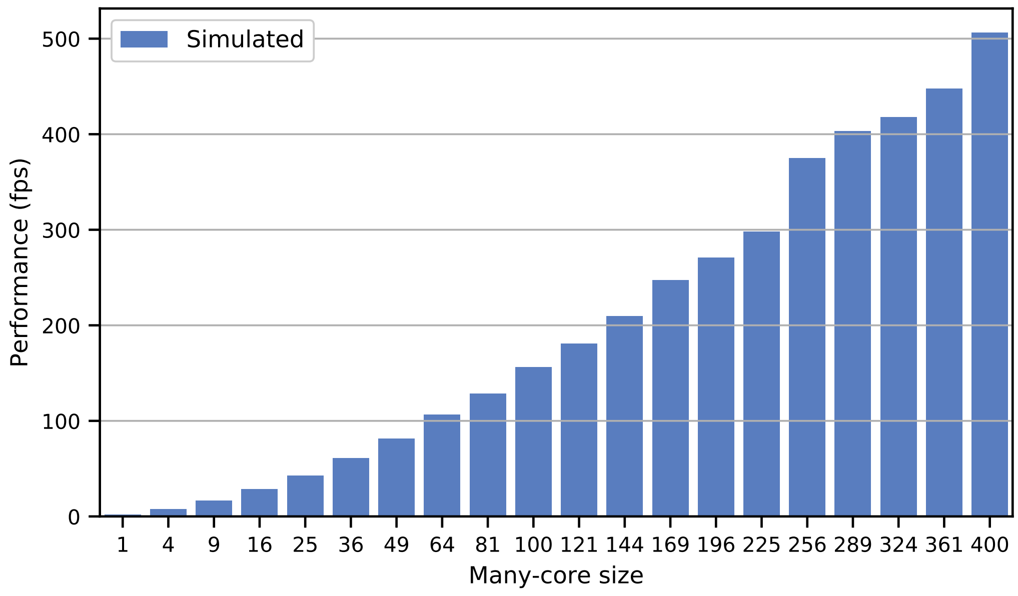

Figure 6.4: Harris Corner Detector performance in frames per second per architecture in Vivado Simulator, 100 MHz, 256x256 image.

The MCVP architecture proposed in this work has the potential to get good performance considering the results above, even compared with other architectures and implementations, refer to subsection 6.3.3 for details. Then, it is safe to continue with more analysis and experiments, to evaluate other practical applications in FPGA and focusing on the final MCVP performance. The next section depicts some meaningful results about it.

6.3 MANYCORE PERFORMANCE

All results obtained in this section use a Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. Nine different architectures are physically implemented with a different number of Processing Elements: 1x1, 2x2, 3x3, 4x4, 5x5, 6x6, 7x7, 8x8, and 9x9 2-D mesh grid manycore. The available FPGA size limited our work to 81 Processing Elements (9x9). However, we also compared implementation results with previous simulated performance data of 10x10, 11x11, 12x12, 13x13, 14x15, 15x15, 16x16, 17x17, 18x18, 19x19, and 20x20 (400 tiles) to show the speedup and scalability of our architecture.

Experiments performed in this Section consider the time consumed by the manycore to process an entire frame implementing different algorithms. Time starts when the initial frame is written in PMs, and the signal SCR.NRST gets low. It ends when all Tiles finish processing and set one to SCR.NDN bit.

We also considered only grayscale images (9 bits) and a 256x256 image resolution. Besides, there is a reduction in the total amount of pixels per tile when we increase the manycore size. This approach favors the speed since there are fewer pixels and more operations are performed in parallel by the PEs. However, this also increases the communication overhead, so the speedup is a linear function as the architecture increases exponentially in manycore sizes because of the NxN 2-D mesh grid network.

6.3.1 Resources utilization

Table 6.2 shows FPGA resources and other parameters used in the implementations, considering a 256x256 image, a maximum of 8 processing steps, and one image frame. These parameters are enough to run the HCD processing chain shown in Fig.6.1. Maximum frequency (in MHz) is calculated from the Worst Negative Slack (WNS) obtained from synthesis and implementation routines, and means the manycore maximum estimated working frequency in this specific FPGA fabric. It also shows the total on-chip power consumption estimated by the Xilinx Vivado tool. We must highlight that there was available space to insert more tiles. However, the synthesis/implementation tool could not find a feasible *place & route* solution.

All architectures ran at $100MHz$ with at least 14 MHz of margin compared to PL maximum frequency, as described in Table 6.2. It means that RTL design fulfilled physical constraints, reflecting in calculation and results integrity achieved. Logic Cells occupation more than duplicated for the 1 PE, 4 PEs, and 9 PEs, reducing the rate above 16 PEs. Network and external interface signals have less impact on bigger architectures, resulting in this rate reduction. On-Chip power shows a near $0.1W$ increase for smaller designs. After the 16 PEs many-core, there is an approximately $0.3W$ growth until 81 PEs.

6.3.2 Performance analysis

Experiments in this section were computed using architecture special registers: Total Counter Register (TCR), Network Counter Register (NCR), and Pixel Memory Counter Register (PMCR), as described in Section 3.5. Time results only take the Manycore performance into account, not considering image transfer to or from the IP. A chessboard picture is used as a benchmark because it contains regular corner numbers for similar HCD execution timing. Data results are taken one

Table 6.2: Implementation report for a 256x256 image, 8 max. steps and 1 frame at 100 MHz.

	Device	1 Tile	4 Tiles	9 Tiles	16 Tiles
PL Max. Freq. (MHz)	-	119.72	130.17	126.04	123.00
LUT	230400	4084	9410	19543	31434
LUTRAM	101760	358	1126	2406	4198
FF	460800	3169	7934	16365	28014
BRAM	312	160	160	167	160
URAM	96	1	4	9	16
DSP Blocks	1728	4	16	36	64
Estimated Power (W)	-	3.470	3.545	3.680	3.852
	25 Tiles	36 Tiles	49 Tiles	64 Tiles	81 Tiles
PL Max. Freq. (MHz)	123.35	123.50	117.36	120.06	114.63
LUT	50860	75991	101424	123399	168748
LUTRAM	6502	9318	12646	16486	20838
FF	43946	62914	86065	112091	142653
BRAM	182	176	179	160	194
URAM	25	36	49	64	81
DSP Blocks	100	144	196	256	324
Estimated Power (W)	4.066	4.373	4.617	4.984	5.246

time for each algorithm and size.

Figure 6.5 shows the performance in frames per second per architecture for all implemented algorithms. This scenario considers the processing time with the input image already distributed inside the PMs and the final image stored in the PMs.

The many-core architecture presents an exponential speedup in all cases because of the exponential size increase. But, there is an almost linear speedup in terms of performance per tile. The 3x3 Convolution and Sobel filters were able to achieve 909 and 1000 fps, respectively. Higher performance denotes better parallelism exploration. Also, the speedup maintains linear for all architectures, meaning that we should expect a constant performance growth for larger architectures, until reaching the point that the communication overhead overcomes the parallelism speed up (see Subsection 6.3.4), and the performance starts to degrade.

As expected, Harris Corner is the most time-consuming task from the graphic, followed by 7x7, 5x5, 3x3 convolutions, and Sobel filter, respectively. This order is maintained in all Many-core sizes, despite similar values in cases where the parallelism is better explored. Comparing HCD block diagram from Fig. 6.1 with Fig. 6.6, 5x5 Gaussian Filters from the algorithms takes almost 75% of the time, comparing to 7x7 Convolution filter from the graphic. This happens because HCD performs 3 convolutions in the same step, using 50 pixels from Sobel X and Y, which is close to 49 neighbor pixels from 7x7 Convolution. Other HCD steps are compared to 3x3 Convolution (Smoothing Filter) and Sobel from Fig. 6.6, taking about 25% of the execution time. The Sobel algorithm takes less time compared to 3x3 Convolution because of branch optimizations.

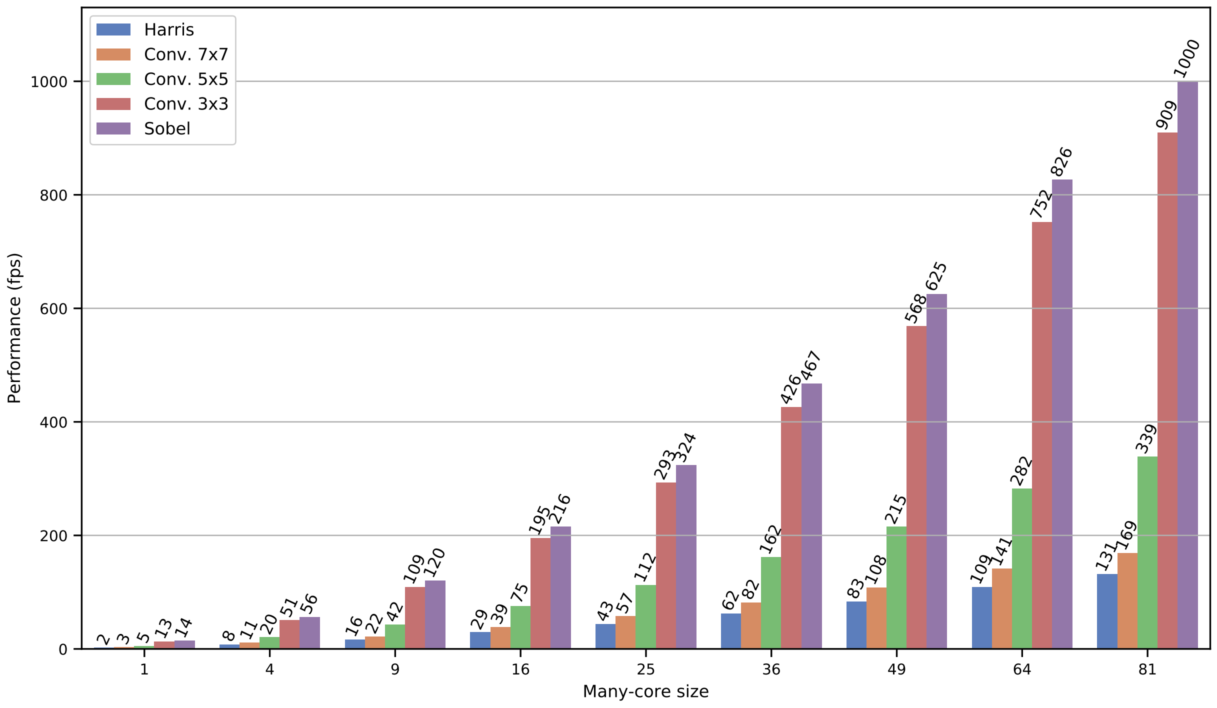

Figure 6.5: Performance in frames per second per architecture, all running at 100 MHz on the physical device.

