Development of a design framework for Parallel Data Processing hardware architectures

PhD Thesis

Jones Yudi Mori Alves da Silva

2018

Development of a design framework for Parallel Data Processing hardware architectures

Dissertation zur Erlangung des Grades eines ***Doktor-Ingenieurs*** der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

Jones Yudi Mori Alves da Silva

geboren in Brasilia, Brasilien

Erscheinungsjahr: 2018

1. Berichter: Prof. Michael Hübner - Ruhr-University Bochum, Germany
2. Berichter: Prof. Stephan Wong - Delft University of Technology, The Netherlands

Tag der mündlichen Prüfung: 12.01.2018

Copyright © 2018 Jones Yudi Mori Alves da Silva

Bochum, Germany, January 2018

Author contact information:

jonesyudi@unb.br

www.unb.br www.enm.unb.br

University of Brasilia - Faculty of Technology

Department of Mechanical Engineering

Automation and Control Research Group

This thesis was submitted to the Faculty of Electrical Engineering and Information Technology of the Ruhr-University Bochum on 18 December 2017 and presented on 12 January 2018.

Examination Committee:

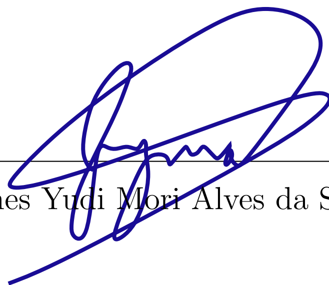
Prof. Michael Hübner	Ruhr-University Bochum	Thesis Advisor	(1st Referee)
Prof. Stephan Wong	Delft University of Technology	Committee Member	(2nd Referee)
Prof. Dorothea Kolossa	Ruhr-University Bochum	Committee Member	
Prof. Martin R. Hofmann	Ruhr-University Bochum	Committee Member	
Prof. Thomas Musch	Ruhr-University Bochum	Committee Member	

"The period of greatest gain in knowledge and experience is the most difficult period in one's life." - Dalai Lama XIV

Eidesstattliche Versicherung

Ich versichere an Eides statt, dass ich die eingereichte Dissertation selbstständig und ohne unzulässige fremde Hilfe verfasst, andere als die in ihr angegebene Literatur nicht benutzt und dass ich alle ganz oder annähernd übernommenen Textstellen sowie verwendete Grafiken, Tabellen und Auswertungsprogramme kenntlich gemacht habe. Außerdem versichere ich, dass die vorgelegte elektronische mit der schriftlichen Version der Dissertation übereinstimmt und die Abhandlung in dieser oder ähnlicher Form noch nicht anderweitig als Promotionsleistung vorgelegt und bewertet wurde.

Bochum, January 29, 2018



Jones Yudi Mori Alves da Silva

Abstract

Smart Cameras are devices able to not only acquire images but to perform complete Image Processing and Computer Vision (IP/CV) applications. A significant part of these applications has real-time constraints, e.g. visual feedback control systems, autonomous vehicles with visual sensors, automatic quality inspection in industry, and so on. All these applications can fail if the data processing is not performed within a specific time frame. IP/CV algorithms, in general, need to perform several operations over massive amounts of data, which is a strong issue for current embedded systems. Besides, there are other typical limitations of embedded systems, e.g. speed, manufacturing cost, power consumption, temperature management, silicon area, fault tolerance, among others. Since some years, there is a trend in the computer architectures to become multi-processed. The Very-Large Scale of Integration (VLSI) technology already allows for the implementation of heterogeneous System-on-Chip (SoC) with several different processing units, allowing the applications to benefit from more efficient and specialised processing power. In the domain of Smart Cameras, one of the main issues is the massive data transfer from the pixel sensor to the processing part. Therefore, several works explore the concept of Near-Sensor Image Processing, which aims to bring acquisition and processing as close as possible, increasing the efficiency of data transfer. An extension of this concept is known as Focal-Plane Image Processing (FPIP), which explores parallel data acquisition and transference of pixel data to multiple processing units, allowing for high degrees of parallelism exploration, increasing the system's efficiency. In this work, we present an analysis of the design of future Many-Core embedded

systems for parallel data processing, with a particular focus on IP/CV applications. This work can be divided into three main parts, with different approaches and abstraction levels. We start our approach in a high-level of abstraction, from the application's developer perspective: how such applications should be developed and how to analyse them to extract enough information to define the underlying processing architectures efficiently. A methodology based on Task-Graph Clustering was developed and integrated with a Task-Graph Simulator, written in SystemC at the Transaction-Level, to profile applications and architectural possibilities simultaneously. The results generated in this part were used to reduce the design-space to fewer possibilities, which were then analysed in a lower abstraction level. In the second approach, another SystemC/TLM simulator was developed based on processing and communication blocks, resembling more detailed architectural features. This simulator was integrated into a tool able to estimate power consumption, silicon area, and timing characteristics. The results provided here offered more detailed information about how the different architectures would perform, however, to have an efficient implementation, an even lower level was explored. The third part of this work was based on the Register-Transfer Level, with cycle-accurate simulations in Very-High Scale of Integration Chip Hardware Description Language (VHDL) and synthesis results. A baseline architecture was defined and profiled to determine its efficiency when compared to related works. Several new concepts were then proposed and discussed with the aim to improve the baseline architecture, generating different conceptual architectures with application-specific optimisations.

Kurzfassung

Smart Cameras sind Kameras welche ein Bild nicht nur aufnehmen, sondern auch komplexe Bildverarbeitung Anwendungen auf diesem ausführen. Ein bedeutender Teil dieser Anwendungen hat Echtzeitbeschränkungen, z.B. visuelle Rückkopplungskontrollsysteme, autonome Fahrzeuge mit visuellen Sensoren und automatische Qualitätskontrollen in der Industrie. Alle diese Anwendungen können scheitern, wenn die Datenverarbeitung nicht innerhalb einer definierten Zeitspanne erfolgt.Darüber hinaus werden in IP/CV Anwendungen große Datenmengen in vielen einzelnen Optionen verarbeitet. Di daraus resultierenden Anforderungen sind bis heute schwer in eingebetteten Systemen zu realisieren und weitern die typischen Anforderungen an eingebettete Systeme, wie beispielsweise geringe Herstellungskosten und leistungsaufnahme, Temperaturmanagement sowie hohe Fehlertoleranz. Seit einigen Jahren gibt es im Bereich der Computer-Architekturen einen Trend hinzu Mehrkernsystem. Die VLSI-Technologie erlaubt bereits heute heterogene und mehrkernige System-on-Chips (SoCs), wodurch Anwedungen von spezialisierten und effizienteren Verarbeitungsressourcen profitieren können. In der Domäne der Smart Cameras ist eine der gößten Herausforderungen / eines der größten Probleme, der Datentransfer vom Pixel-Sensor zur Datenverarbeitungseinheit. Aus diesem Grund existieren verschiedene Arbeiten welche das Konzept der sensorhahen Bildverarbeitung untersuchen. Sensornahe Bildverarbeitung ziel darauf ab, die Datenaufnahme und Datenverarbeitung so nah wie möglich zusammenzubringen, und so die Effizienz der Datenübertragung zu steigern. Eine Erweiterung dieses Konzepts ist unter dem Namen "Focal-Plane Image Processing" (FPIP) bekannt. FPIP nutzt die parallele

Datenafnahme und den Transport zu mehreren Datenverarbeitungseinheiten. Dieses Konzept ermöglicht die Untersuchung hochgradiger Parallelisierung und dadurch eine Steigerung der System-Effizienz. In dieser Arbeit stellen wir unsere Analyse von zukünftigen eingebetteten Many-Core Systemen für die parallele Datenverarbeitung mit Fokus auf IP- und CV-Anwendungen vor. Die Arbeit kann in drei Hauptteile gegliedert werden, welche sich hinsichtlich der Ansätze und Abstraktionsschicht unterscheiden. Der erste vorgestellte Ansatz verfügt über einen hohen Abstraktionsgrad aus der Sicht eines Anwendungsentwicklers. Auf dieser Abstraktionsschicht definieren wir die Programmiersicht. Diese beinhaltet, wie Anwendungen entwickelt und analysiert werden, um die benötigten Informationen für die Entwicklung einer effizienten Systemarchitektur zu ermöglichen. Hierzu wurde eine Methodik für das effiziente Profiling von Anwendungen und der effizienten Exploration der Systemarchitektur entwickelt, welche auf Task-Graphen, High-Level-Synthese und Graph-Clustering basiert. Für die Umsetzung dieser Methodik wurde ein Task-Graph-Simulator auf Basis des SystemC/Transaction-Level-Modeling (TLM) implementiert. Die im Rahmen dieser Arbeit entstandenen Ergebnisse ermöglichen eine Reduktion des Entwicklungsraums. Die übrigen Implementierungsansätze wurden im Folgenden auf einem tieferen Abstraktionslevel untersucht. Der zweite vorgestellte Ansatz untersucht die verbleibende Menge der Implementierungsansätze im Entwicklungsraum auf einer tieferen Abstraktionsschicht. Dazu wurde ein weiterer SystemC/TLM Simulator für die Kommunikation zwischen den Datenverarbeitungseinheiten entwickelt. Der Simulator wurde in ein Werkzeug integriert, welches die Abschätzung der Leistungsaufnahme, Chip-Fläche sowie des zeitlichen Verhaltens ermöglicht. Die in diesem Abschnitt erstellten Ergebnisse liefern detailliertere Informationen über die Leistungsfähigkeit der untersuchten Architekturen. Der dritte vorgestellte Ansatz führt die Analyse auf einer weiteren, tieferen Abstraktionsschicht weiter. Dieser Ansatz basiert auf einer Modellierung auf dem Register-Transfer-Level (RTL) mit einer zyklenakkuraten Simulation in VHDL sowie Syntheseergebnissen der Systemarchitekturen. Darauf aufbauend wurde eine Basis-

architektur definiert und untersucht, deren Effizienz mit dem Stand der Technik verglichen wurde. Mehrere neuartige Konzepte werden vorgestellt und diskutiert. Unter Verwendung der gezogenen Schlüsse wurde die Basisarchitektur verbessert und verschiedene Architekturkonzepte mit anwendungsspezifischen Optimierungen entwickelt.

Acknowledgements

This work would never be possible without the support of several people and institutions to which I would like to express my gratitude.

My supervisor, Prof. Michael Hübner, who accepted the guidance of this thesis and provided all the necessary support (material and intellectual) to the successful development of this work.

Prof. Stephan Wong, who provided several suggestions/corrections to the final version of this text. I am sure that the text is much better after his work. I extend my compliments to the committee members, Prof. Dorothea Kolossa, Prof. Martin R. Hofmann, and Prof. Thomas Musch: the refined questions and comments were a challenge in the defence and also helped to improve the final version of this thesis.

The (patient) people from the administrative/technical staff of the Ruhr-University Bochum: Maren Arndt, Horst Gass, Linda Trogant, and Franziska Großman. Special acknowledge to the Welcome Centre team for the support of all the bureaucracy within the Foreigner's Office.

A special mention to all the journey companions from ESIT/MCA/AIS teams during this years: André W., Florian F., Javier, Keyvan, Tobias, Habib, Horst, Nadine, Jens, Safdar, Florian K, Tomás, Benedikt, Phillip, Max, Osvaldo, Fynn, Muhammed, Lester, Salma, Maren, Franziska, Nadine, Diana, Michael, Ivan, Lukas, André F., Dominik, Pierre, and Christian. Without all the scientific/technical/bureaucratic discussions, problem-solving, complains, jokes, and fun, these four years would have been more difficult. Many thanks to the students who worked under my supervision and are partially responsible for some results presented in this thesis:

André Werner, Arij Shallufa, Carsten Gautstuck, Fabian Stuckmann, Florian Fricke, Frederik Kautz, and Yannick Bihege.

To the friends spread over Europe, old and new, who gave me the mind-free time to distract myself from work: Mafrénd André, Johanna, Pablo, Dezyrre, Tomás, Eliane, Gustavo, André Carmona, Nathália, Mariana, Marina. I would be (even more) crazy without your friendship.

My old friends far in Brazil, who always enjoyed to talk about life, studies, work, politics, music, movies and non-sense. Technology has shortened the distances.

I am grateful to my family, who have always supported my natural curiosity about "how things work": you are the main reason for my choice of the academic career. I missed home every day.

I also acknowledge the Professors from the Department of Mechanical Engineering of the University of Brasília, who provided special support for this work.

This work was supported by a scholarship from the Brazilian Ministry of Education, through the CAPES Foundation (www.capes.gov.br), in the context of the "Science without Borders" program. Special acknowledge to the Brazilian People, whose taxes financed this work. I hope this thesis helps to return the investment.