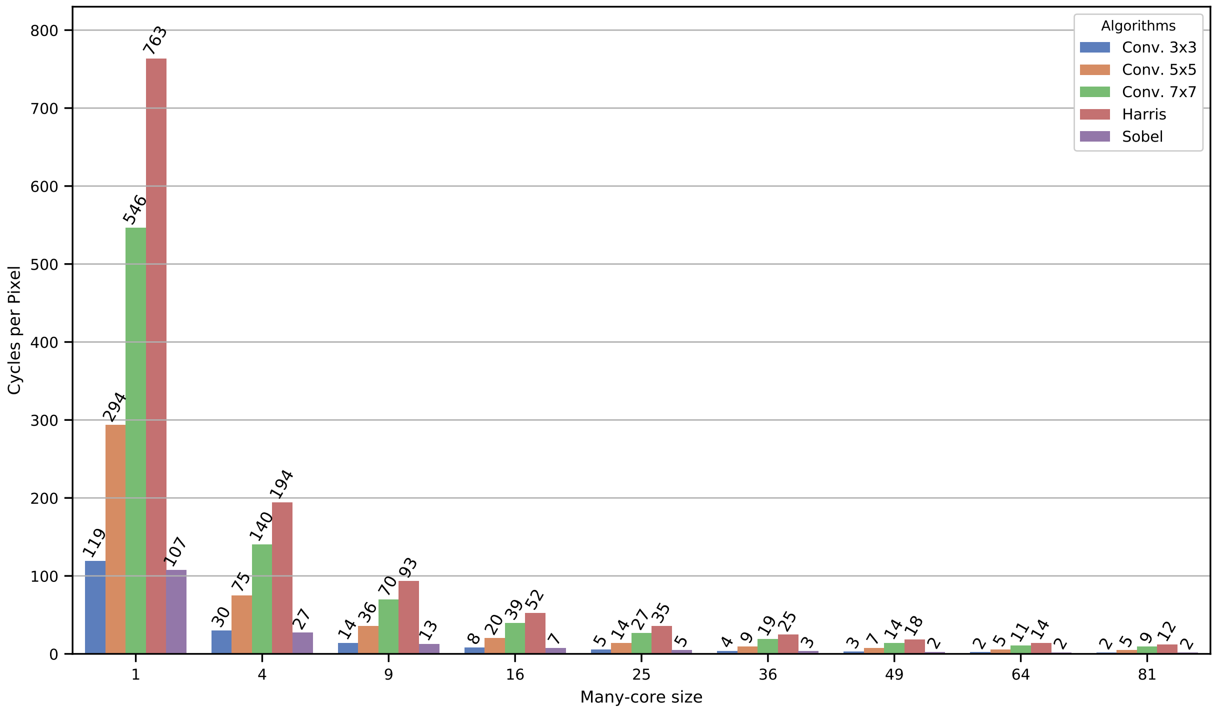

Figure 6.6: Cycles per Pixel per architecture for all algorithms, 256x256 image at 100 MHz.

6.3.3 Architecture comparison

Table 6.3: Comparison of implementations of Sobel Edge Detector.

Reference	Architecture	N^o of cores	cycles/ pixel	Programmable?
(36) 2019 *	MPSoC-FPGA	4	1.06	no
Ours (81 PEs implementation)	MPSoC-FPGA	81	1.52	yes
(29) 2015	MPSoC-ASIC	16	2.67	yes
(35) 2019	MPSoC-FPGA	4	64.13	yes

* includes the initial image transfer time.

A comparison, only for the Sobel Edge Detector against our 81 tiles implementation, is shown in Tab.6.3. All related works involve Multiprocessor SoCs. (36) builds a method to program heterogeneous MPSoCs using the Xilinx SDSoC framework. It implements an Edge Detection algorithm in multiple scenarios: software only, HW/SW with static and runtime task mapping/scheduling. (29) uses a 2D mesh NoC based 16 RISC core processor to implement different image processing task for its parallel programming model called threaded MPI. (35) develops a high-level synthesis tool to facilitate time-to-market Heterogeneous MPSoCs design, using a MPI-based programming model and Vivado Tools for HLS and TCL scripting.

Table 6.4 compares our best timing performance MPSoC with related architectures for the Harris Corner Detector application. We show in the table two results: the simulated one, with 400 tiles, and the implemented one (limited by the FPGA size), with 81 tiles. (42) utilizes a Jetson (ARM and GPU) similar to this work, with a GPU instead of using programmable logic. (41) implements HCD on a ASIC SIMD architecture. (29) uses an NoC-based MPSoC with 16 RISC cores assisted by an external ARM CPU. (19) builds the application in a processor array using VLIW PEs and point-to-point communication, building a tightly-coupled processor array. (37), (40) (18), and (17) implements pipelined HCD architectures in FPGA.

It is not easy to compare different architecture types, as well as different VLSI systems. To have a kind of normalization, we computed the $cycles/pixel$, which is a metric that shows how efficiently a computing architecture implements an IP/CV application. This metric is independent of the VLSI technology and the operating frequency. We believe that, with this normalization, we can fairly compare the architectural designs from several years ago until now. Our architecture achieves closer cycles per pixel compared to similar programmable devices.

In general, non-programmable pipelined FPGA architectures have better results but lacks flexibility. However, other device types can also achieve similar performance, in terms of frames per second, for ICs that can run at higher frequencies but consume more power. It is important to emphasize that this work explored the platform physical limits in the Zynq ZCU104 board and still can reduce HCD execution time for larger FPGAs. Moreover, our solution brings full flexibility with a performance close to non-programmable architectures. (29) is also a NoC-based MPSoC with RISC processors, and the most similar implemetation to our architecture in Table 6.4. Our

Table 6.4: Comparison of implementations of the HCD.

Reference	Architecture	cycles/ pixel	Programmable?
(17) 2014 **	FPGA	1.00	no
(18) 2014	FPGA	1.00	no
(19) 2013	FPGA	1.36	yes
(40) 2017 **	FPGA	2.11	no
Ours (400 PEs simulation)	MPSoC-FPGA	3.02	yes
(37) 2013 **	FPGA	3.03	no
(41) 2010	SIMD ASIC	3.42	yes
(29) 2015	MPSoC-ASIC	7.07	yes
(42) 2018 *	Embedded GPU	7.19	yes
Ours (81 PEs implementation)	MPSoC-FPGA	11.61	yes

* includes the initial image transfer time.

** includes acquisition and visualization scheme.

Many-core performed more than two times faster compared with their work, considering the 400 PEs version.

6.3.4 Pixel Memory access

To check the architecture's parallelism limit, Figure 6.7 shows the time the PE spent in pixel accesses through the Router (using the NoC to request/receive pixels from other tiles), the local PM, and total pixel access, as a percentage of the execution time. We can see that pixel accesses represent from 26% to almost 35% of the application time. The rest of the execution time is spent by the PE effectively processing data. Network overhead increases on a logarithmic scale more than PE-PM usage decreases. It means more time is spent to access memory for larger architectures and consequent reduction in performance gain with scalability. These results denote that there will be no sensitive improvement after a specific many-core size and, from this point, any increase in the number of Tiles would start to degrade the performance.

6.3.5 Parallelism preview

Fig. 6.8 depicts a comparison among implemented and simulated results for the HCD algorithm. It gives a perspective of future performance and parallelism exploration for larger many-core systems as we are limited to the FPGA fabric size in our implementations.

Both simulated and implemented data have similar results for all cases until 81 PEs, showing that simulation can be a good estimator for our MPSoC real-world design. There is an increase in performance for all simulated architectures after 81 PEs, achieving more than $500\ fps$. It indicates that our many-core is capable of exploring until at least 400 PEs without performance saturation. This graphic confirms suppositions in Section 6.2 that the architecture can scale.

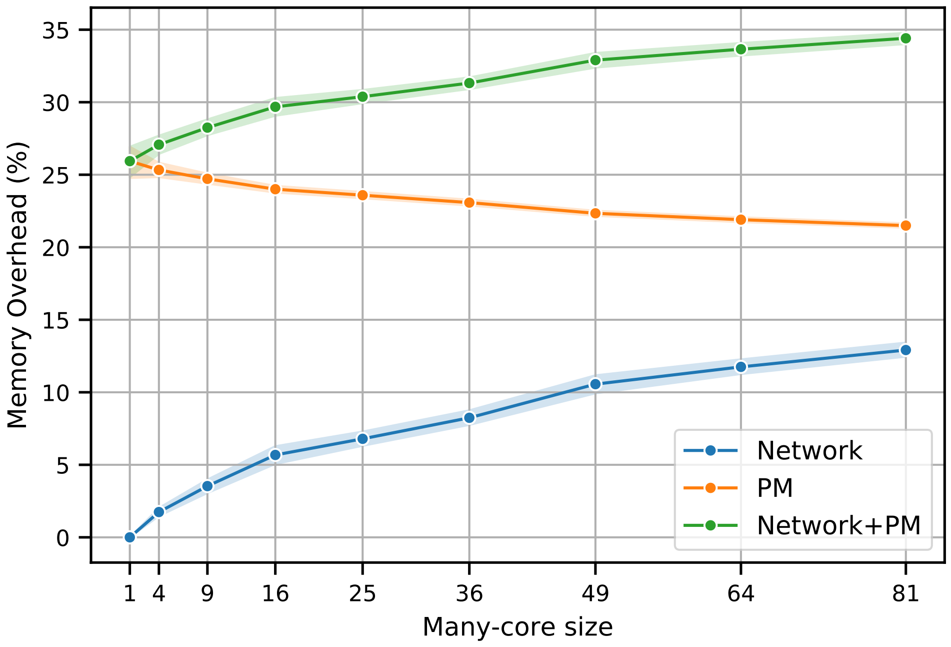

Figure 6.7: Pixel access time as a percentage of the total execution time.

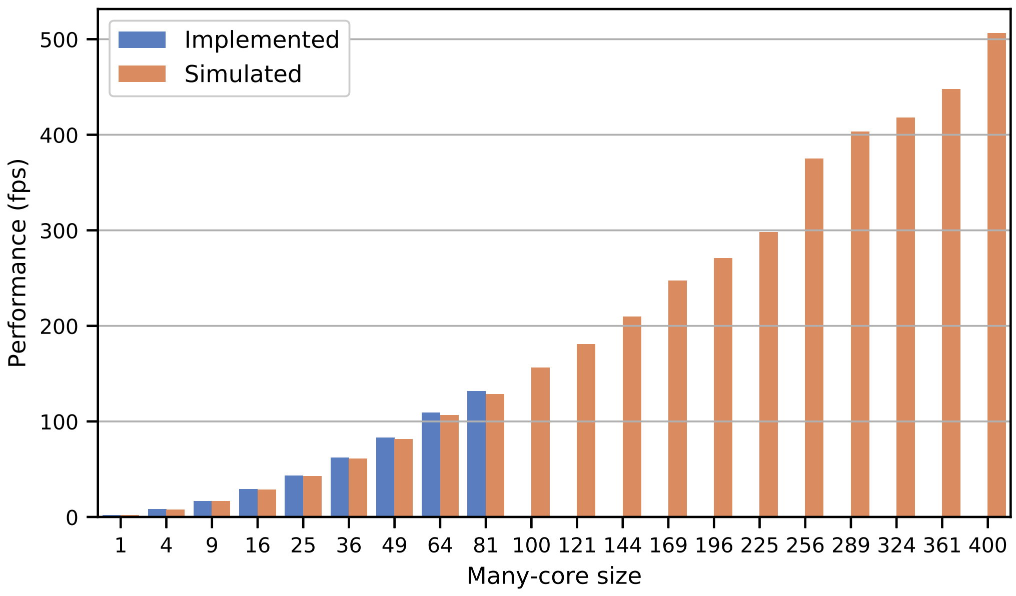

Figure 6.8: Performance in frames per second comparing implemented and simulated results for the Harris Detector.

As expected, implemented designs present slightly better performance because of synthesis and implementation optimizations.

6.4 COMPLETE IP/CV PROCESSING CHAIN

This section has the results of the complete IP/CV processing chain, from the acquisition in the OV7670 sensor to the ARM DRAM memory availability. All data use QVGA image resolution with the PL running at 100 MHz. We intend to address general characteristics of the Many-core, discussing its advantages and disadvantages. Figure 6.9 shows a photography of the OV7670 CMOS sensor and Infineon power monitor device attached to the ZCU104 Development Kit, used in this Section's results.

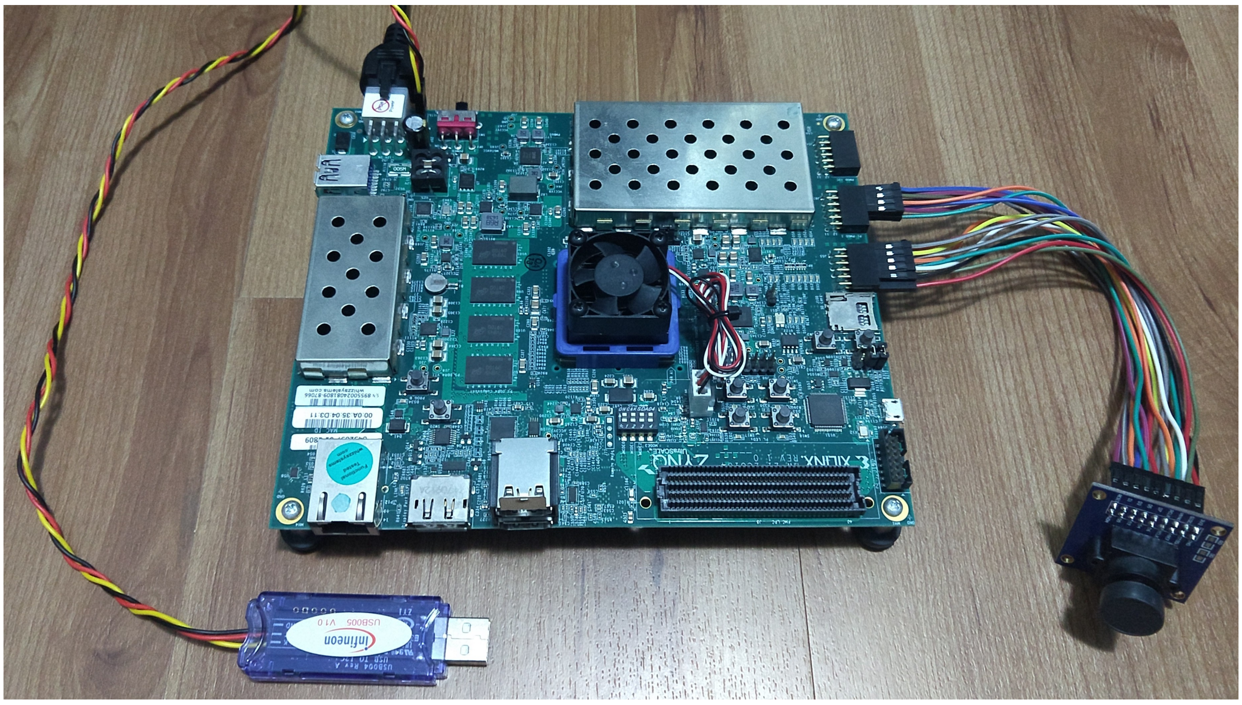

Figure 6.9: Photography of the ZCU104 Development Kit with the OV7670 CMOS sensor and the Infineon power monitor device.

6.4.1 Resource usage

Table 6.5 has resources usage of the complete chain, for different architecture sizes, until 81 PEs, which gets to the physical limit of the Xilinx Zynq UltraScale+ MPSoC ZCU 104 Evaluation Kit. Compared to Table 6.2, there are more than 4 thousand LUTs for 1 PE on the final version, mainly due to the increase in AXI Interconnect IPs that have five master and two slave interfaces. These masters control the Visualization Frame Buffer, the Acquisition IP, the AXI DMA IP, a BRAM Controller for the DMA Scatter Gather Mode, and the Many-core Vision Processor. Two slave interfaces get two High-Performance Full Power Domain (HPM FPD) from the ARM processor. Also, there is a 307 % increase in FF usage, suggesting that the complete version has more sequential logic components from these new IPs and interfaces.

Block Memories have no constant increase rate, even with the same image size, and consume different units depending on sub-images size and distribution. For example, 25 PEs uses the same number of memories as the 1 PE version because each tile receives a 64x48 sub-image, which has optimal placement in the FPGA fabric. However, the sub-image has 40x30 size for the 8x8 manycore and presents the least optimized placement, and utilizes more resources.

Table 6.5: Resources for the complete IP/CV processing chain, QVGA image at 100 MHz.

	Device	1 Tile	4 Tiles	9 Tiles	16 Tiles
PL Max. Freq. (MHz)	-	134.77	128.85	122.80	122.88
LUT	230400	9841	16057	26117	39765
LUTRAM	101760	789	1557	2837	4629
FF	460800	9733	15219	24164	36843
BRAM	312	195	197	198	205
URAM	96	1	4	9	16
DSP Blocks	1728	6	18	38	66
Estimated Power (W)	-	3.568	3.646	3.774	3.969
	25 Tiles	36 Tiles	49 Tiles	64 Tiles	81 Tiles
PL Max. Freq. (MHz)	120.61	122.01	122.56	122.70	111.42
LUT	56837	80944	110941	141772	178582
LUTRAM	6933	9749	13077	16917	21269
FF	53240	73421	97350	124997	156400
BRAM	195	219	234	237	207
URAM	25	36	49	64	81
DSP Blocks	102	146	198	258	326
Estimated Power (W)	4.198	4.444	4.742	5.077	5.401

Each tile uses an Ultra RAM block as in the previous case. Power consumption, using the Xilinx Power Estimation tool, had approximately a 3 % increase. It shows that the off-chip camera attached to the board does not demand significant power compared to the ZCU board. Next, there are camera captured image results and analysis.

6.4.2 Performance evaluation

The performance experiment was set to capture 500 frames, where the camera is set to a free-capture mode, saving all the frames in the Acquisition Frame Buffer. The ARM processor controls the MPSoC to wait for the camera VSYNC signal and guarantee correct frame processing. As image transfer time from the Acquisition FB to PMs takes about 1 ms, which is similar to the DMA transfer time, there is no pixel loss in free-capture with the camera producing 60 fps[1].

The timer starts to count when the ARM finishes to set up and initialize peripherals (the

[1]Chapter 3 explains these signals and IPs in details.

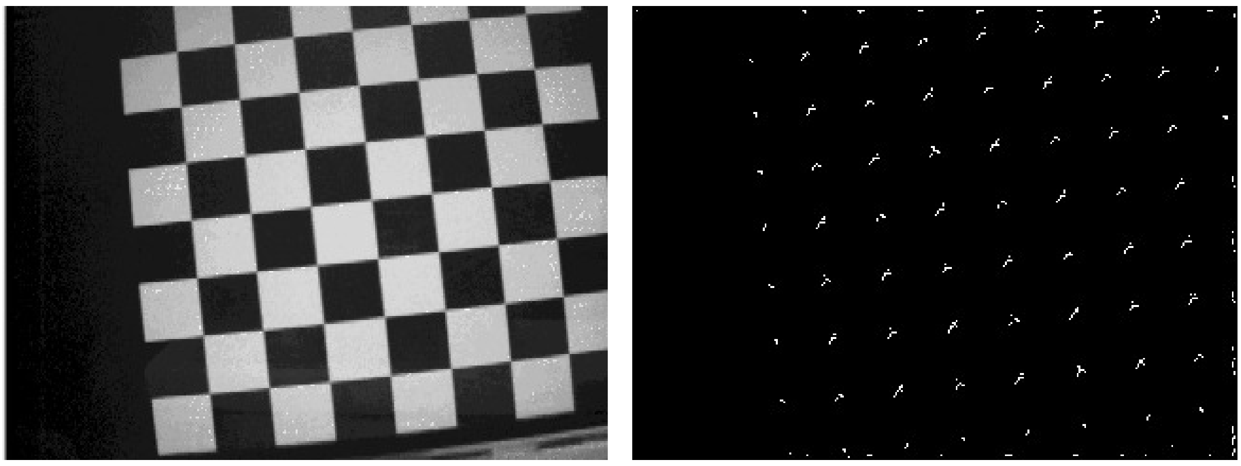

Figure 6.10: Example of initial camera captured image of a chessboard (left) and the final Harris frame for the same picture (right).

MCVP, DMA, camera, and Acquisition FB) and stops to count when the DMA IP sends the last Descriptor high-level status bit (50), after writing the last pixel in ARM's DRAM. We use the APU MPCore System Counter, sometimes referred to as the global counter, as a clock counter to compute performance in the current experiment. It is a 64-bit counter documented in the Cortex-A53 MPCore Technical Reference Manual (51).

Five algorithms are implemented: the Harris, Sobel, 3x3, 5x5, and 7x7 Convolutions. Figure 6.10 (left) shows a chessboard captured image from the sensor and read from ARM DRAM through a UART interface, for evaluation purposes. Figure 6.10 (right) depicts the final Harris detector for the left image. Note that the algorithm detects corners even with our architecture implementing an integer version of it, with pixel data precision limited to 9 bits. Hence, we can conclude from visual inspection that the MCVP implementation produces valid output images in the complete processing chain. It is not shown here, but this result is also achieved for the other algorithms.