Contents

Abstract iii

Kurzfassung v

Acknowledgements ix

I Introduction 1

1 Motivation 3

1.1 Embedded Systems 3

1.2 Image Processing and Computer Vision 5

1.3 Real-Time IP/CV Systems 7

1.4 Vision Processors 9

1.5 Multi/Many-Core Vision Processors 11

2 Thesis Description 15

2.1 Overview of the methodology 15

2.2 High-Level approach 17

2.3 Intermediate-Level approach 17

2.4 Low-Level approach 17

2.5 Thesis Contributions 18

2.6 Document Organization 18

II Preliminary Study 21

3 Related Work 23

3.1 Vision Processors 23

3.2 Processor Architecture Design 31

3.3 High-Level Synthesis methods 34

3.4 Many-core design and methodologies 38

4 Image Processing Concepts 47

4.1 Image Processing and Computer Vision Chain 47

4.2 Operation Types 49

4.3 The OpenVX Project 50

5 Application-Specific Hardware/Software Co-Design 53

5.1 Image Acquisition Schemes 53

5.1.1 Throughput 54

5.1.2 Initial Latency 55

5.1.3 Image Delay 56

5.1.4 Overview 57

5.2 Parallelism Analysis 59

5.2.1 Macro-Pipeline 60

5.2.2 Operation-Level Parallelism 60

5.2.3 Loop-Tiling / Spatial Parallelism 60

5.3 Tiled Parallel Architecture 62

III High-Level approach 65

6 The high-level methodology 67

6.1 Introduction 67

6.2 Methodology's Overview 69

7 Task-Graph Generator **73**

7.1 Introduction 73

7.2 Indexing of Pixels and Processing Elements 75

7.3 Tools and Libraries used 76

7.3.1 The Boost Graph Library 76

7.3.2 The Clang compiler Front-End 77

7.4 Generating the Task-Graph 78

7.4.1 Literals 78

7.4.2 Variable Access 79

7.4.3 Unary operators 81

7.4.4 Binary operators 83

7.4.5 Ternary operators 85

7.4.6 Function calls 87

7.4.7 Control Flow 89

8 Task-Graph Clustering **95**

8.1 Introduction 95

8.2 Force-Directed Clustering 95

8.3 Clustering in the multi-core architecture 98

8.4 Clustering Results 100

8.4.1 Convolution algorithm 100

8.4.2 Median-Filter 104

8.5 Analysis of the clustering method 106

9 Task-Graph Simulator **109**

9.1 Introduction 109

9.2 Task-Graph interpretation 110

9.3 Implementation Overview 112

9.4 Functional Sequence 114

9.5 Simulation Example 114

9.5.1 Clustering . . . 115
9.5.2 Simulation results . . . 116

10 Discussion **119**

IV Intermediate-Level approach **121**

11 Introduction **123**

12 The MPSoC Simulator **125**
12.0.1 Processing Elements . . . 126
12.0.2 Pixel Memory Organization . . . 128
12.0.3 Communication Structure . . . 130
12.0.4 Programming Model . . . 132
12.0.5 The Event Monitor . . . 133
12.0.6 Estimation of Power, Area and Timing . . . 134

13 Simulation Results **137**
13.1 Simulated parameters . . . 137
13.2 Area estimation . . . 138
13.3 Power estimation . . . 139
13.4 Performance estimation . . . 140
13.5 Results discussion . . . 141

V Low-Level approach **143**

14 The Baseline Architecture **145**
14.1 Pixel Memory . . . 146
14.2 Processing Element . . . 146
14.3 Router . . . 148
14.4 Profiling Results . . . 149
14.4.1 Pre-Processing . . . 149

14.4.2 Segmentation . . . 151
14.4.3 Feature Extraction . . . 151
14.4.4 Complete Application . . . 152
14.5 Conclusion . . . 154

15 Processing Element Design **155**
15.1 The High-Level Synthesis Approach . . . 155
15.1.1 Introduction . . . 155
15.1.2 System Development . . . 157
15.1.3 Use-cases . . . 164
15.1.4 HLS in the Many-Core architecture . . . 167
15.2 The ASIP Approach . . . 168
15.2.1 Introduction . . . 168
15.2.2 Application's Analysis . . . 169
15.2.3 ASIP Design . . . 169
15.2.4 Results . . . 175
15.2.5 The ASIP in the Many-Core Architecture . . . 179
15.3 Remarks . . . 180

16 Conclusion and Future Work **181**

VI Backmatter **185**

References **187**

Acronyms **215**

About the Author **219**

Publications **221**
16.1 Publications within the Ruhr-University Bochum . . . 221
16.2 Publications within the University of Brasilia . . . 223

Part I

Introduction

Chapter 1

Motivation

In this Chapter, we explain the motivation for this work in the context of current trends for embedded systems design and applications. We discuss the need for new types of Smart Sensors, and why the design of future embedded systems requires a paradigm change in the direction of Multi/Many-Core Architectures. In particular, we focus on the design of future generation Many-Core Vision Processors to be used in Smart Cameras. [1]

1.1 Embedded Systems

In the past decades, the computing systems evolved from large facilities to small embedded devices, following the technology evolution of the microelectronics industry. The Very-Large Scale of Integration (VLSI) technology, following the prediction of the famous Moore's Law, has allowed the integration of billions of transistors in a single chip, enabling the creation of different applications that were not possible in former times.

Entirely new concepts, like the Cyber-Physical System (CPS) and the Internet of Things (IoT), emerged some years ago, with the aim of bringing computing power to everywhere and everything. Distributed networks of smart devices are converging to

[1] This Chapter is based on our previously published papers [Jan+14], [Mor+16b] and [YLH17], with excerpts and some adaptations.

the so-called Ubiquitous Computing (UbiC), where the computational systems are so deeply integrated into the environment, that it should not be possible to distinguish between them anymore [Raj+10]. It is also predicted that all these computational tasks will be performed transparently to the users, in a context called Pervasive Computing (PvC).

To couple with the requirements of trends mentioned, to sense will be one of the essential features. In this context, the concept of Smart Sensors emerged, as a device which does not only acquire data but also to process these data and actuate in the environment. Figure 1.1 depicts the evolution from a typical sensor system (top) to a Smart Sensor (bottom), by transferring the processing part from the remote site to the embedded device.

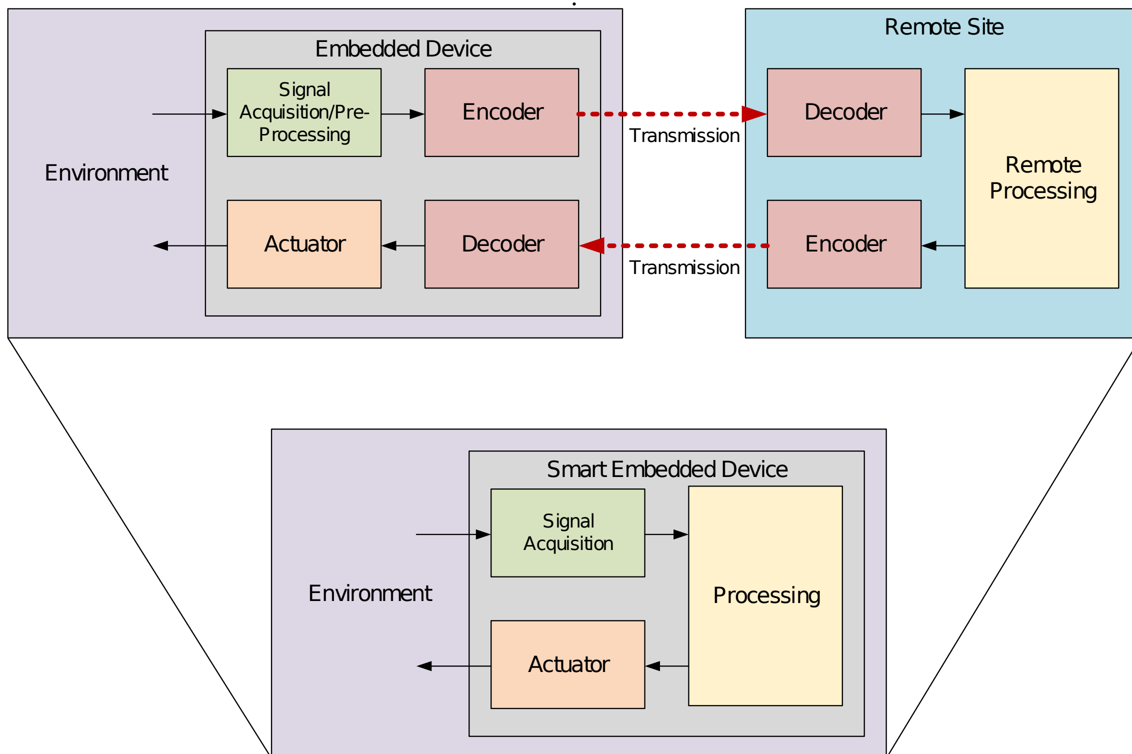


Figure 1.1: Top: Common sensor system with remote processing; Bottom: Smart Sensor system with local processing.

Several types of sensors might be used, and the information provided by each one must be merged for a better understanding of what is going on in the monitored environment [Gub+13]. It is predicted that particular devices able to acquire and process vast amounts of sensor data will be needed, and the current processing

architecture models are not able to deal with these requirements [Jan+14].

Figure 1.2 shows a conceptual model of such Multiple-Input/Multiple-Output (MIMO) heterogeneous processing architecture. The figure shows a multi/many-core processing architecture with spatially distributed inputs and outputs, as well as the mapping of three different applications. Each tile in the architecture would be defined as Input/Output, Memory, Processor or Co-Processor (Reconfigurable Fabric), allowing for different configurations, increasing the flexibility and the processing power.

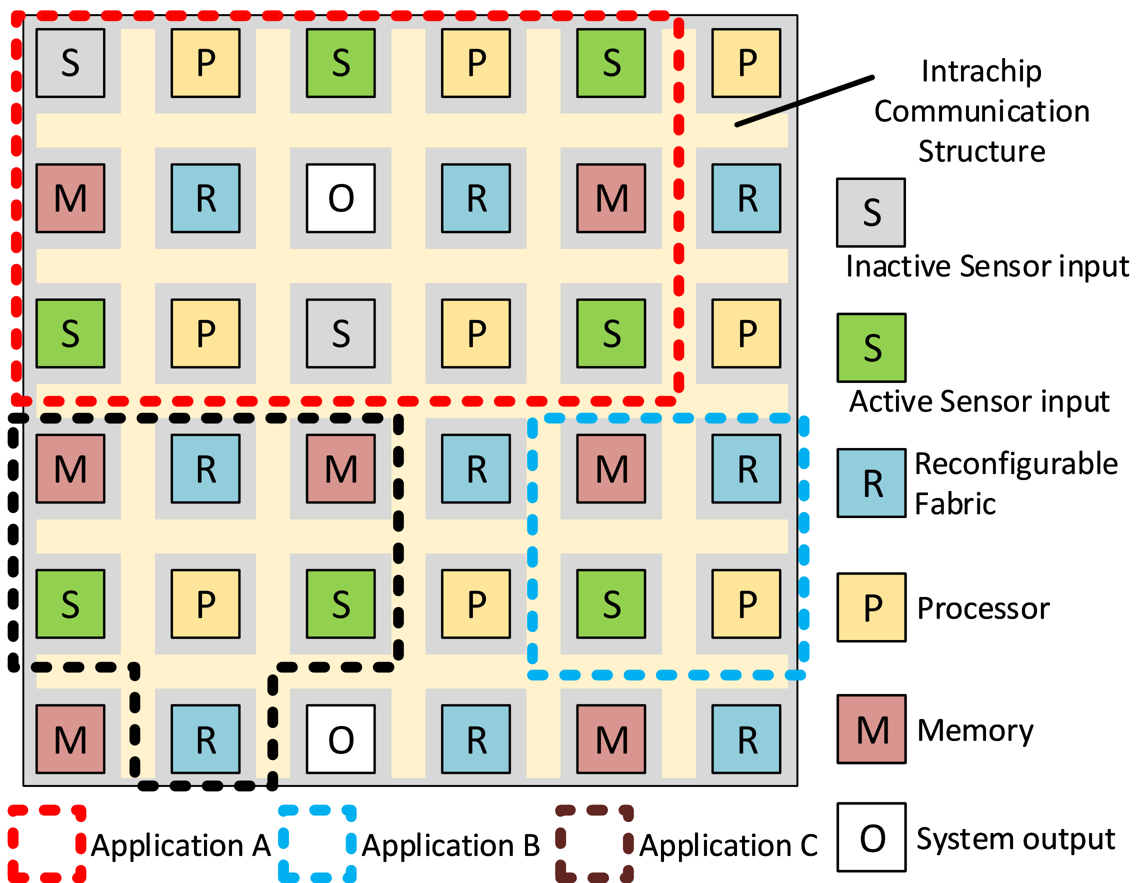


Figure 1.2: Conceptual model of a MIMO heterogeneous processing architecture [Jan+14].