Figure 6.11 shows time performance results for the algorithms executed, from 1 to 81 PEs. There are near linear performance improvements in all algorithms until 9 PEs, after that, Sobel and 3x3 Convolution stopped at near 30 fps. These two just doubled performance after reaching less than 16 ms per frame, where the architecture was able to overcome the camera VSYNC limitation, and 49 tiles until to 81 could get maximum performance of 60 fps for the current acquisition scheme.

More demanding algorithms, the HCD, 5x5, and 7x7 Convolutions, have performance gain but do not achieve near linear speedup, keeping 30 fps after 49, 16, and 64 PEs, respectively. Even with these inferences, the camera limits performance and increases the gaps between neighbor architectures until its saturation. This is a pitfall of the VSYNC acquisition control scheme, which has a performance reduction to guarantee frames coherency. Hence, a possible improvement is to choose a camera able to produce more frames per second to improve performance and granularity.

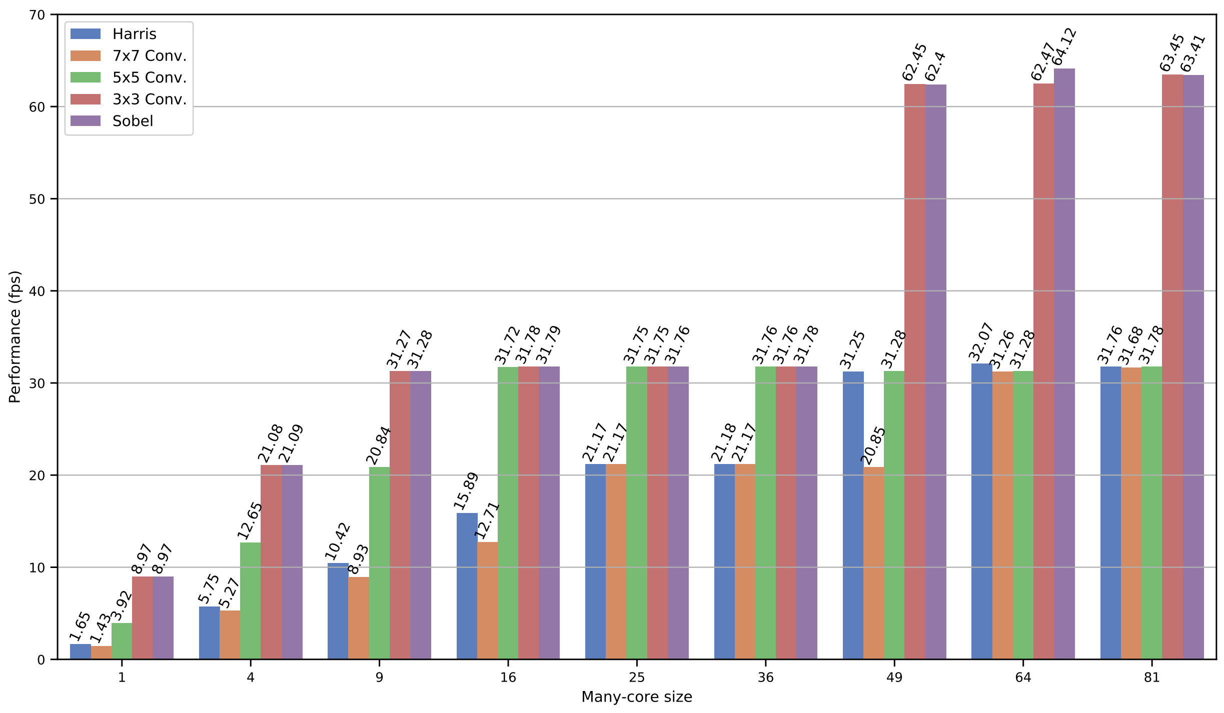

Figure 6.11: Execution time for multiple algorithms with acquisition and visualization scheme.

6.4.3 Power consumption

The ZCU104's power system has components to monitor voltage and current on the power rails by the Infineon manufacturer (58). Even though the board already has the components to power monitoring, and they are linked to the internal I2C bus, we chose to use a commercial solution to ease data acquisition and confidence. For this reason, an Infineon USB005 USB cable (59) is used to get power results directly from an I2C pin connection from the board.

We used the PowIRCenter GUI application to get power results from all rails. Considering that the ZCU104 development kit shares different peripherals in the same rail, mixing the FPGA fabric with DRAM power supply (60), for example, it is not possible to separate FPGA from ARM or other subsystems. For this reason, experimental power results concern the total instantaneous consumption in Watts (W) for all rails, which is also available in the desktop application.

As the PowIRCenter does not have a data logging system or any way to store the data from power monitors ICs, we filmed the app screen to analyze its data using video editing tools. Similar to precious timing results, this subsection also tests the Harris, Sobel, 3x3, 5x5, and 7x7 Convolutions, varying the number of processing frames with the algorithm and manycore size. Table 6.6 shows base frames[2] for each architecture used in the experiment. All algorithms process these frames quantity except for the Sobel and 3x3 Convolution filters, that process base frames times two. These last algorithms are faster than the others, that is why we decided to increase the frame count, to take more time, and get more power measurements.

[2]Number of frames processed for most algorithms.

Table 6.6: Power experiment base frames for each archtecture size.

Arch. size (Tiles)	Base frames
1	50
4	200
9	500
16	500
25	500
36	500
49	500
64	500
81	500

Figure 6.12 has the power performance graphic for different algorithms and architecture sizes. The programs have linear power consumption increase in most cases, except for the Harris detector. Lines and points represent the measurements mean, and the shaded area is the 95 % confidence interval based on the mean estimator (61). The power behavior relates to the whole architecture components: the image capture system, DMA, MCVP, ARM processor, and other internal board subsystems. The camera runs in free-capture, like in Subsection 6.4.2, with no clock-gating or power shutoff features. The linear increase in mean power is most caused by the FPGA fabric area usage, because the architecture components keep the same for all Many-core sizes except for the PL part.

With more tiles, more instantaneous power consumption because there are more pixel transmission between each of them. With more routers and memory in use, less predictable is the data transferences between each tile and routers' queues occupation. Besides, as we deal with a big digital system, in terms of resource usage that consumes thousands of logic units, the Flip-Flops clock distribution is not completely uniform, causing propagation delays. Combining the MCVP behavior with the FPGA and physical characteristics, the system has variable and less predictable instantaneous power consumption, leading to a standard deviation increase.

We explained the higher power demand but did not describe a reason for algorithm curve differences. Memory write operation usually demands more power than memory read, write can consume from 1.3 to 16.5 times more power for NAND flash (62) (63). We can infer that FPGAs Block RAMs, even being a different type of memory, also demands more power for write operations. The Harris algorithm is the only one that consumes more than 2 steps to finish. As Sobel, 3x3, 5x5, and 7x7 Convolution filters have one initial step (the captured image) and the final one (with the filtered image), the Harris has to write in four intermediary images, and finally get the last one.

This would probably be the main reason for its higher power consumption, in addition to the fact that 49 and 64 Tiles uses more BRAM units due to lack of optimization. More significant are the write operations with more units working, because there are more logic elements to be powered and maintained. A proof of the last argument is the reduction rate of the Harris for 81

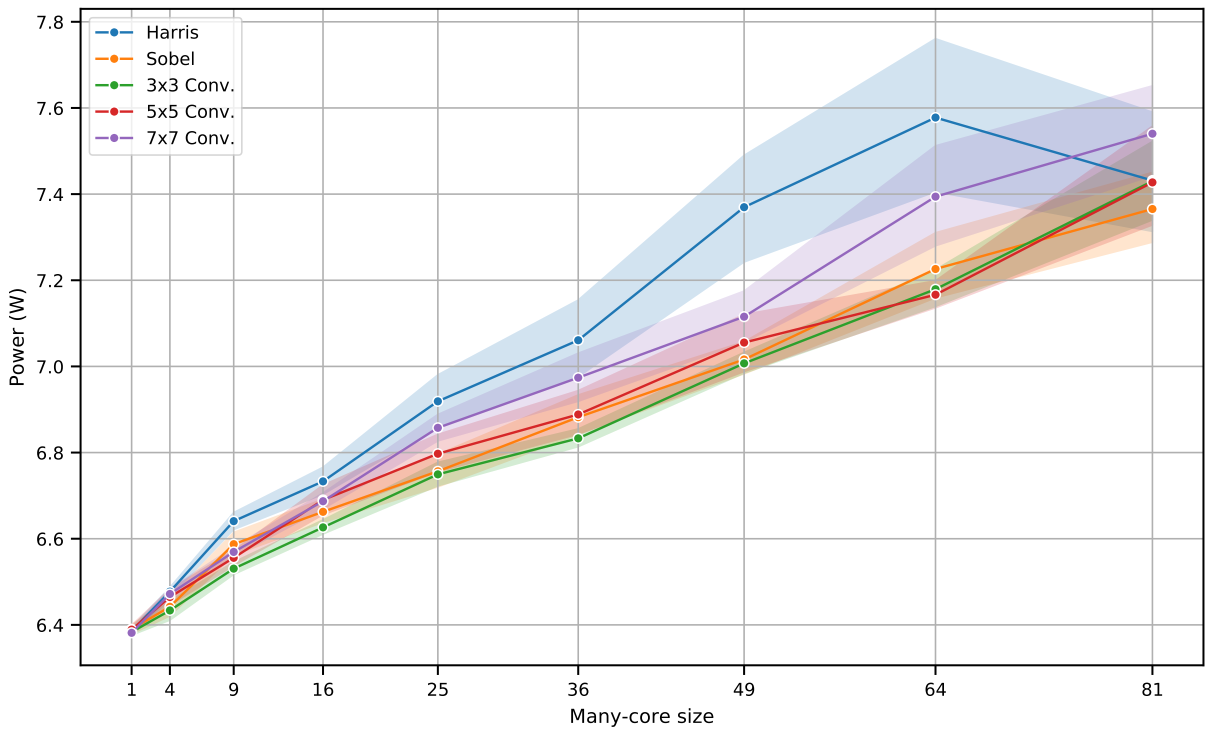

Figure 6.12: Power consumed for different algorithms and architecture sizes.

Tiles because, according to Table 6.5, it uses fewer Block Memories and has better performance.

In addition to that, idle time can also influence power consumption. With the camera running in free-capture and the MCVP waiting for the VSYNC signal, depending on the timing performance, the manycore can wait longer for frames and consume less power.

7 CONCLUSION

In this work, we developed a manycore architecture for real-time Image Processing and Computer Vision applications. Our approach makes use of *Region-based* acquisition scheme and operation level parallelism to optimize the processing time. Additionally, we designed a pixel distribution and control unit utilizing an embedded processor commonly available in modern FPGAs. All subsystems are addressed and explained with construction details, focusing on describing an FPGA overlay to explore the platform design possibilities better.

As a proof-of-concept, we implemented some IP/CV algorithms: the Harris Corner Detector, Sobel Edge Detector, and Convolution filters. The results show that our architecture is capable of overcoming similar real-time architecture depending on the manycore size and application demands. The architecture is also flexible and easy to scale for higher numbers of tiles. The linear enhancement of processing speed while increasing the number of PEs shows that we could achieve the aim of exploring the natural parallelism of IP/CV algorithms.

We built an automation tool to generate multiple architecture sizes and tested it in a complete IP/CV processing chain, from acquisition with an OV7670 camera, to storing the final processed image in ARM processor DRAM. Timing results showed that our architecture has increasing performance with the size, but the camera system is a significant bottleneck. We concluded that FPGA resource usage, memory write operations and idle time are the most significant variables that impact power consumption.

We implemented an Application Specific Instruction Set Processor (ASIP) with 16 instructions for specific IP/CV applications needs. With the automation tool, we were able to produce 81 cores in the FPGA fabric and up to 400 in Vivado Simulator. Also, power results have linear increase with the manycore size, which usually presents exponential behavior, and means that the architecture has efficient energy consumption considering the power differences between manycore sizes.

As future works, we envision the development of an EDA tool, from our automation tool, to provide better memory management and test different task mapping techniques (as in this work we executed the same code in all Processing Elements). The architecture shall also be validated for multiple-frames algorithms, like motion estimation, tracking, and pattern recognition. A custom compiler using the LLVM is also another possible upgrade for this works' ASIP architecture, as shown in (64).

We also envision installing embedded Linux in the ARM Application processor, leaving the Real-time one to deal with the MCVP and other time-constrained tasks. Linux is a good choice because allows easy use of its drivers for WiFi connection, for example. In this case, the DMAC referred in (16) can be a good option depending on the application demand. With this feature, we could send a video stream with the post-processed images for an external computer or server, to enable remote control and monitoring.

Besides, we envision to optimize the PE and PM glue logic to reduce its size and complexity, and to analyse and optimize (if necessary) overall memory access time. Also, we envision to create a pipelined version of the PE and analyse its impact on the manycore performance and resource usage. Adopt a wormwhole data flow in the Router is another idea for future scalability improvements.

BIBLIOGRAPHY

1 BURGER, W.; BURGE, M. *Digital Image Processing: An Algorithmic Introduction Using Java*. [S.l.]: Springer London, 2016. (Texts in Computer Science). ISBN 9781447166849.

2 CORKE, P. *Robotics, vision and control: fundamental algorithms in MATLAB® second, completely revised*. [S.l.]: Springer, 2017. v. 118.

3 SZELISKI, R. *Computer vision: algorithms and applications*. [S.l.]: Springer Science & Business Media, 2011.

4 FORCHHEIMER, R.; ASTROM, A. Near-sensor image processing: A new paradigm. *IEEE Transactions on Image Processing*, IEEE, 1994.

5 FOSSUM, E. R. Low power camera-on-a-chip using cmos active pixel sensor technology. In: IEEE. *1995 IEEE Symposium on Low Power Electronics. Digest of Technical Papers*. [S.l.], 1995. p. 74–77.

6 YUDI, J.; LLANOS, C. H.; HUEBNER, M. System-level design space identification for many-core vision processors. *Microprocessors and Microsystems*, 2017.

7 SILVA, B. A.; LIMA, A. M.; YUDI, J. A manycore vision processor architecture for embedded applications. In: IEEE. *2020 X Brazilian Symposium on Computing Systems Engineering (SBESC)*. [S.l.], 2020. p. 1–8.

8 KOMURO, T.; KAGAMI, S.; ISHIKAWA, M. A dynamically reconfigurable simd processor for a vision chip. *IEEE journal of solid-state circuits*, IEEE, v. 39, n. 1, p. 265–268, 2004.

9 MORI, J. Y.; LLANOS, C. H.; HÜEBNER, M. A framework to the design and programming of many-core focal-plane vision processors. In: IEEE. *2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing*. [S.l.], 2015.

10 SCHMITZ, J. A.; GHARZAI, M. K.; BALKIR, S.; HOFFMAN, M. W.; WHITE, D. J.; SCHEMM, N. A 1000 frames/s vision chip using scalable pixel-neighborhood-level parallel processing. *IEEE Journal of Solid-State Circuits*, IEEE, v. 52, n. 2, p. 556–568, 2016.

11 TAYLOR, M. B. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In: *Proceedings of the 49th Annual Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 2012. Available at: ⟨https://doi.org/10.1145/2228360.2228567⟩.

12 BENINI, L.; MICHELI, G. D. Networks on chips: A new soc paradigm. *computer*, IEEE, v. 35, n. 1, p. 70–78, 2002.

13 SAHU, P. K.; CHATTOPADHYAY, S. A survey on application mapping strategies for network-on-chip design. *Journal of Systems Architecture*, v. 59, n. 1, p. 60 – 76, 2013. ISSN 1383-7621. Available at: ⟨http://www.sciencedirect.com/science/article/pii/S1383762112000902⟩.

14 DONZELLINI, G.; ONETO, L.; PONTA, D.; ANGUITA, D. Introduction to fpga and hdl design. Springer International Publishing, Cham, p. 465–517, 2019. Available at: ⟨https://doi.org/10.1007/978-3-319-92804-3_9⟩.

15 SADRI, M.; WEIS, C.; WEHN, N.; BENINI, L. Energy and performance exploration of accelerator coherency port using xilinx zynq. In: *Proceedings of the 10th FPGAworld Conference*. New York, NY, USA: Association for Computing Machinery, 2013. (FPGAworld '13). ISBN 9781450324960. Available at: ⟨https://doi.org/10.1145/2513683.2513688⟩.

16 MOLANES, R. F.; COSTAS, L.; RODRIGUEZ-ANDINA, J. J.; FARINA, J. Comparative analysis of processor-fpga communication performance in low-cost fpsocs. *IEEE Transactions on Industrial Informatics*, IEEE, 2020.

17 POSSA, P. R.; MAHMOUDI, S. A.; HARB, N.; VALDERRAMA, C.; MANNEBACK, P. A multi-resolution fpga-based architecture for real-time edge and corner detection. *IEEE Transactions on Computers*, IEEE, v. 63, n. 10, p. 2376–2388, 2013.

18 AMARICAI, A.; GAVRILIU, C.-E.; BONCALO, O. An fpga sliding window-based architecture harris corner detector. In: IEEE. *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. [S.l.], 2014. p. 1–4.

19 SOUSA, É. R.; TANASE, A.; HANNIG, F.; TEICH, J. Accuracy and performance analysis of harris corner computation on tightly-coupled processor arrays. In: IEEE. *2013 Conference on Design and Architectures for Signal and Image Processing*. [S.l.], 2013. p. 88–95.

20 KEHTARNAVAZ, N.; GAMADIA, M. Real-time image and video processing: from research to reality. *Synthesis Lectures on Image, Video & Multimedia Processing*, Morgan & Claypool Publishers, v. 2, n. 1, p. 1–108, 2006.

21 YUDI, J. *Development of design framework for parallel data processing hardware architectures*. Thesis (doctoralthesis) — Ruhr-Universität Bochum, Universitätsbibliothek, 2018.

22 MORI, J. Y.; LLANOS, C. H.; BERGER, P. A. Kernel analysis for architecture design trade off in convolution-based image filtering. In: IEEE. *2012 25th Symposium on Integrated Circuits and Systems Design (SBCCI)*. [S.l.], 2012. p. 1–6.

23 KADI, M. A.; JANSSEN, B.; YUDI, J.; HUEBNER, M. General-purpose computing with soft gpus on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 2018. Available at: ⟨https://doi.org/10.1145/3173548⟩.

24 HOOZEMANS, J.; JONG, R. de; VLUGT, S. van der; STRATEN, J. V.; ELANGO, U. K.; AL-ARS, Z. Frame-based programming, stream-based processing for medical image processing applications. *Journal of Signal Processing Systems*, Springer Science and Business Media LLC, 2019. Available at: ⟨https://doi.org/10.1007/s11265-018-1422-3⟩.

25 JOSHI, J.; KARANDIKAR, K.; BADE, S.; BODKE, M.; ADYANTHAYA, R.; AHIRWAL, B. Multi-core image processing system using network on chip interconnect. In: IEEE. *2007 50th Midwest Symposium on Circuits and Systems*. [S.l.], 2007. p. 1257–1260.

26 JOSHI, J.; BADE, S.; BATRA, P.; ADYANTHAYA, R. Real time image processing system using packet based on chip communication. *The National Conference on Communications, 2007*, 2007.

27 FRESSE, V.; AUBERT, A.; BOCHARD, N. A predictive noc architecture for vision systems dedicated to image analysis. *EURASIP Journal on Embedded Systems*, Springer, 2007.

28 SAPONARA, S.; FANUCCI, L.; PETRI, E. A multi-processor noc-based architecture for real-time image/video enhancement. *Journal of Real-Time Image Processing*, Springer, 2013.

29 ROSS, J. A.; RICHIE, D. A.; PARK, S. J. Implementing image processing algorithms for the epiphany many-core coprocessor with threaded mpi. In: *2015 IEEE High Performance Extreme Computing Conference*. [S.l.: s.n.], 2015.

30 KADI, M. A. Fgpu: a flexible soft gpu architecture for general purpose computing on fpgas. 2018.

31 LEE, S.-H.; YANG, C.-S. A real time object recognition and counting system for smart industrial camera sensor. *IEEE Sensors Journal*, IEEE, v. 17, n. 8, p. 2516–2523, 2017.

32 ALI, K. M.; ATITALLAH, R. B.; HANAFI, S.; DEKEYSER, J.-L. A generic pixel distribution architecture for parallel video processing. In: IEEE. *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. [S.l.], 2014.

33 KHALIL, K.; ELDASH, O.; KUMAR, A.; BAYOUMI, M. A speed and energy focused framework for dynamic hardware reconfiguration. In: IEEE. *2019 32nd IEEE International System-on-Chip Conference (SOCC)*. [S.l.], 2019. p. 388–393.

34 MORI, J. Y.; HÜBNER, M. Multi-level parallelism analysis and system-level simulation for many-core vision processor design. In: IEEE. *2016 5th Mediterranean Conference on Embedded Computing (MECO)*. [S.l.], 2016. p. 90–95.