1.2 Image Processing and Computer Vision

When we talk about monitoring an environment and its agents, we are talking about measuring their characteristics and states. For a human being, in general, the idea

of sense is directly related to our sensory functions: vision, hearing, touch, smell, and taste. Considering the amount of input data, the most complex of our sensory functions is our visual sense, composed not only by our eyes but also by our visual cortex (the part of our brain responsible for processing visual information acquired by our eyes). In comparison to other animals, our senses are not the most evolved. However, our capacity of processing the information acquired and make assumptions (to reason) is what differentiate us as the dominant species in Nature.

One of the most important technological challenges is to replicate in machines our reasoning abilities. The Artificial Intelligence (AI) area has evolved a lot in the past decades, following the development of the computers. However, there is still a considerable gap between what we can build on machines and what we can do in our brains. The brain's computational model is still unknown and today's best computers have only a small fraction of the brain's processing efficiency. Nevertheless, the computers can still be beneficial if correctly used.

Image Processing and Computer Vision (IP/CV) applications are present in several different areas as medicine, social networks, consumer electronics, industrial inspection, entertainment, security, safety, autonomous driving assistance, and so on. The quality of image acquisition devices has evolved a lot in the past decades, with today's resolution reaching dozens of Mega-pixels in high-end smartphones. In the context of the IoT, IP/CV applications will be in the mainstream, mainly due to the power of visual information.

A standard camera system is, in general, able to acquire, compress, store, and transmit images. However, for the new UbiC Era, these devices will act as Smart Sensors, with the ability to also perform sophisticated algorithms to extract information and actuate over the environment. This new set of cameras is also known as Smart Cameras and will play an essential role to capture and interpret the human behaviour in an environment, among other events/objects. Figure 1.3 shows an example of Smart Camera with a complete embedded application. These future cameras must be able to perform complex applications simultaneously, coping

with different requisites: energy consumption, chip temperature control, reliability, Quality of Service (QoS), data security, privacy, power management, cost (silicon area), and so on [Mor+16b].

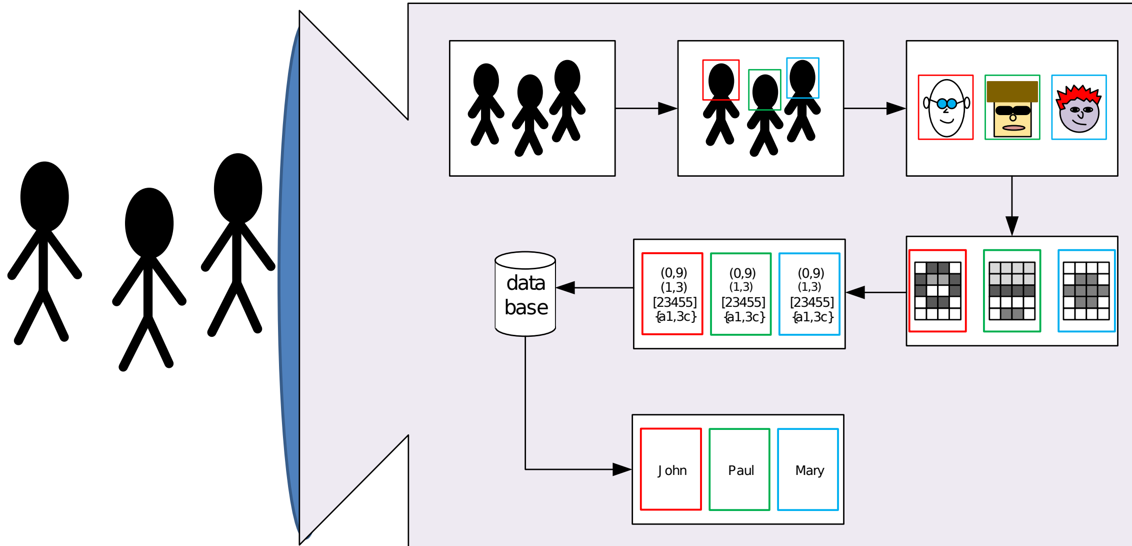


Figure 1.3: A Smart Camera with a complete multiple face recognition embedded application.

1.3 Real-Time IP/CV Systems

IP/CV applications are computationally costly, mainly due to the massive amount of data to be processed. New technologies are enabling the use of increasing resolutions, and new applications are demanding high-performance with tight deadlines. In this scenario, there are cameras with frame rates over one hundred thousand frames per second in high resolutions. When the application allows storing the images for later processing, the problem is then related to the amount of storage memory. On the other hand, several applications, e.g. autonomous vehicles, visual feedback control loops, high-speed robot manipulation, industrial inspection, and so on, depend on the results of IP/CV algorithms. In the context of these real-time applications, the tight timing constraints can lead them to fail when not attended. The success of Real-Time IP/CV processing systems is highly dependent on how efficiently the inherent parallelism is explored and, therefore, a particular hardware/software architecture must be used to fulfil the application's needs [KG06].

Simple processing architectures, such as the General-Purpose Processor (GPP), are not able to provide an energy efficient real-time performance for IP/CV applications. The sequential nature of a GPP does not offer many opportunities for parallelism exploration, which means that high operating frequencies are necessary, resulting in more power dissipation. In this context, there are single-core alternative architectures, such as the Very-Large Instruction Word (VLIW), which can overcome simple Reduced Instruction Set Computer (RISC) GPPs by exploring more efficiently the Instruction-Level Parallelism (ILP) for IP/CV applications [HWA15a]. However, the VLIW architectures do not scale well for a high number of issues, mainly due to the register file and multiplexing overhead, limiting the amount of processing parallelism [Vii+14]. Direct silicon implementations would provide the highest efficiency, considering power consumption, throughput, and area. However, implementations made of custom or standard cells are fixed and do not allow the flexibility to process a broad range of applications. Field-Programmable Gate Array (FPGA) implementations, however, have can also be good candidates for a semi-direct implementation. High-Level Synthesis (HLS) techniques can be used to create libraries of blocks specialised for different applications in the IP/CV domain [Mor+16a].

For several years, to enhance the processor's speed, the industry relayed on increasing the transistor count per area, and the circuit switching frequency. However, problems with the power density (related to the Dennard Scaling model) limited this trend [SBH14]. The industry entered then in the so-called "post-Dennardian Scaling Era" when it was forced into a transition to multi-core systems [Tay12]. On the other hand, this architectural change was already expected, due to the natural evolution of parallel computing area from multi-computer clusters to intra-chip multi-processors, which are envisioned to provide more scalable power efficiency when increasing the number of cores [Bor07]. To be able to handle all the constraints and also offer enough flexibility for different IP/CV applications, a solution for the Smart Cameras is also to migrate from single to multi/many-core processing

architectures [MKH15a].

1.4 Vision Processors

An in-depth analysis of the hardware characteristics is essential to overcome the limitations of Smart Camera devices, from image acquisition aspects to processing architectures features. From a historical perspective, the first commercial cameras used Charge-Coupled Devices (CCD), which dominated the market for some decades. Several technological aspects, such as manufacturing difficulties and power consumption, limited the use of Complementary Metal-Oxide-Semiconductor (CMOS) technology for image sensors. However, in the 1990's, the CMOS and Active Pixel Sensor (APS) (CMOS-APS) invention allowed for efficient integration of all image sensor electronic components in a single chip [Fos97]. The advances in the CMOS technology (mainly due to the processor's market needs), were also used by the camera sensors manufacturers, enabling significant cost reduction, and replacing the CCD cameras for the CMOS ones in most devices.

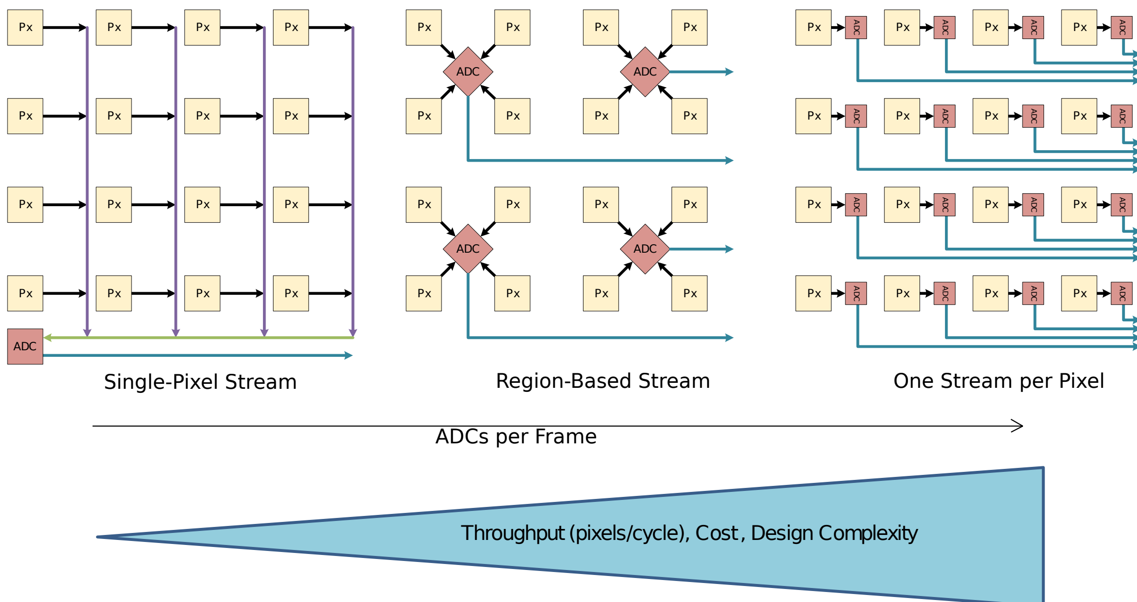


Figure 1.4: CMOS pixel array acquisition types: single and multiple pixel streams [Mor+16b], and the design trade-off related.

Figure 1.4 shows different sensor acquisition configurations [Mor+16b], which provide single or multiple pixel streams. Each configuration has its advantages and

disadvantages, implying a trade-off analysis by the system's designer:

- **Single-Pixel Stream**: For the complete sensor array there is only a single output channel (Analog-to-Digital Converter (ADC) + readout circuitry). Small throughput, lower cost, and simpler design complexity are the main characteristics of this scheme. It is also the commercially most common system.
- **Region-Based Stream**: The sensor array is divided into regions, and to each region is assigned one output channel. This configuration offers a design trade-off, where the main tuning parameter is the number of pixels per region.
- **One Stream per Pixel**: This configuration offers full acquisition parallelism by assigning a single output channel for each pixel. However, despite the higher throughput, this configuration has also the highest cost.

Real-Time IP/CV systems should start the processing phase as soon as the pixels are available. The acquisition architecture choice depends not only on the speed and parallelism but is also based on the IP/CV algorithms to be executed. A more quantitative analysis of the acquisition schemes is provided in Section 5.1.

Two similar architectural concepts, Focal-Plane Image Processing (FPIP) and Near-Sensor Image Processing (NSIP), have emerged in the late 1980's and early 1990's, as a solution for high-speed acquisition and processing in Real-Time IP/CV systems. The basic idea was to merge the pixel sensors with processing elements in the camera's focal-plane [Fos89], or as close as possible one to the other [FA94a], with the goal of offering pixel access parallelism and low latency.

Several architectures from the literature use this concept. Most of them were implemented completely with analog circuits (both acquisition and processing) [FA94a] [EKC01b] [Elo+04b]. The analogue implementations have advantages over digital ones, such as power consumption and speed. However, they also have a lack of programmability, and their implementation complexity makes them hardly adopted in IP/CV applications. Also, due to physical limitations, the scale of integration

achieved better results for digital circuits than for analogue ones [EYF99]. Digital technology offers the advantages of flexibility through programmability, the reuse of standard cells and more scalability, in comparison with the analogue circuits. Due to these aspects, modern implementations unify analog and digital circuits as follows: analog ones for the acquisition and ADC, and digital ones for the processing part [KII97a] [FZR08] [WD12] [Sch+16].

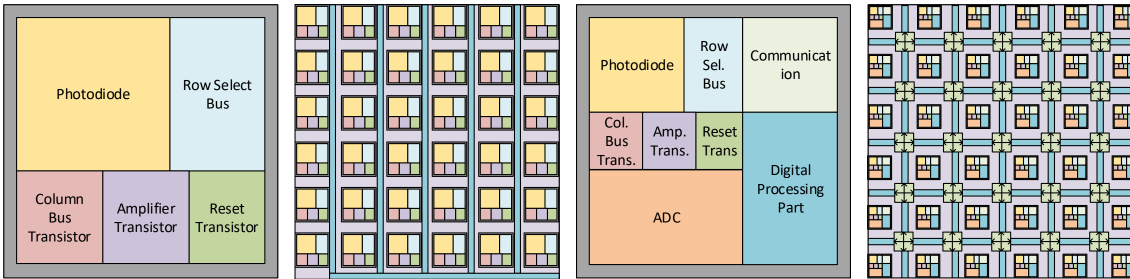


Figure 1.5: Fill-factor reduction when adding more functionalities to the image sensor [Mor+16b].

In Figure 1.5, we can see the structure of a single standard pixel. As can be seen in the picture, the addition of a ADC and a Processing Element (PE) to the pixel area would reduce the fill-factor considerably, and by consequence, the image quality would be degraded. Besides, due to the limited area available, the PEs found in the literature are mostly analogue filters and/or small digital ones and do not offer too much flexibility. Figure 1.5 shows issues related to the communication structure which must be present to integrate the PEs. This structure would contribute to reduce the sensor's fill-factor considerably.

1.5 Multi/Many-Core Vision Processors

In Chapter 3, the historical evolution of the Vision Processors is traced, the technological evolution is detailed, and the state-of-art is determined. The main technological trends in the design of IP/CV systems is shown in Figure 1.6:

- **Parallelisation**: this is the main direction to deal with the massive amount of data to be processed in IP/CV applications. The intrinsic parallelism present in such application favours this trend.

- **Integration**: the need for portability and flexibility lead to the design of System-on-Chips (SoCs) with the Pixel Sensor, and the Processing Unit integrated to reduce the size, power consumption and data transport delays.

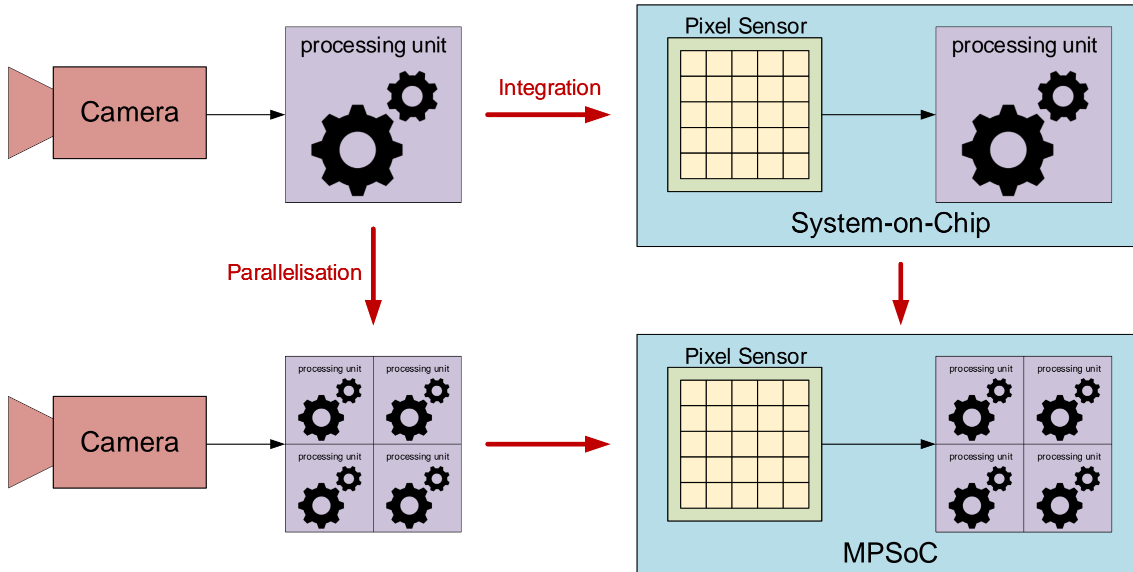


Figure 1.6: Technological trends in the design of IP/CV systems: Parallelisation and Integration.

Over the years, analogue/digital integration achieved success. However, most implementations are quite simple from the processing architecture point of view. Most architectures were more focused on showing the feasibility of the concept through prototyping, without exploring the vast design space available for both acquisition and processing architectures.

Figure 1.7 shows the concept we explore in this project. The sensor array has spatially distributed pairs amplifier/ADC, and each pair is responsible for a region of the image. Using the Through-Silicon Vias (TSV) technology, the outputs of the ADCs are sent to an underlying processing layer. The processing layer is a many-core architecture composed of distributed pixel registers which receive the pixel stream from the ADCs. A PE is responsible for processing each region, and a communication infrastructure allows for data exchange among them. With this configuration, both the acquisition and the processing parts can explore a higher amount of parallelism.

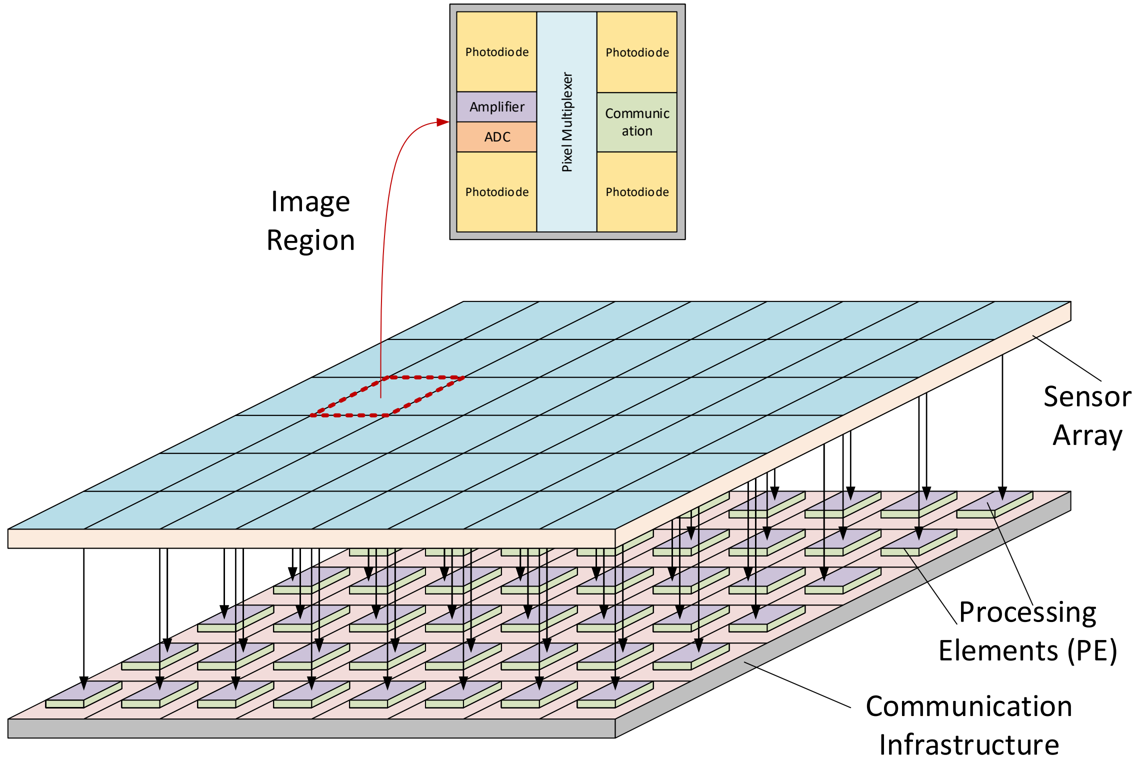


Figure 1.7: *Focal-Plane/Near-Sensor* Image Processing concept [Fos98] [MH14] [MKH15a].

The design of such architecture is not trivial since the PEs, and the communication structure must be developed with the focus in the application and the embedded hardware constraints (silicon area, power consumption, thermal distribution and so on). Also, software related issues must be solved. The programming model must be able to explore the parallelism in the applications, considering the spatial distribution of PEs, the distributed input streams, and the synchronisation and data exchange issues.

In this way, the PEs from Figure 1.7 would be complex *Processing Tiles*, able to communicate with them, and process data accordingly to the desired application. The design of such architectures must be oriented to the specific needs of the IP/CV applications like in an Application-Specific Instruction set Processor (ASIP) [MK16] [Eus+14]. In the context of embedded parallel processing, we propose a Multi-ASIP architecture as a solution for future Smart Cameras [MKH15a] [Mor+16b].

The proposed Multi-ASIP solution will be hence called a Many-Core Vision Processor architecture, tightly integrated with the pixel sensor array. The design space

of such architecture encompasses several different aspects in both analogue and digital domains. The focus of this work is in the digital hardware architecture only, considering CMOS sensors with different acquisition configurations. There is a lack of design methodologies able to bring together both the application-specific domain and the hardware/software co-design problem [KT11]. Multiple parallelism levels must be analysed, to provide precise and detailed information to the architecture designer. The challenges in the development of such many-core vision processors are not only in the hardware itself but also in the efficient algorithm implementation and mapping. The complexity envisioned requires an integrated hardware/software analysis able to determine several features, such as the number of tiles, pixel distribution, communication structure, processing needs, programming model, memory organisation and so on [MH14].

In this work, we provide a Design Space Exploration (DSE) of the digital hardware solutions to design and program what we envision as the next generation of vision processors. Table 1.1 shows a summary of the main DSE parameters explored in this work.

Table 1.1: Summary of the main DSE parameters explored in this work.

Acquisition Scheme (pixels per ADC)	Processing Element	Communication Structure	Application Domain
Single-Pixel Stream Region-Based Stream One Stream per Pixel	GPP (RISC) VLIW ASIP FPGA	Point-to-Point (P2P) Bus-based Network-on-Chip (NoC)	Programmability Throughput Parallelism

Chapter 2

Thesis Description

In this chapter, we show the description of the thesis project, starting with an overview of the methodology used for this research. After that, we provide a general description of each abstraction level used and the approach for each one. Then we illustrate the main contributions of this research and how this document is organised.

2.1 Overview of the methodology

The goal of this research is to determine the hardware/software architecture of future vision processors. As highlighted in Chapter 1, there is a trend for embedded systems powered by Chip Multi-Processor (CMP), and we suggest that vision processors shall benefit from multi/many-core architectures.

A more specific trend is the exploration of concepts like Focal-Plane Image Processing (FPIP) and Near-Sensor Image Processing (NSIP), to reduce the image acquisition delay and provide more parallelism to the processing system. This trend is vastly explored in the literature, divided into pure-analogue processing, mixed analogue-digital processing, and pure-digital processing.

The design space of CMPs is quite large, and it is not practical to have physical implementations of all design possibilities. Besides, considering the application-specific nature of the current research, more focused effort should be made in exploring solutions which better match the application's needs. We provide in this

thesis a holistic approach: a multi-level integrated design-space exploration.

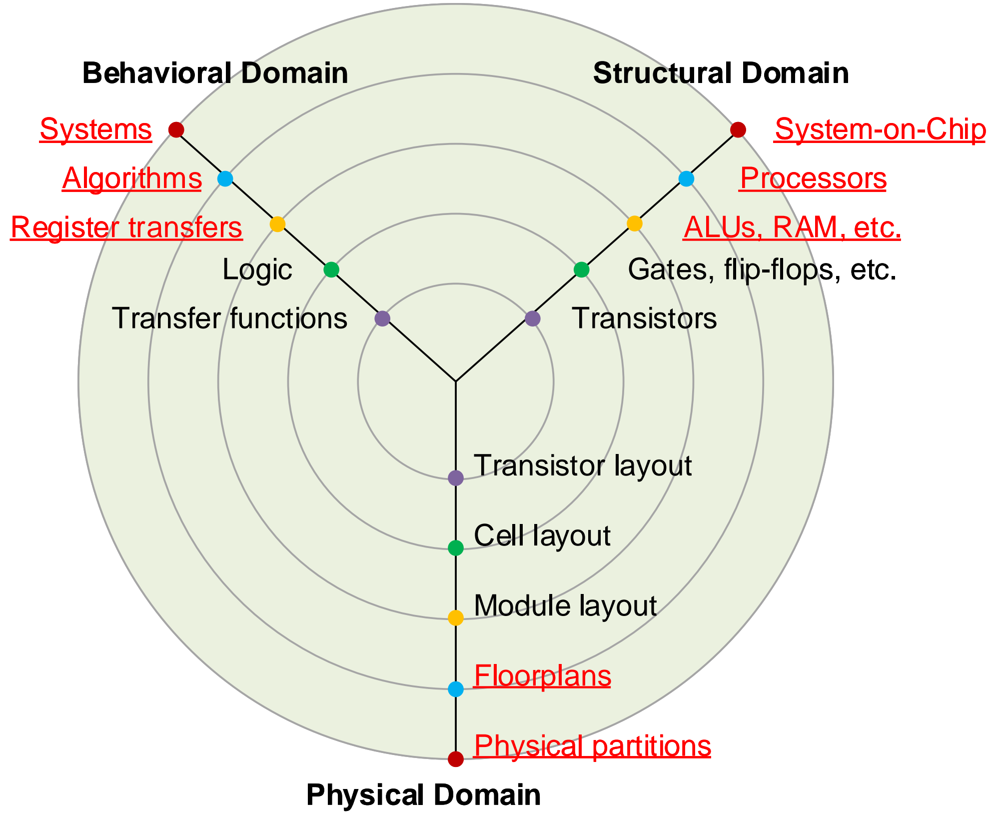


Figure 2.1: A Y-chart showing the levels of an electronics design, with the parts covered in this work highlighted [Gaj+09].