35 RETTKOWSKI, J.; GÖHRINGER, D. Sdmpsoc: Software-defined mpsoc for fpgas. *Journal of Signal Processing Systems*, Springer, 2019.

36 SURIANO, L.; ARRESTIER, F.; RODRíGUEZ, A.; HEULOT, J.; DESNOS, K.; PELCAT, M.; TORRE, E. de la. Damhse: Programming heterogeneous mpsocs with hardware acceleration using dataflow-based design space exploration and automated rapid prototyping. *Microprocessors and Microsystems*, 2019. ISSN 0141-9331.

37 AYDOGDU, M. F.; DEMIRCI, M. F.; KASNAKOGLU, C. Pipelining harris corner detection with a tiny fpga for a mobile robot. In: IEEE. *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. [S.l.], 2013. p. 2177–2184.

38 AMARA, A. B.; PISSALOUX, E.; GRISEL, R.; ATRI, M. Zynq fpga based memory efficient and real-time harris corner detection algorithm implementation. In: IEEE. *2018 15th International Multi-Conference on Systems, Signals & Devices (SSD)*. [S.l.], 2018. p. 852–857.

39 HSIAO, P.-Y.; LU, C.-L.; FU, L.-C. Multilayered image processing for multiscale harris corner detection in digital realization. *IEEE Transactions on Industrial Electronics*, IEEE, v. 57, n. 5, p. 1799–1805, 2010.

40 LIU, S.; LYU, C.; LIU, Y.; ZHOU, W.; JIANG, X.; LI, P.; CHEN, H.; LI, Y. Real-time implementation of harris corner detection system based on fpga. In: IEEE. *2017 IEEE International Conference on Real-time Computing and Robotics (RCAR)*. [S.l.], 2017. p. 339–343.

41 HOSSEINI, F.; FIJANY, A.; FONTAINE, J.-G. Highly parallel implementation of harris corner detector on csx simd architecture. In: *Euro-Par 2010 Parallel Processing Workshops*. [S.l.]: Springer Berlin Heidelberg, 2011.

42 CHAHUARA, H.; RODRÍGUEZ, P. Real-time corner detection on mobile platforms using cuda. In: IEEE. *2018 IEEE XXV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*. [S.l.], 2018. p. 1–4.

43 HAGGUI, O.; TADONKI, C.; LACASSAGNE, L.; SAYADI, F.; OUNI, B. Harris corner detection on a numa manycore. *Future Generation Computer Systems*, v. 88, p. 442 – 452, 2018. ISSN 0167-739X. Available at: ⟨http://www.sciencedirect.com/science/article/pii/S0167739X1732188X⟩.

44 CANNY, J. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, IEEE, n. 6, p. 679–698, 1986.

45 OMINIVISION. 2008 fall product guide. 2008.

46 XILINX. *UltraScale Architecture SelectIO Resources v1.12 - User Guide*. [S.l.], 2019. UG571.

47 KENDRI, D. Fpga camera system. *Hackster.io*, 2019. Available at: ⟨https://www.hackster.io/dhq/fpga-camera-system-14d6ea⟩. Acessed on: 21 Oct 2020.

48 WIKIPEDIA, T. F. E. Grayscale. 2020. Available at: ⟨https://en.wikipedia.org/wiki/Grayscale⟩. Acessed on: 24 Oct 2020.

49 XILINX. *Vivado Design Suite User Guide - Synthesis*. [S.l.], 2020. V2019.2.

50 XILINX. *AXI DMA v7.1 - LogiCORE IP Product Guide*. [S.l.], 2019. PG021.

51 XILINX. *Zynq UltraScale+ Device v2.1 - Technical Reference Manual*. [S.l.], 2019. UG1085.

52 ASHWORTH, M.; RILEY, G.; ATTWOOD, A.; MAWER, J. First steps in porting the lfric weather and climate model to the fpgas of the euroexa architecture. *Scientific Programming*, IOS Press, out. 2019. ISSN 1058-9244.

53 PRABHU, G. Dlx instruction set. Available at: ⟨http://web.cs.iastate.edu/~prabhu/Tutorial/PIPELINE/DLXinstrSet.html⟩. Acessed on: 9 Feb 2021.

54 PATTERSON, D.; WATERMAN, A. *Guia prático RISC-V Atlas de uma Arquitetura Aberta Primeira edição, 1.0.0*. [S.l.]: Strawberry Canyon LLC, 2019.

55 INC., T. K. G. Khronos releases openvx 1.3 open standard for cross-platform vision and machine intelligence acceleration. 2019. Available at: ⟨https://khr.io/sc⟩. Acessed on: 5 Dec 2020.

56 ECMA. Ecma-404 the json data interchange standard. 2017. Available at: ⟨http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf⟩. Acessed on: 5 Dec 2020.

57 HARRIS, C.; STEPHENS, M. A combined corner and edge detector. In: *In Proc. of Fourth Alvey Vision Conference*. [S.l.: s.n.], 1988. p. 147–151.

58 XILINX. *ZCU104 Evaluation Board v1.1 - User Guide*. [S.l.], 2018. UG1267.

59 INTERNATIONAL RECTIFIER - INFINEON TECHNOLOGIES. *USB005 User Guide*. [S.l.], 2014. V1.0.

60 XILINX. *HW-Z1-ZCU104 Evaluation Board (XCZU7EV-2FFVC1156) - Schematic*. [S.l.], 2018. V1.0-rev01.

61 WASKON, M. Seaborn lineplot (seaborn.lineplot). 2020. Available at: ⟨https://seaborn.pydata.org/generated/seaborn.lineplot.html⟩. Acessed on: 23 Jan 2021.

62 PARK, S.; KIM, Y.; URGAONKAR, B.; LEE, J.; SEO, E. A comprehensive study of energy efficiency and performance of flash-based ssd. *Journal of Systems Architecture*, Elsevier, v. 57, n. 4, p. 354–365, 2011.

63 LIN, N.; DONG, Y.; LU, D. Providing virtual memory support for sensor networks with mass data processing. *International Journal of Distributed Sensor Networks*, SAGE Publications Sage UK: London, England, v. 9, n. 3, p. 324641, 2013.

64 GOLDBERG, C. J. *The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend*. Thesis (Thesis) — Rochester Institute of Technology, August 2017. Available at: ⟨https://scholarworks.rit.edu/theses/9550/⟩.

APPENDICES

A THE HW/SW PLATFORM REGISTER SPACE

A.1 ACQUISITION IP

This section explains the AXI Camera Control IP organization in terms of register space. It uses the AXI4-Lite interface with 32-bit wide registers to communicate with the ARM processor.

A.1.1 CCR: Camera Control Register

Offset: 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
															STOP

Figure A.1: Camera Control Register.

This register provides control to the Acquisition IP. Figure A.1 shows the Camera Control Register (CCR) addresses, as following:

- **STOP** (undefined by default, read/write): when active, stops new frame acquisition by the OV7670 Capture module. If this bit is zero, new frames are acquired following the camera configuration.

A.1.2 CSR: Camera Status Register

Offset: 0x04

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
															VSYC

Figure A.2: Camera Status Register.

This register gives the status of the Acquisition IP. Figure A.2 shows the Camera Status Register (CSR) addresses, as follow:

- **VSYC** (undefined by default, read-only): gives the state of VSYNC pin. High indicates the end of a frame, low elsewhere.

A.2 MANY-CORE VISION PROCESSOR

There are fourteen registers to control the MCVP, read/write data in Pixel and Instruction Memories, selects a tile in GPI, get timing performance, and store the final step image during run-time. Each of them is explained below.

A.2.1 SCR: Status and Control Register

Offset: 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODE															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					NDN	IMF	PMF				NRST	IWEN	PWEN	PEN	PSEL

Figure A.3: Status and Control Register.

This register provides general purpose control and status signals in Pixel and General Purpose Interfaces. Bits 0 to 4 are for control, 8 to 10 are for status and 16 to 31 are for mode control in GPI. Figure A.3 shows the Status and Control Register (SCR) addresses, as following:

- **PSEL** (undefined by default, read/write): enables Pixel Selection from the ARM. If this signal is high, the processor has exclusive access to Pixel Memories asynchronously. Otherwise, Frame Buffers have access to PMs as the MCVP's FSM controls pixel transfer, without external intervension;
- **PEN** (undefined by default, read/write): Enables Pixel Memories by ARM side if equals to '1'. Disables if '0';
- **PWEN** (undefined by default, read/write): Pixel Memories' Write Enable by ARM side. This registers allow the processor to write pixels if high. Else, it has no effect on PMs;
- **IWEN** (undefined by default, read/write): Instruction Memory Write Enable. As the name refers, it lets IMs writing if high. Nothing happens if this bit is low. Note that IMs are always enable, this design choice was made because these memories need to be on during Processing Elements work (which occurs most of the time);
- **NRST** (undefined by default, read/write): Network Reset signal. If high resets all Processing Elements, Routers and Pixel Memories in the Network. Otherwise, does nothing. When PEs get the reset signal, they restart their Program Counter to run that program other time;
- **PMF** (undefined by default, read-only): Pixel Memory Flag, gives the state of a PM based on Pixel Addresses inserted in XYR and SFR. If high, the PM finished its operation (read or write), if low the memory is still processing or disabled;

- **IMF** (undefined by default, read-only): Instruction Memory Flag, gives the state of a IM based on MODE signal. If high, the IM finished its operation (read or write), if low the memory is still processing or disabled;
- **NDN** (undefined by default, read-only): Network Done. If high, the tile given by MODE signal in the Network have finished to process the task. Which includes: read initial image from Acquisition FB, execute the program and write final image in Visualization FB;
- **MODE** (undefined by default, read/write, 16 bits): select a Tile to interface via GPI. Because of this, the Many-core is limited to $65,535$ cores: 1 address is exclusive for stand-by mode, or no Tile selected, and others $2^{16} - 1 = 65,535$ can be used. Each Tile has a unique address accordingly to the following equations,

$$x(mode) = \begin{cases} 1, & \text{if } mode \bmod x_{lim} \text{ is } 0 \\ (mode \bmod x_{lim}) + 1, & \text{otherwise,} \end{cases} \tag{A.1}$$

$$y(mode) = \begin{cases} 1, & \text{if } mode/y_{lim} \text{ is } 0 \\ mode/y_{lim} + 1, & \text{otherwise,} \end{cases} \tag{A.2}$$

  where $mode$ is **MODE** signal value, $x(mode)$ and $y(mode)$ are tile's x and y coordinate in 2D-mesh grid (starting from top left corner), x_{lim} and y_{lim} are Network's x and y limits, respectively. mod operator is modulo, equivalent to remainder of a division, and the $\%$ operator in C programming language. This work treats the special occasion where $x_{lim} = y_{lim}$.

A.2.2 XY Register

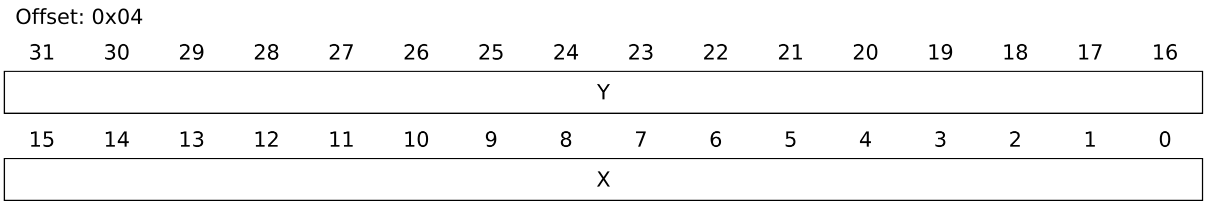

Figure A.4: X/Y Register.

This register provides control over x and y properties of Pixel Interface used in Pixel Memories addressing, Section 4.2 has more information about it. They represent discrete cartesian coordinates corresponding to pixels in a given image. Figure A.4 shows the X/Y Register (XYR) addresses, as following:

- **X** (undefined by default, read/write, 16 bits): set x image coordinate for PI, limited to a $2^{16} = 65,536$ x image size. It is compatible with all comercial image resolutions available in nowadays market, with a reasonable gap;

- **Y** (undefined by default, read/write, 16 bits): set y image coordinate for PI, limited to a $2^{16} = 65,536$ y image size. Like **X** case, it is also compatible with comercial image resolutions.

A.2.3 Step/Frame Register

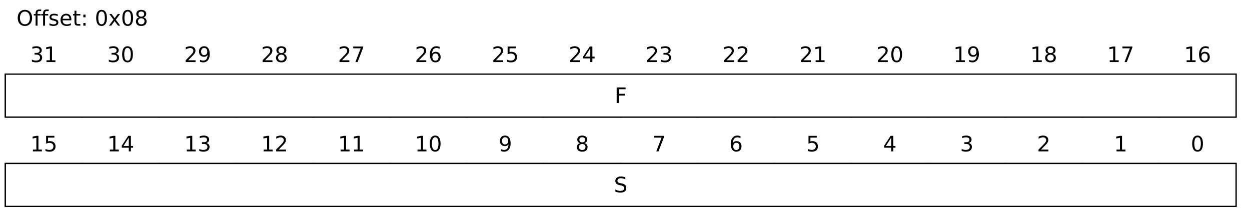

Figure A.5: Step/Frame Register.

This register provides control over *step* and *frame* pixel parameters used in Pixel Interface, refer to Section 4.2. Figure A.5 shows the Step/Frame Register (SFR) addresses, as following:

- **S** (undefined by default, read/write, 16 bits): set s (step) image coordinate for PI, limited to $2^{16} = 65,536$ steps. A step is a block, or unique image, in the processing pipeline that serves as input for other image operations, or it is the final image itself;
- **F** (undefined by default, read/write, 16 bits): set f (frame) image coordinate for PI, limited to $2^{16} = 65,536$ frames. A frame is a different image of the same step, taken at a distinct time or using other sensor. For example, some stereo vision algorithms need two different cameras for the same step, in this context, one can address 0 and 1 to f to distinguish between these two sensors.

A.2.4 Write Pixel Register

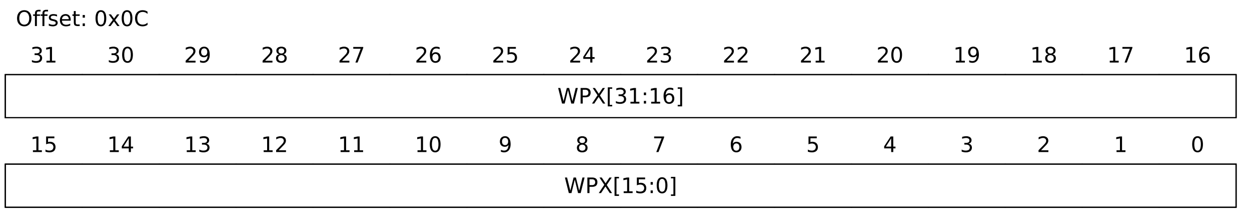

Figure A.6: Write Pixel Register.

This register stores the Pixel value to be written in Pixel Memories. PEN and PWEN enable pixel transfer, while XYR and SFR address pixel parameters. Figure A.6 shows the Write Pixel Register (WPR) addresses, as following:

- **WPX** (undefined by default, read/write, 32 bits): set Pixel value to be written in Pixel Memory, depends on XYR, SFR, PEN, and PWEN (both from SCR).

A.2.5 Instruction Address Register

Offset: 0x10

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IA[31:16]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IA[15:0]															

Figure A.7: Instruction Address Register.

This register controls the address of the Instruction Memory through General Purpose Interface. Depending on the MODE value from SCR, this register chooses which address the ARM is going to write. Figure A.7 shows the Instruction Address Register (IAR) addresses, as following:

- **IA** (undefined by default, read/write, 32 bits): set Instruction Address value in a specific Instruction Memory, defined by MODE value in SCR. **Warning:** IA has to be an unsigned integer multiple of 4: 0, 4, 8, etc. This decision allows word memory addressing by IMs, but until now, this feature was not released.

A.2.6 Instruction Data LSB Register

Offset: 0x14

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ID[31:16]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID[15:0]															

Figure A.8: Instruction Data LSB Register.

This register stores the first half of the data to be written in Instruction Memories. Figure A.8 shows the Instruction Data LSB Register (IDLR) addresses, as following:

- **ID** (undefined by default, read/write, 64 bits): set bits 31 to 0 (Least Significant Bytes) of Instruction Data for an Instruction Memory defined by MODE value in SCR. ID holds instruction for the Processing Element and has a total of 64-bit do guarantee enough space for multiple Many-core settings, for example, 8k resolution processing.

A.2.7 Instruction Data MSB Register

This register stores the second half of the data to be written in Instruction Memories. Figure A.9 shows the Instruction Data MSB Register (IDMR) addresses, as following:

- **ID** (undefined by default, read/write, 64 bits): set bits 63 to 32 (Most Significant Bytes) of Instruction Data for an Instruction Memory defined by MODE value in SCR. ID holds

Offset: 0x18

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ID[63:48]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID[47:32]															

Figure A.9: Instruction Data MSB Register.

instruction for the Processing Element and has a total of 64-bit do guarantee enough space for multiple Many-core settings, for example, 8k resolution processing.

A.2.8 Read Pixel Register

Offset: 0x1C

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
RPX[31:16]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RPX[15:0]															

Figure A.10: Read Pixel Register.

This register gets the Pixel value stored at a specific address in Instruction Memories. PEN and PWEN enable pixel transfer, while XYR and SFR address pixel parameters. Figure A.10 shows the Read Pixel Register (RPR) addresses, as following:

- **RPX** (undefined by default, read-only, 32 bits): Read Pixel, gets Pixel value stored in Pixel Memory, depends on XYR, SFR, PEN, and PWEN (both from SCR).

A.2.9 Total Counter Register

Offset: 0x20

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TC[31:16]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TC[15:0]															

Figure A.11: Total Counter Register.

This register stores a counter value containing the Total execution time of a Tile, defined by the MODE signal. Figure A.11 shows the Total Counter Register (TCR) addresses, as following:

- **TC** (undefined by default, read-only, 32 bits): Total Counter stores the number of clock cycles when the Tile starts processing (its reset signal gets low) until the processor triggers the done flag. This counter measures the total execution clock cycles of a Tile. The MODE signal from SCR controls the reception of TC using the General Purpose Interface.

A.2.10 Network Counter Register

Offset: 0x24

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NC[31:16]															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
NC[15:0]															

Figure A.12: Network Counter Register.

This register stores a counter value containing the Network time spent of a Tile, defined by the MODE signal. Figure A.12 shows the Network Counter Register (NCR) addresses, as following:

- **NC** (undefined by default, read-only, 32 bits): Network Counter stores the number of clock cycles when the Tile starts processing (its reset signal gets low). It ends with the last pixel transmission from the Network-on-Chip, made by the Tile's Processing Element. This counter measures the number of clock cycles spent in the network by a single Tile. The MODE signal from SCR controls the reception of NC using the General Purpose Interface.

A.2.11 Pixel Memory Counter Register

Offset: 0x28

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PMC[31:16]															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
PMC[15:0]															

Figure A.13: Pixel Memory Counter Register.

This register stores a counter value containing the Pixel memory access time of a Tile, defined by the MODE signal. Figure A.12 shows the Pixel Memory Counter Register (PMCR) addresses, as following:

- **PMC** (undefined by default, read-only, 32 bits): Pixel Memory Counter stores the number of clock cycles when the Tile starts processing (its reset signal gets low). It ends with the last pixel transmission from the Pixel Memory and Processing Element (PE-PM) interface (read Chapter 4 for details). This counter measures direct Pixel Memory access time, in clock cycles, spent by a single Tile. The MODE signal from SCR controls the reception of PMC using the General Purpose Interface.

A.2.12 Sensor Interface Counter Register

Offset: 0x2C

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SIC[31:16]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SIC[15:0]															

Figure A.14: Sensor Interface Counter Register.

This register stores a counter value with time spent by the CMOS Sensor Interface for the entire MCVP. Figure A.14 shows the Sensor Interface Counter Register (SICR) addresses, as following:

- **SIC** (undefined by default, read-only, 32 bits): Sensor Interface Counter stores the number of clock cycles when the MCVP starts to receive camera pixels. It ends with the last pixel transmission from the Acquisition IP. This counter measures time spent to get the complete initial image, in clock cycles, by the manycore in the Pixel Interface.

A.2.13 Visualization Interface Counter Register

Offset: 0x30

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
VC[31:16]															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VC[15:0]															

Figure A.15: Visualization Interface Counter Register.

This register stores a counter value with time spent by the Visualization Interface for the entire MCVP. Figure A.15 shows the Visualization Interface Counter Register (VICR) addresses, as following:

- **VIC** (undefined by default, read-only, 32 bits): Visualization Interface Counter stores the number of clock cycles after the MCVP finishes processing the algorithms. It ends with the last pixel transmission from the MCVP to the Visualization IP. This counter measures time spent to write the complete final image, in clock cycles, by the manycore in the Pixel Interface.

A.2.14 Final Step Register

This register stores the final step design parameter (from Section 3.1). Figure A.15 shows the Final Step Register (FSR) addresses, as following:

Offset: 0x34

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FS[31:16]															
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
FS[15:0]															

Figure A.16: Final Step Register.