In Figure 2.1 we can see the classical *Y-Chart*, also known as *Gajski-Kuhn* chart, which shows the levels of an electronics design [Gaj+09]. In this work, we study the parts/levels highlighted in red. We followed a design flow known as Specify, Explore-and-Refine (SER), which starts at the System Level and with specifications about the application needs [Gaj+09]. In this flow, iteratively, we analyse and refine the design possibilities trying to constrain the design space. When no more refinements are worth doing at this level, we create specifications at a lower level using the constrained design as input. This process is repeated until the design reaches the desired level and/or acceptable specifications.

2.2 High-Level approach

The first approach is more in the application domain, trying to see the problem from the user perspective: how an Image Processing and Computer Vision (IP/CV) expert would like to write programs to our architecture? We use the ideas discussed in the Part II of this thesis to develop some new concepts. At this level, we explore algorithm analysis techniques based on Task-Graphs (TGs) and develop a multi-agent simulator which emulates a many-core vision processor. As a result, we select some design possibilities for further analysis.

2.3 Intermediate-Level approach

The second approach comes as an evolution from the previous one, going to a lower level of abstraction. At this level, we have already constrained the design-space, and we analyse a smaller set of hardware/software design possibilities. In this approach, we evolved the previous simulator to a more detailed hardware architecture simulator. Also, we estimated the power consumption, the performance, and the silicon area of each design choice, considering only the digital processing architectures. The results allowed us to constrain even more the design space, leading to the next level.

2.4 Low-Level approach

The third approach is at the Register Transfer Level (RTL) level, where we describe some hardware possibilities in Very-High Scale of Integration Chip Hardware Description Language (VHDL), focusing on obtaining cycle-accurate simulations and hardware synthesis estimations. The result of this approach is a parameterised hardware/software many-core architecture which can be configured accordingly to the specific IP/CV algorithms needed by the application. Besides, we discuss the design of the Processing Elements (PEs) using two approaches: a High-Level Synthesis (HLS) method and an Application-Specific Instruction set Processor (ASIP)

design.

2.5 Thesis Contributions

The main contributions of this thesis are:

- A holistic analysis of the design space for future vision processors. This analysis considers not only the hardware/software architecture, but also how we can optimise the end-user benefits of such system.
- A multi-level design framework composed by different tools integrated to handle both the application intrinsics and the hardware/software design space.
- The proposal and implementation of a parameterizable many-core vision processor architecture and its corresponding tool-flow.

2.6 Document Organization

The rest of this document is organized as follows:

- A preliminary part with the knowledge background for this thesis:
    1. Bibliographic review about real-time IP/CV System-on-Chip (SoC)s (Chapter 3).
    2. Basic IP/CV concepts used in this thesis (Chapter 4).
    3. An analysis of the application/software/hardware co-design details and definitions (Chapter 5).
- The high-level approach, divided in:
    1. An overview of the approach (Chapter 6).
    2. The Task-Graph Generator tool (Chapter 7).
    3. The Task-Graph Clustering method (Chapter 8).

4. The Task-Graph Simulator tool (Chapter 9).

- The intermediate-level approach, composed by:

  1. Description of the approach (Chapter 11).
  2. Detailed Multi-Processor System-on-Chip (MPSoC) simulator description (Chapter 12).
  3. Simulation results and performance estimation (Chapter 13).

- The low-level approach:

  1. Standard tiled architecture description (Chapter 14).
  2. Description of two approaches to develop PEs (Chapter 15).

- Conclusions and suggestions for further research (Chapter 16)

Part II

Preliminary Study

Chapter 3

Related Work

In this chapter, we make a review of some relevant bibliographic references related to this work. Each section is focused on a different aspect of the system's design and implementation. We start with works more related to the pixel sensor design and analogue implementation of Image Processing and Computer Vision (IP/CV) algorithms. This is followed by approaches with mixed-signal devices: analogue acquisition with minor processing functions integrated to a digital processing part. The subsequent section reviews techniques for processor architecture design, which we can use to define the Processing Element (PE) of our many-core system. Afterwards, there is a review of multi/many-core architectures and methodologies to their design. The last section is devoted to discussion and remarks about the literature, establishing the guidelines for the development of this work. [2]

3.1 Vision Processors

[AFI93] and [FA94b] present a first work showing an implementation of hybrid integrated analogue/digital circuit, with a sensor and low-level algorithms as median and convolution filters. In these works, the authors implement a *Sensor Processing Element* (SPE) in a fully parallel way (one SPE per pixel). Each SPE has a

[2] This chapter is based on our previously published papers [MH14], [Mor+16b], [MK16], [MH16], [Mor+16a] and [YLH17], with excerpts and some adaptations.

photodiode, an ADC, a bit-serial ALU, and a small memory. In that work, the authors propose the concept of *Near-Sensor Image Processing* (NSIP), as a SIMD array of SPEs. Pixel-level operations, like *Adaptive Thresholding*, *Median Filtering*, *Histogramming*, *Grayscale Morphology*, and *Convolution* were implemented. Besides, a simple feature extraction technique, based on geometric moments was also implemented, which allows for object recognition.

[ESÅ96] extended the previous approach with a *Global Logic Unit*, which is responsible for operations involving the complete image. Additionally, this work shows an implementation with image processing capacity for every single pixel.

Vision chips have been studied for more than thirty years. In work from the year 1997, [Moi97], there are shown more than fifty different vision chips, most of them analogue and with no programmable possibility. Several design blocks (with the technology of that period) are depicted, as well as the primary design considerations of each chip.

In [Fos97], the author provides data about the divergence among the technology feature size and the pixel size, from 1970 to 2005. A relevant data published in this work was the practical optical limit of the pixel size, estimated at 5 μm. This limitation would predictably constrain the image resolution, avoiding the spread of this technology. However, to prevent this limitation, current manufacturers use micro-lenses, allowing for reducing the fill-factor while maintaining image quality. With the micro-lenses, it is possible to integrate more functionality into the processing part, making use of the reductions in technology feature size. Also, 3D stacked chips, making use of Through-Silicon Vias (TSV), are expected to be cost-effective in some years. Several discussions regarding pixel-level processing challenges are presented in [Abb99], together with future trends in this area.

The design of a vision chip architecture was implemented into an FPGA, with each processing element connected directly to photo-detectors in [KII97b]. With a sampling rate of 1ms, the system could be used in a robot control system, enabling high-speed visual feedback. This high-speed vision system used General-Purpose

processing elements. This is one of the first works reportedly using digital processing elements, instead of analogue ones. A full-parallelism is explored, with each PE responsible for a single pixel. In this architecture, each PE is connected to its four neighbours (North, South, East, West) by point-to-point connections. The PEs are composed of local memory, an ALU, and the I/O interfaces. There is a global instruction memory, and the instructions are transmitted to all PEs in parallel. This organisation allows for automatic processing synchronism among the PEs. A similar application is also shown in [Na00].

The architecture presented in [LD06] and [LD08] is a digital vision chip focused on the efficient implementation of global image processing algorithms, instead of point and neighbourhood algorithms. The architecture is composed of an array of PEs in a point-to-point 4-connected mesh. Each PE has a single pixel, a simple ADC, an ALU, and registers, as well as a communication unit. The instructions are provided by a global controller, and a particular block in each PE is responsible for long communications, enhancing the performance of global operations. One of the advantages of this architecture is the scalability provided by the use of a single type of PE, without extra row/global processors.

The works from [EKC01a], [Lin+08] and [Mia+08a] show how to better explore parallelism in architectures of image processor chips. The work of [Mia+08b] shows the implementation of an architecture for real-time machine vision applications. This system is composed of a hybrid architecture, which combines a SIMD PE array with row-parallel processors (Figure 3.1).

The authors in [Mia+08b] focused on low-level and medium-level image processing algorithms, mainly for mathematical morphology methods. An I/O interface is responsible for sending extracted features to external systems for processing/storage/ visualisation. The architecture is composed of a mesh of PEs, and at the end of each column and each row, there are XPUs and YPUs (X and Y Processing Units), respectively. Each PE contains a single pair Pixel-ADC, a small memory, and can perform boolean operations (which are the core of mathematical morphol-

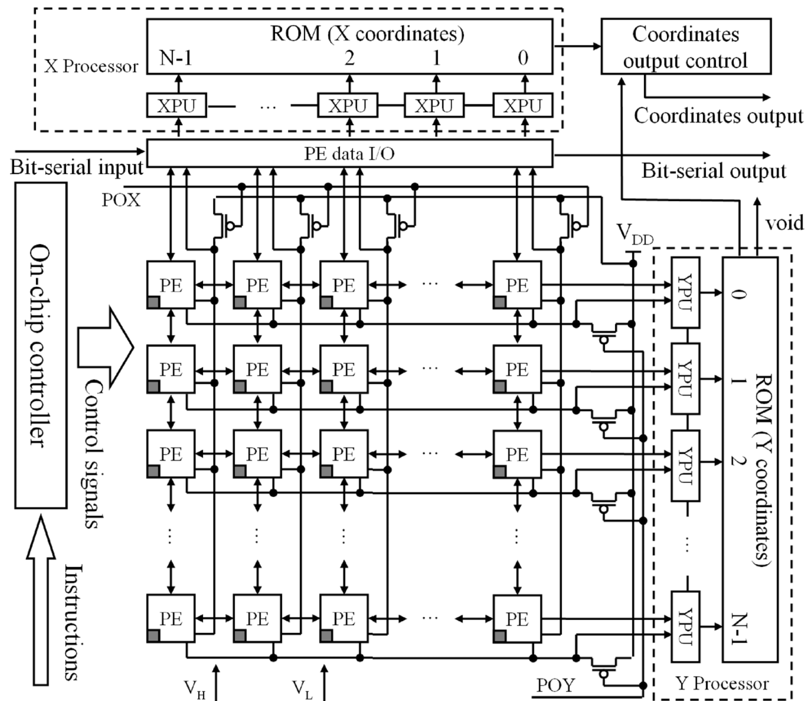


Figure 3.1: Architecture of the SRVC (SIMD Real-time Vision Chip) [Mia+08a].

ogy). To allow for operations in sequences of images (or an image processing chain composed of a sequence of operations), the local PE memory can store pixels for several images, with the same X-Y coordinates. There are two-bit structures, X-Processor and Y-Processor, which group the XPUs and YPUs. The X/Y-PUs are responsible for acquiring the outputs of the PEs, as well as its coordinates. The X/Y-Processors operate determining the connected pixels which form objects, to extract image features. The communication between the PEs is point-to-point, and the synchronisation is controlled by a global instruction memory.

In the search for optimisation, some authors use biologically-inspired approaches, emulating how animal's and insect's eyes work [Elo+04a] [FZR08]. In [FZR08], the author innovates the design of standalone vision systems by stacking different layers (sensor array, frame buffers, processing elements) in a 3D chip.

The approach in [FZR08] provides advantages in the image quality (an increase of the fill factor), speed, and complexity of the processing part. The processing elements are organised as a SIMD system, where each tile is responsible for a region of the image, but all tiles operate synchronously with the global instruction memory. A highlighted characteristic of this architecture is that the pixels are mapped to the

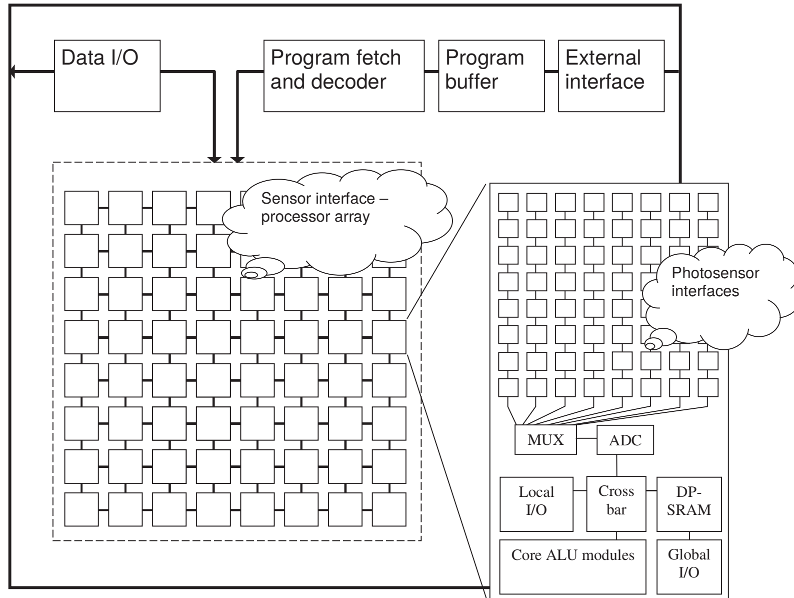


Figure 3.2: The SIMD multilayer chip from [FZR08].

local tile memory, which manages pixel requests to other tiles, resulting in a memory structure transparent to the processing element. The proposed architecture achieved a performance of 2 ns per pixel, for a 3×3 convolution. Besides convolution, other image processing algorithms were tested: grey-scale morphology, contour detection, and diffusion.

The work in [Zar+11] presents the VISCUBE chip. In this approach, the processing is distributed in both analogue and digital parts: early image processing in the mixed-signal processor array; foveal processing in the digital processor.