- **FS** (undefined by default, read-only, 32 bits): gets the Final Step of the final image that the MCVP has to read from PMs and store in the Visualization IP. It allows real-time selection of desired step and the implementation of different algorithms (with different step number) during programming time.

A.3 VISUALIZATION FRAME BUFFER

This section shows the register space for the Visualization Frame Buffer.

A.3.1 Visualization Frame Buffer Register

Offset: 0x00

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
															SD

Figure A.17: Visualization Frame Buffer Register.

This register has the function of informing the ARM processor when Visualization Frame Buffer finished transfering the final image and has sent it to AXI DMA via the AXI Stream interface. Figure A.17 shows the Visualization Frame Buffer Register (VFBR) addresses, as following:

- **SD** (undefined by default, read-only): Stream Done, gets active high when the Visualization FB's FSM finishes processing all pixels in the image, sending them to the ARM DRAM via AXI DMA IP. Otherwise, the signal is always low.

B PROGRAMMING SEQUENCE

This appendix presents an explanation about the complete Many-core Vision Processor programming sequence, from the Acquisition to the AXI DMA IPs in Figure 3.1, to get the architecture working in the same way used in our work.

First, the DMA IP should be initialized using the scatter-gather mode (50) using the Stream to Memory Map (S2MM) interface. We assume the reader to have basic knowledge about AXI DMA operation modes. Initially, the DMA descriptor is set up using the sequence:

1. Write S2MM_NXTDESC withe the next descriptor address;
2. Store the initial DRAM data address to be written in the S2MM_BUFFER_ADDRESS;
3. Set the buffer length (S2MM_CONTROL.BufferLength) in bytes of data that will be transferred in current desciptor, also S2MM_CONTROL.RXSOF and S2MM_CONTROL.RXSOF fields if necessary;
4. Repeat first step until the last descriptor.

The DMA IP is configured following steps bellow:

5 Write the address of the starting descriptor for a 32 bit address space;

6 Start the S2MM channel by setting the run/stop bit (S2MM_DMACR.RS = 1). The halted bit (DMASR.Halted) has to be deasserted, indicating the channel is running;

7 Enable interrupts if desired setting the S2MM_DMACR.IOC_IrqEn and S2MM_DMACR.Err_IrqEn registers;

8 Write a valid address to the tail descriptor, indicating the last descriptor in the chain. It triggers the DMA to work, and stores data from the Stream interface directly in ARM DRAM.

Next, reset the Acquisition IP CCR.STOP to ensure the camera free-capture mode. Them, initialize the Many-core as following:

9 Write final step image in FSR.FS;

10 Set SCR.NRST to ensure that the MCVP is not working during program upload.

After, begin the Processing elements programming as the sequence:

11 Select the target tile to write the program using the SCR.MODE field;

12 Write IAR.IA with the instruction memory address that will receive IDLR.ID and IDMR.ID data (and should also be written if necessary);

13 Set SCR.IWEN bit to start instruction memory write process;

14 Wait for SCR.IMF gets high logic level;

15 Repeat from twelfth step to fourteenth until finishes to write the last instruction in a specific tile. Change SCR.MODE from step eleven to reach all available tiles and write all processors.

Wait for the next complete frame from the CMOS sensor, which occurs when CSR.VSYNC is equal to 1. Next, re-initialize the Many-core to start processing by resetting the SCR.NRST field. Wait the manycore finish its processing:

16 Select the target tile to get the done flag with the SCR.MODE field;

17 Get SCR.NDN signal. If its high, the tile finished, if not, it is still processing;

18 Repeat step 18th until all tiles have finished.

Get VFBR.SD value until it gets high logic level, meaning that the Frame Buffer finished to send all pixels to the AXI DMA IP. Wait for S2MM_STATUS.RXEOF triggers to '1' and flag the last byte sent from DMA to ARM's DRAM. Restart the process if needed by:

19 Reset S2MM_STATUS.RXEOF (all bits to zero);

20 Write a valid address to the tail descriptor to start DMA;

21 Return to step 19th.

C THE IP/CV ARCHITECTURE BLOCK DESIGN

Figure C.1 depicts the block diagram of the complete architecture based on the Vivado Block Design.

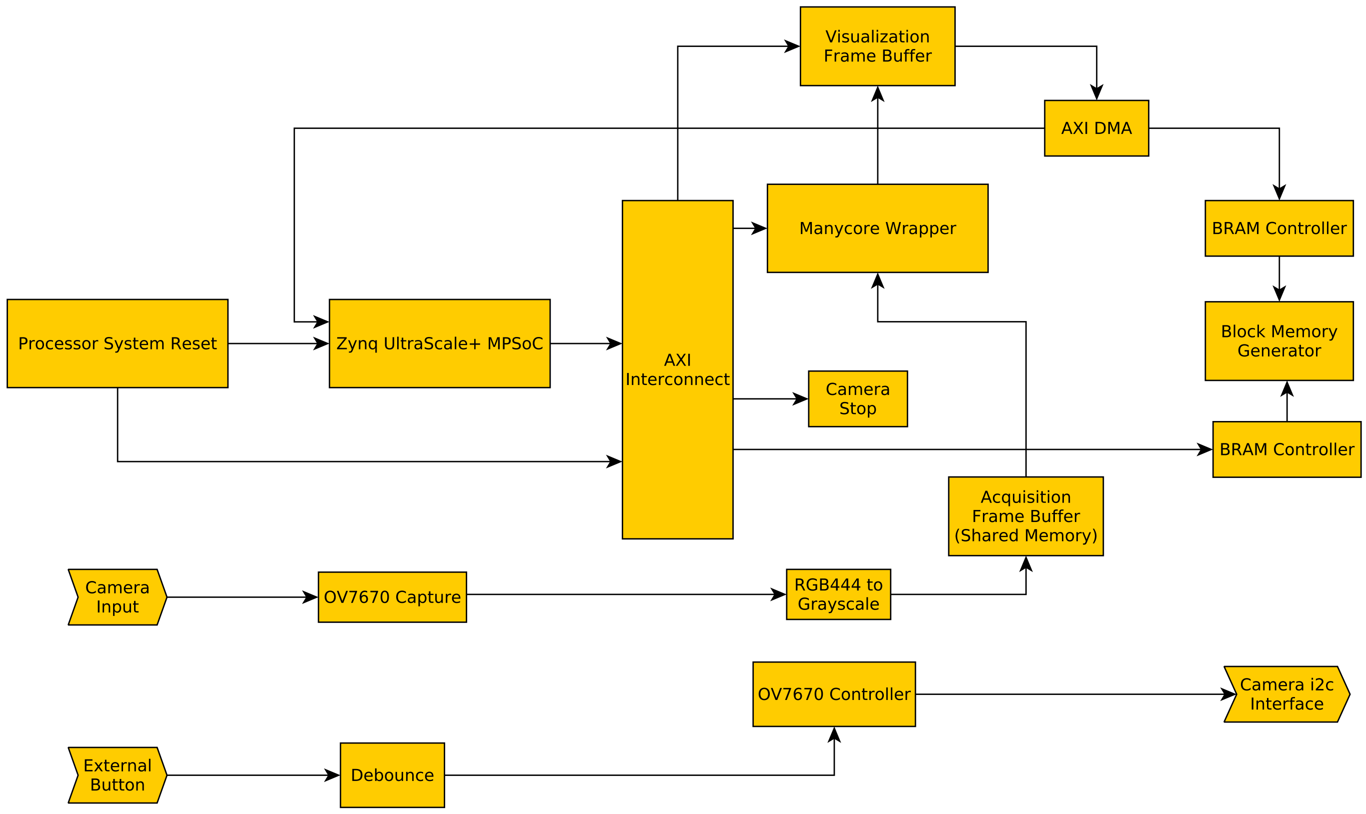

Figure C.1: Complete IP/CV architecture block diagram.

Figure C.2 shows the complete IP/CV architecture Block Design built in this work and represented in Figure C.1. It was captured from Vivado Design Suite.

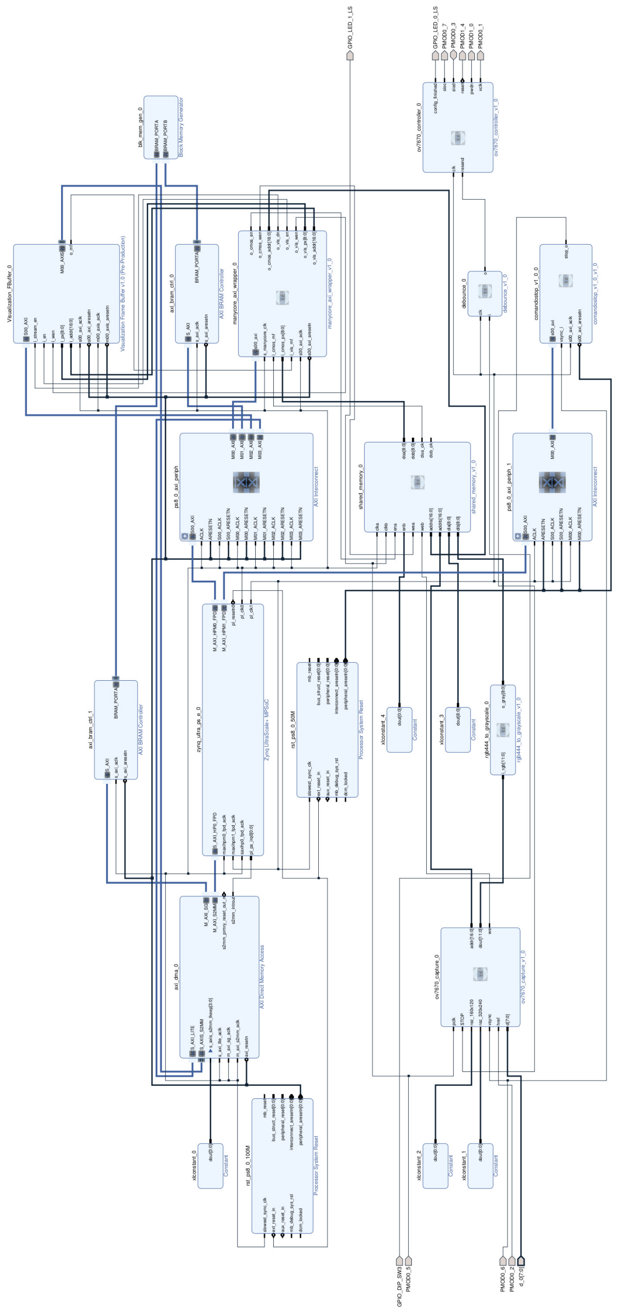

Figure C.2: Complete IP/CV architecture developed in Vivado Design Suite.