Following this 3D idea, the architecture proposed by [ZFW11] is composed of multiple levels of parallel processors. The first level is a microcontroller, and the second one contains row processors and the third an array of Processing Elements (PEs). This architecture provides different levels of parallelism exploration, which can cover a significant amount of applications. The mapping of algorithms to the architecture is dependent only upon the parallelism identified. The authors classified the IP/CV algorithms in three sets, accordingly to the type of operations:

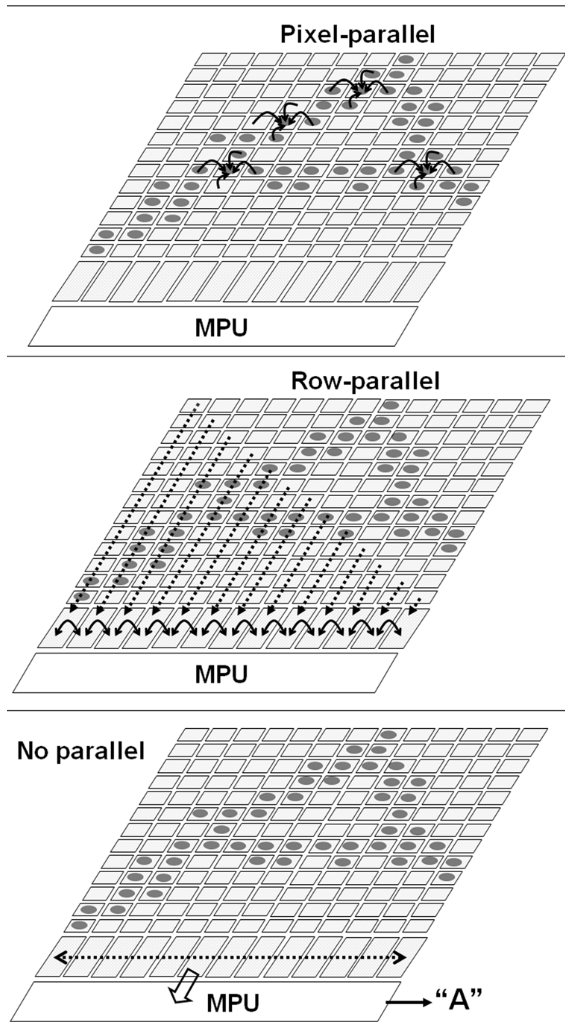


Figure 3.3: Type of operations [ZFW11].

- *Pixel-Parallel*: these operations are implemented in the PE array, corresponding to low-level algorithms, with low complexity, e.g. convolution, morphology, and thresholding.
- *Row-Parallel*: medium-level algorithms, like FFT, DCT and statistical operations, are implemented in the RPs.
- *No-Parallel*: implemented in the MPU, these operations are from high-level algorithms, like object tracking and pattern recognition.

In the architecture of [ZFW11] (Figure 3.4), the Analog-to-Digital Converters (ADCs) are allocated per rows, controlled by the MPU. However, pixel data are sent from each ADC to the corresponding RP memory. An RP shares its memory with a row of PEs, allowing for fast data exchange among them. The RPs are connected to neighbour RPs through point-to-point communication, and the PE mesh is integrated in a 4-connected way, also with point-to-point communication. This organisation provides different levels of processing, as well as different communication patterns. Edge Detection, Motion Detection, Filtering, DCT, Character Recognition, and Target Tracking are among the applications implemented in the resulting architecture.

Figure 3.5 shows the ASPA Sensor-Processor Chip ([LD11]), which has, in each PE, sensor/ADC, a memory, communication unit, and a small mixed analog/digital

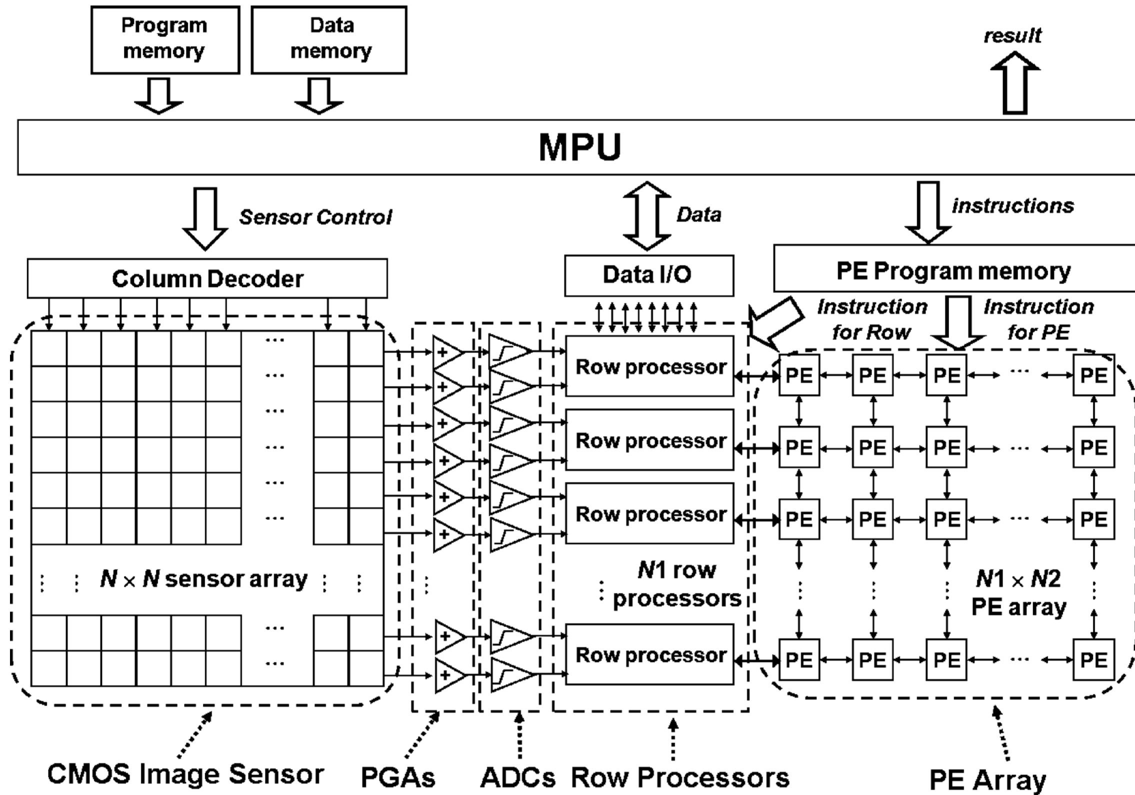


Figure 3.4: Schematic of the [ZFW11] chip architecture: image sensor array, A/D converters, PE array, RP array, and MPU.

processor. This architecture was developed with only a single pixel per PE, which allowed for an extreme parallelism exploration. The applications implemented in the ASPA chip were low-level algorithms, and the programming was performed in assembly code, focused on SIMD exploration.

The SCAMP-3 vision chip (Figure 3.6) is a programmable pixel-parallel processor array which operates in a single-instruction-multiple-data mode [PDu11]. It is a mixed processor, with a digital control part with and analogue data-path. This processor can implement mainly low-level IP/CV algorithms, like the convolution. A 2D-Mesh topology was considered for the interconnections and the central aspect identified was how the amount of local memory (register file, in the case) affects the processing speed.

To the best of our knowledge, the latest work developing a vision processor integrating the pixel sensor array and processing units is shown in Figure 3.7 [Sch+16]. In this work, the acquisition is performed using one ADC per group of 8×8 pix-

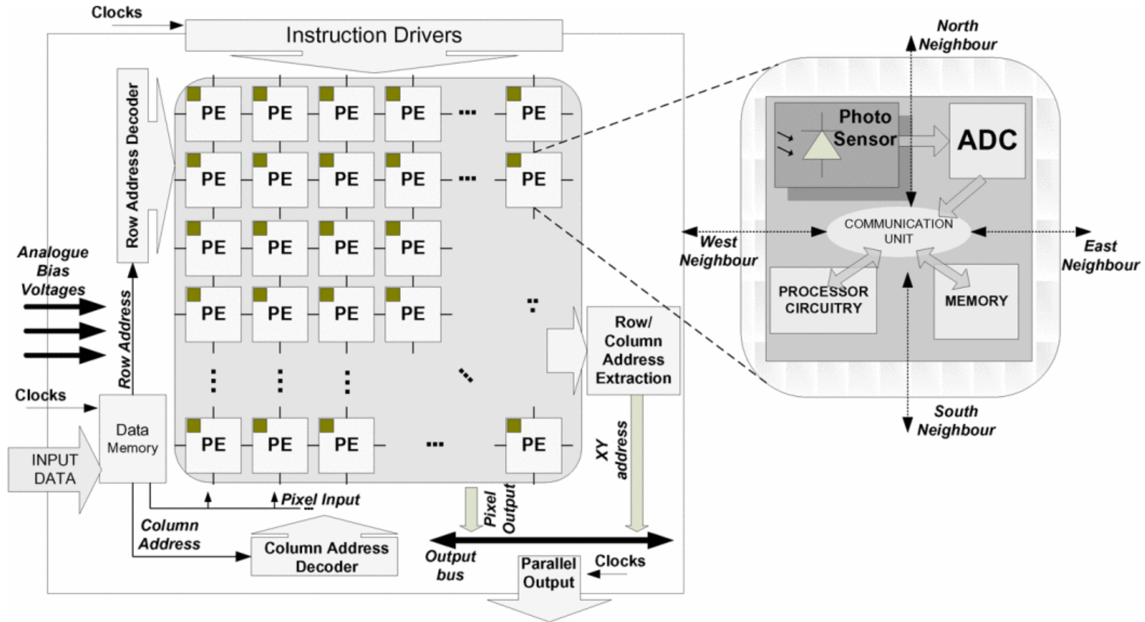


Figure 3.5: The ASPA chip architecture [LD11].

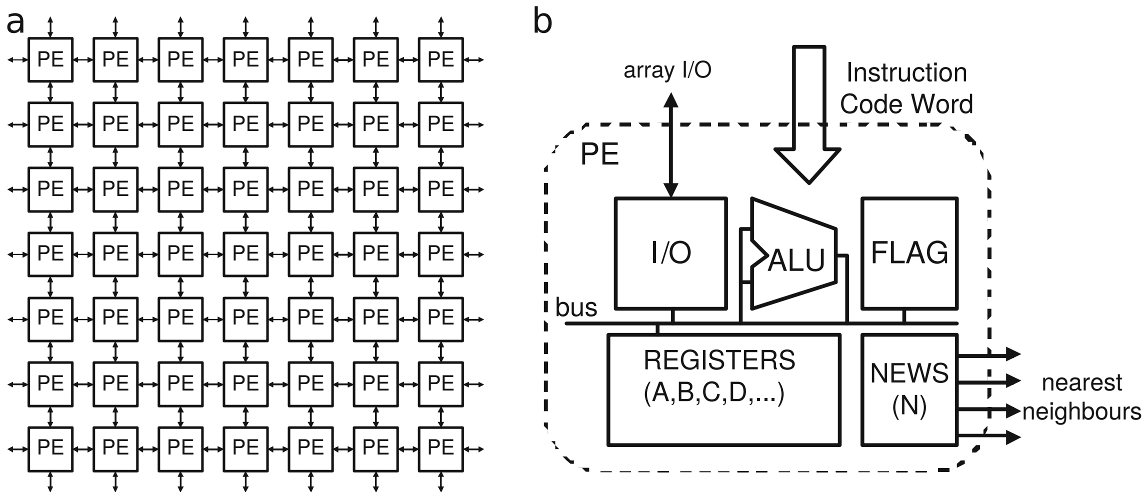


Figure 3.6: The SCAMP3 chip architecture [PDu11].

els. To each group is assigned an 8-bit ADC processor, which communicates to the closest neighbours (North, South, East, West) with Point-to-Point (P2P) links. There is a global program memory, and the instructions are propagated through the processing array using the P2P links. A broad range of IP/CV applications can be implemented, e.g., 8×8 DCT, Edge Detection, Histogram, and Tracking.

Table 3.1 shows a summary of the Vision Processors discussed in this section. As can be seen, more recent works present dominance of the Region-Based acquisition type, digital processing, and SIMD organisation. Most of the works use P2P-based communication among the PEs. In some of them, there is also one bus for a complete

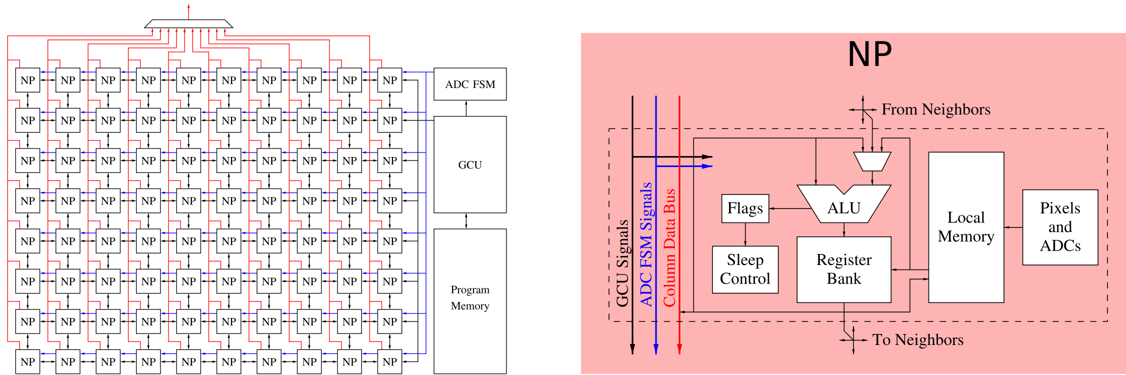


Figure 3.7: The Neighborhood Processor Array [Sch+16].

row, to allow for fast communication.

Table 3.1: Summary of most relevant related Vision Processors.

Reference	Acquisition Type	Analog Processing	Digital Processing	Processing Type
[AFI93]	One per pixel	yes	yes	SIMD
[FA94b]	One per pixel	yes	yes	SIMD
[ESÅ96]	One per pixel	yes	yes	SIMD + Global Unit
[Fos97]	Region-Based	yes	no	SIMD
[KII97b]	One per pixel	no	yes	SIMD
[Na00]	Region-Based	no	yes	SIMD
[Mia+08b]	One per pixel	no	yes	Array + Row/Column Units
[FZR08]	Region-Based	no	yes	SIMT
[Zar+11]	Region-Based	yes	yes	SIMD
[ZFW11]	One per pixel	no	yes	SIMD
[PDu11]	One per pixel	yes	yes	SIMD
[Sch+16]	Region-Based	no	yes	SIMD

3.2 Processor Architecture Design

The analysis of the selected works in the last section allows us to identify some characteristics of the PEs used there. Most of them are merely programmable Finite State Machines (FSMs) specifically designed to deal with the low-level operations required by the IP/CV algorithms: get/set pixel values and arithmetic operations. The most used Processing Type is the Single-Instruction Multiple-Data (SIMD), where the PEs share the instruction memory, and work with the synchronization

made per instruction. In general, General-Purpose Processors (GPPs) have a good average performance among different application domains; however, for some specific applications, they struggle to be efficient. Application-Specific Instruction set Processors (ASIPs) are specially adapted/designed to handle the most important characteristics of a given application domain. The design of such architectures requires knowledge about both the application domain and hardware/software co-design. This section shows a review of methods and systems developed to help in the design of ASIPs.

There are several works in the literature regarding ASIP design methodologies and embedded real-time IP/CV processing platforms. In [SML07], a methodology for the development of ASIPs based on LISA is described in details. The work of [EWL13] shows a new method for fast ASIP design, which uses abstract processor models to minimise the time of development. The basic methodology in this work uses application profiling to extract the relevant features.

[Dia+12] uses profiling methods to the design of efficient computer vision algorithms for robotics applications in space. [FH14] proposes a methodology based on Peephole optimisation to identify, in the assembly code, the most repeated sequences of instructions, considered as the best candidates for optimisation. This work shows the design trade-off (area, power, and timing) for an Instruction Set Extension, applied to Sequence Alignment problem. This methodology was followed in the current work to propose the design of new instructions.

In [Mus+10], DSP-based implementation and comparison of several different algorithms for edge detection are performed. Several different optimisation techniques were used, like Cache Optimization, Compiler Intrinsics, and Software Pipelining. No concerns about the image acquisition stage are present in this work; however, the Compiler Intrinsics concept was used to ease the creation of custom instructions. [MSL12] shows an FPGA implementation of image filtering by convolution, with camera synchronisation. In [MLB12], redundant operations are identified and eliminated, resulting in more efficient resource use.

In [MSL12], the authors show similar FPGA-based designs, not only for convolution but other neighbourhood operations, such as rank-order filters and binary morphology. The architecture shown in these works can efficiently store only a region of the image per time, reducing the memory usage in the acquisition stage. The current work was developed based on the standard architecture shown in [MLB12] [MSL12].

A reconfigurable ASIP for image processing is described in [Lia+12]. Some approaches for the design of application-specific processors are shown in several works. A hardware/software partitioning technique, together with a resource management analysis can be seen in [GBC09]. This system is implemented in a run-time reconfigurable system, with great flexibility.

One of the steps in the proposed methodology is the selection of processors, or the development of new custom ASIPs, to be used as PEs. In [T+06], [TG08b], and [TG08a], it can be seen a technique to generate the structure of a custom processor by analyzing the application's C-code. The CoEx methodology, proposed by [EWL15], is based on profiling an application in the LLVM Intermediate Representation ([LA04a]) and, with the help of processor's abstract templates, suggesting a microarchitecture. In [MKH15b], an ASIP for low-level image processing operations is shown as a result of an instruction's sequences optimisation.

Table 3.2: ASIP design methodologies/tools from the literature review.

Reference	Methodology	Automatic Architecture Generation	ADL	Simulation	RTL
[T+06]	Task-Scheduling	yes	-	-	no
[SML07]	Profiling	no	LISA	Instruction/Cycle-Accurate	yes
[Mus+10]	Compiler Intrinsics	no	-	Hardware Implementation	no
[RAS11]	Profiling	no	SystemC-based	Instruction-Accurate	no
[Dia+12]	Profiling	no	-	Cycle-Accurate	no
[Lia+12]	Task-Scheduling	yes	LISA	Instruction/Cycle-Accurate	yes
[EWL13]	Profiling	yes	LISA	Instruction/Cycle-Accurate	yes
[FH14]	Peephole Optimization	no	LISA	Instruction/Cycle-Accurate	yes
[Jor+17]	Probabilistic Analysis	yes	no	-	no

Table 3.2 shows a summary of the main ASIP design methodologies/tools extracted from the literature. By taking advantage of: the profiling techniques [Dia+12] [FH14]; the design methodology [SML07] [EWL13]; the standard convolution archi-

tecture [MLB12] [MSL12]; and the compiler optimizations [Mus+10] from the literature, it was possible to design an an efficient hardware/software structure, turning the standard RISC processor into an efficient ASIP. In Section 15.2.3, the design process is described in more details.

3.3 High-Level Synthesis methods

Another option to design the PEs is the High-Level Synthesis (HLS), which takes an algorithm description in a high-level programming language and generates an Register Transfer Level (RTL) description. The EDA industry emerged and grew to seek for efficient ways to transform the idea of an application into a hardware processing system. Since the first CAD tools used to help the design of analogue parts, until the current complex commercial HLS tools, the primary goal of an EDA tool is to make the application's implementation more friendly to the designer.

In [GR94], an overview of the nomenclature of HLS area and the central concepts related to it are presented. The work in [WO00] shows a tool flow for ASIC design based on C input, to be used for high-level synthesis and verification. The presented tool allows the user to actuate in several different levels, with suggestions for modifications given by the tool. They also report an industry case where it was possible to achieve a reduction in the project duration in about six times. In our work, we also take advantage of the C-based input to verify the created hardware.

The *Spark* framework is presented in [Gup+04]. It is an HLS tool which relies on source code transformations to enable better synthesis results. This tool has the advantage of minimal user interaction, which can lead to faster results. The authors of [Per09] highlight that Model-Based Design, used in tools like Mathworks Simulink, requires from the designer the knowledge about both application and hardware architectures, which results in longer design cycles and also non-portable results. The work suggests the application's implementation using abstract functional units, like multipliers, adders, memories, and so on, generating a data-flow graph. After that, a set of transformations is performed to identify patterns and structures, which are

then mapped to the FPGA resources. Our method works similarly, but with the advantage of having a higher abstraction level with the C code input.

In [MHS10], an HLS framework was developed to map DSP algorithms to Coarse Grain Reconfigurable Architectures (CGRAs). The algorithm's description is done in a subset of the C language, due to restrictions on handling dynamic data structures and functions. The designer must use pragmas to identify SIMD-prone blocks of code, like in OpenCL [Cza+12]. A Control-Data-Flow-Graph is then extracted, and a configware is generated to the CGRA. The main disadvantage of this approach is that the designer must know which type of structures should be highlighted by the pragmas, what means that the success of implementation depends highly on the user's experience.

The ROCCC (Riverside Optimizing Compiler for Configurable Circuits) is a tool to create hardware accelerators from applications described in the C language [Vil+10]. It presents a 15x reduction in the design time, in comparison to hand-written VHDL code. However, the flexibility of application's description is constrained by the necessity of using specific templates and patterns to enable the tool to extract the hardware structures from the C code correctly. The integration among domain-specific design platforms and HLS tools is discussed in [Con+11]. A review of the state-of-art tools is shown, highlighting the trend to have higher abstraction levels, going from the programming language level to the application domain level. This trend means that these tools converge to particular platforms offering to the designer application-specific design options, which can improve the quality and reduce the design time for such applications.

The *LegUp* ([Can+13],[For+14]) framework is an open source HLS tool focused on FPGA technology. The tool receives an application described in standard C language and automatically generates a hardware/software architecture composed of a RISC soft-processor and custom hardware accelerators, connected using a bus interface. The key point of this approach is to offer the flexibility of a RISC processor together with the hardware accelerators. [Hua+15] uses the *LegUp* framework

to analyse the effect of different compiler optimisation options on the quality of the synthesised hardware, showing that the careful choice of the optimisations can enhance the wall-clock time circa 16%.

The framework described in [PMR14] proposes an automatic tool for fast Design Space Exploration (DSE), targeted to FPGA technologies and integrated to commercial HLS tools. The synthesis is done by first creating a standard synthesised implementation which is profiled and transformed. This sequence of steps iteratively enhances the implementation, until the design constraints are met. An essential characteristic of this work that we explore in our project is not to rely on a fixed HLS tool. In our case, a previously synthesised library must be present.

In [Geo+13], the authors propose a methodology to enable the use of Domain-Specific Languages (DSLs) with HLS tools, creating a DSL-HLS design flow. In this approach, the application is analysed depending on its specific domain, for example, matrix computations should be handled and decomposed in an algebraic environment, and so on. In this way, different Intermediate Representations (IRs) are used to extract properties incrementally and optimise the implementation. This method is enhanced in [Geo+14], where automatic analysis are performed to extract execution kernels and dependency graphs, used to generate the hardware.

[Zuo+13] presents an HLS methodology for inter-block and intra-block optimisations based on Polyhedral Transformations, targeted to FPGA technology. The application's code is transformed iteratively, and directives are inserted automatically, to provide optimised HLS results. The flow starts by taking a data-dependent multi-block program defined by the user and determines the data access patterns and the loop transformations needed. The computation blocks are then transformed using the polyhedral model, and the communication part is generated in a fine-grained way [Zuo+14].

Pre-Characterized Function Implementations (FIMPs) are suggested by [Li+13b] to help the system level synthesis of DSP applications. The user input is a Synchronous Data Flow (SDF) graph created using FIMPs. By analysing the FIMP

libraries for the targeted hardware (ASIC, FPGA, CGRA) and the function-level parallelism, a set of feasible schedules is generated and then incrementally enhanced by testing different alternatives until the design constraints are met. The concept of FIMPs is used in our work (as a library of synthesised Function Models) to avoid the Combinatorial Optimization step.

The *bambu* framework is shown in [PF13]. It is a semi-automatic system to help the design through the HLS flow. The framework interfaces with commercial tools to get hardware parameters, like latency and resources usage. This tool was developed focused on memory-intensive applications and presents several optimisation steps to create efficient memory access patterns. However, the need for user interaction is high and represents the main weakness of this tool.

[Cza+12] presents the Altera OpenCL HLS tool, which takes as input an OpenCL implementation of the application and generates RTL code. OpenCL is based on C language, and the user must include pragmas to determine parallelism opportunities, which can be a drawback for inexperienced designers.

Table 3.3: Comparison of methods and tools for HLS

Reference	Input Language	User interaction	Methodology
[WO00]	C	moderate	Source code transformations
[Gup+04]	C	minimal	Source code transformations
[Per09]	Model-Based Design	strong	Block diagram using function units
[MHS10]	C	strong	Pragmas to identify SIMD code blocks
[Vil+10]	C	moderate	Templates and patterns
[Cza+12]	OpenCL	strong	OpenCL kernels and synthesis pragmas
[Geo+13]	DSL	minimal	Optimisations over Intermediate Representations (IRs)
[Zuo+13]	C	moderate	Polyhedral transformations
[Li+13b]	Model-Based Design	moderate	Libraries of pre-characterized FIMPs
[PF13]	C	moderate	Optimisation of memory access patterns
[PMR14]	C	moderate	Iterative optimisations with tool suggestions
[For+14]	C	moderate	Hardware/software

Table 3.3 shows a comparison of methods and tools for HLS. The majority of HLS tools found in the literature and reported here offer to the designer a C-based input. As our goal is to provide to software developers a fast way to have good-performer hardware synthesised, the use of C-language is a straightforward choice.

Also, the use of pragmas and user-handled code transformations should be avoided, since there is a need for intermediate to advanced hardware knowledge to execute them successfully. In other words, our work is based on minimum user interactions, while offering fair performance results.

3.4 Many-core design and methodologies

Different strategies for profiling source codes are shown in [Che+11], and in [LZL12] there can be seen a methodology for memory subsystem analysis for ASIP design. The issues and challenges on Adaptive ASIPs design are depicted in [JL11], focused on the development of massively-parallel heterogeneous MPSoCs and an approach for the design of processors capable of deal with parallel algorithms is shown in [Kar+13].

The profiling of computer vision algorithms is shown in [Dia+12], to the optimisation of a system for rover navigation. In [Iss+12], ASIP design methods are applied to an MPSoC architecture. This methodology is validated by the implementation of a wireless image transmission system.

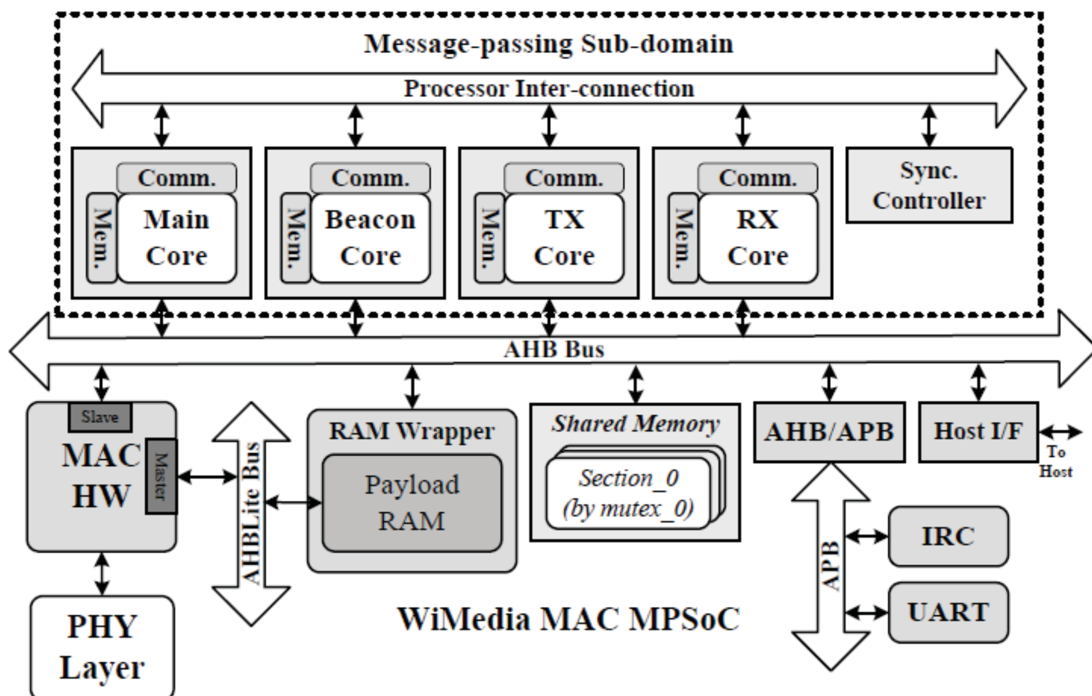


Figure 3.8: The WiMedia MAC MpSoC architecture [Iss+12].

The work presented in [KOA05] describes a parallel SIMD processor and memory array architecture which explores data level parallelism from image rows as well as from memory access patterns identified in IP/CV algorithms. The architecture is composed by 128 PEs, each one as an 8 bit 4-way Very-Large Instruction Word (VLIW) processor, and one 16 bit RISC processor.

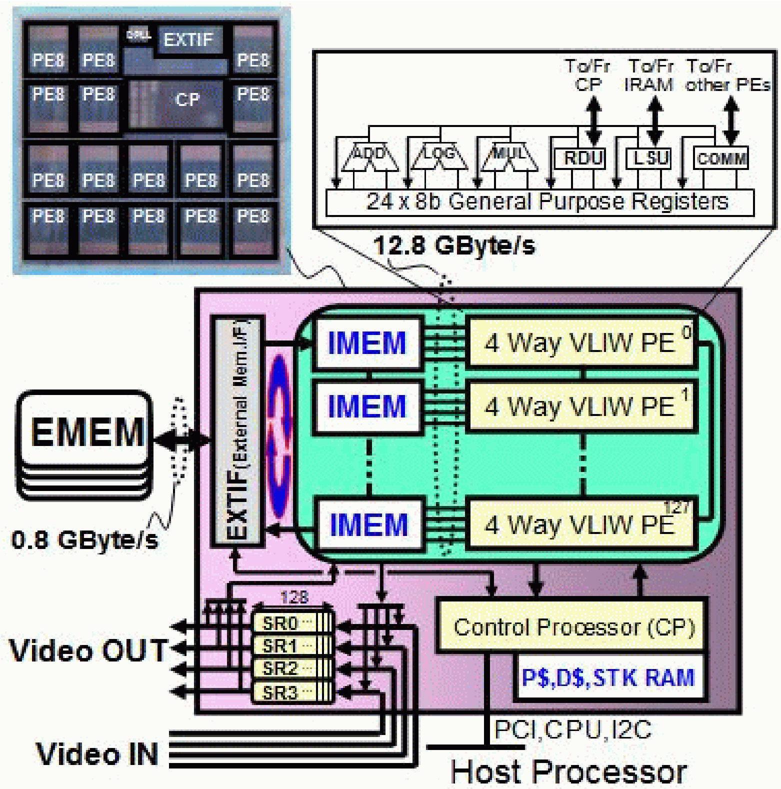


Figure 3.9: Block diagram of the IMAP-CE MPSoC and PE [KOA05].

In [Han+05] a methodology for mapping applications and evaluation of a Weakly-Programmable Processor Array (WPPA) is presented. A WPPA is composed of small optimised Weakly-Programmable Processing Elements (WPPEs) having the advantage of a sub-word parallelism, allowing for extreme parallelism exploration.

The design and programming of multi/many-core architectures are still hot topics in both academic and industry communities. The techniques and methodologies developed for an old technology sometimes are not feasible for new technologies (and new applications). The Kahrisma project, [K+10], presents a reconfigurable architecture composed of different processor models and interconnections. It also presents a software framework, providing retargetable compile-time tools.

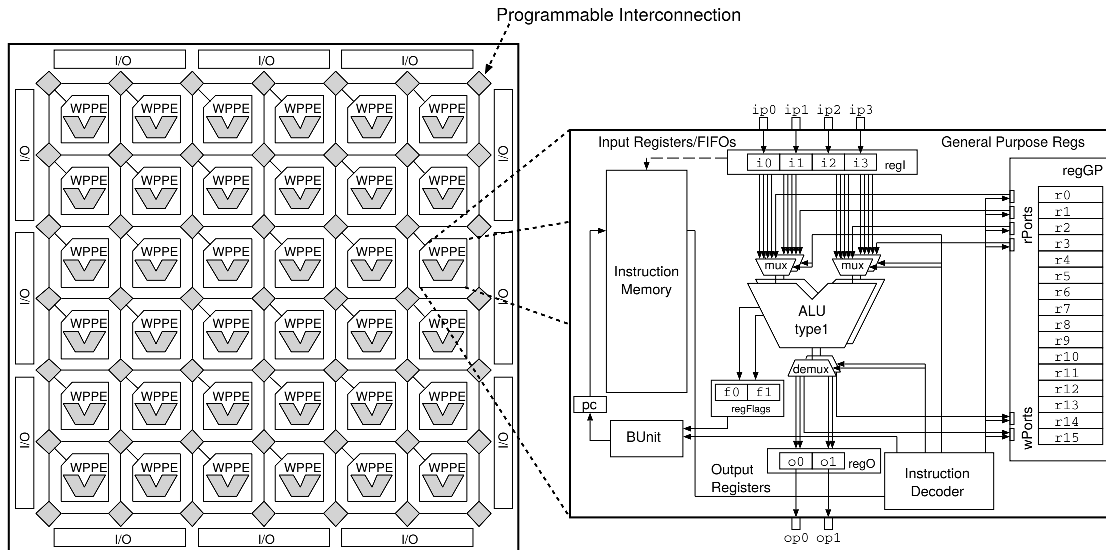


Figure 3.10: An example of a WPPA with parameterizable PEs [Han+05].

In [C+08], a framework for application parallelization to MPSoCs is presented. The applications are described using an extended version of the Kahn Process Networks (KPNs). Building blocks of different granularities are extracted from the process networks and can be then compiled for the MPSoC. The exploration of loops is one of the key concepts to speed up IP/CV algorithms.

The Tightly-Coupled Processor Arrays (TCPAs), [H+14]) was developed with the aim of having thousands of small PEs integrated as a single module to be inserted into heterogeneous MPSoCs. In this case, a heterogeneous MPSoC is composed of several different elements (homogeneous/heterogeneous multicores, I/Os, memory blocks, GPUs, and so forth.) interconnected through a NoC. The TCPA architecture seems to outperform embedded GPUs for applications with integer arithmetic. In our case, the majority of IP/CV applications use mainly integer and fixed-point arithmetic, which suggests that an intermediary architecture between a TCPA and a GPU can be a good solution.

The parallelisation of loop programs to Massively Parallel Processor Arrays (MPPAs) is shown in [TTH13], using a TCPA as an accelerator in a heterogeneous MPSoC. Most of these approaches share an essential step: the parallelism extraction (from loops, instructions, among others.). The use of KPNs goes in the direction of forcing the programmer to highlight the parallelism. In our approach, we consider

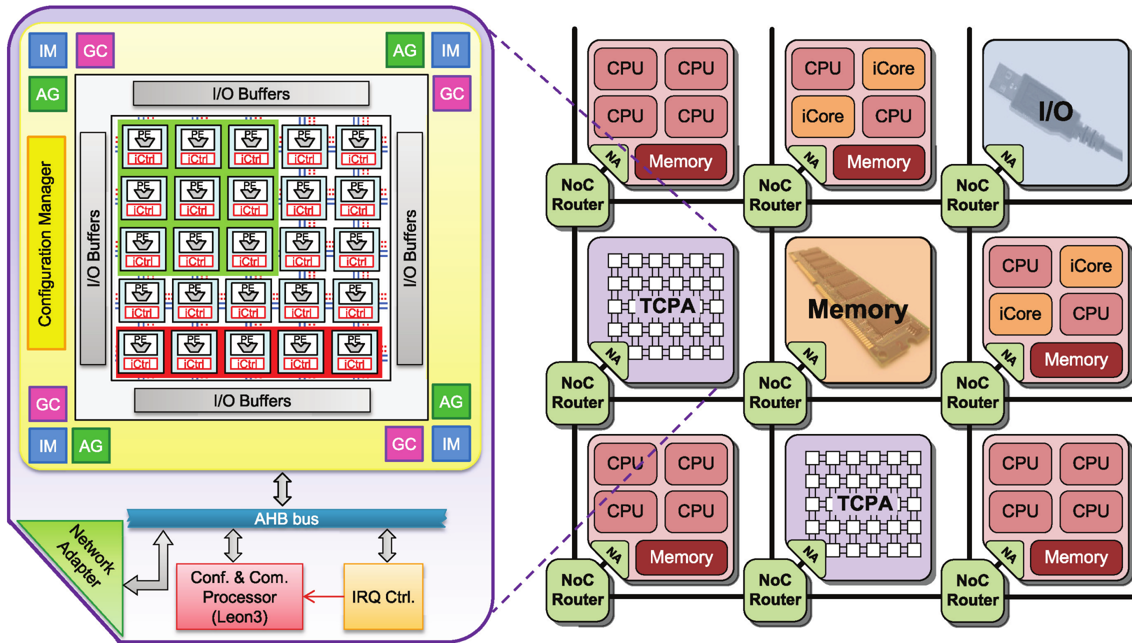


Figure 3.11: On the left, the TCPA architecture; on the right, an heterogeneous MPSoC example [H+14].

the use of a graph-based Domain Specific Language, to couple the spatial distribution of pixel/processors and the IP/CV parallelism exploration.

The works from [Goh+08] and [RG14] include a new dimension in the design space of Multi/Many-Core architectures: the hardware runtime adaptivity. The dynamic hardware reconfiguration can enhance the performance of the system in real-time, enabling for more flexibility when dealing with the design constraints. In IP/CV applications different widths can be configured in an ALU, depending on the algorithm to be executed, or even the choice among RISC or VLIW to explore sequential or parallel sequences of instructions.

The analysis of trends and issues for MPSoCs is the focus of [BD05]. The authors discuss how SoCs composed by tens or hundreds of PEs should be integrated through communication structures. Systems with tens to thousands PEs operating at high frequencies are expected to be common soon. The authors highlight that global wires are likely to have propagation delays too significant for the desired clock period. This statement constrains the use of buses and point-to-point connections to local communications only. Also, the synchronisation of this massive amount of PEs with a single clock may lead to an area explosion due to clock buffers, also preventing

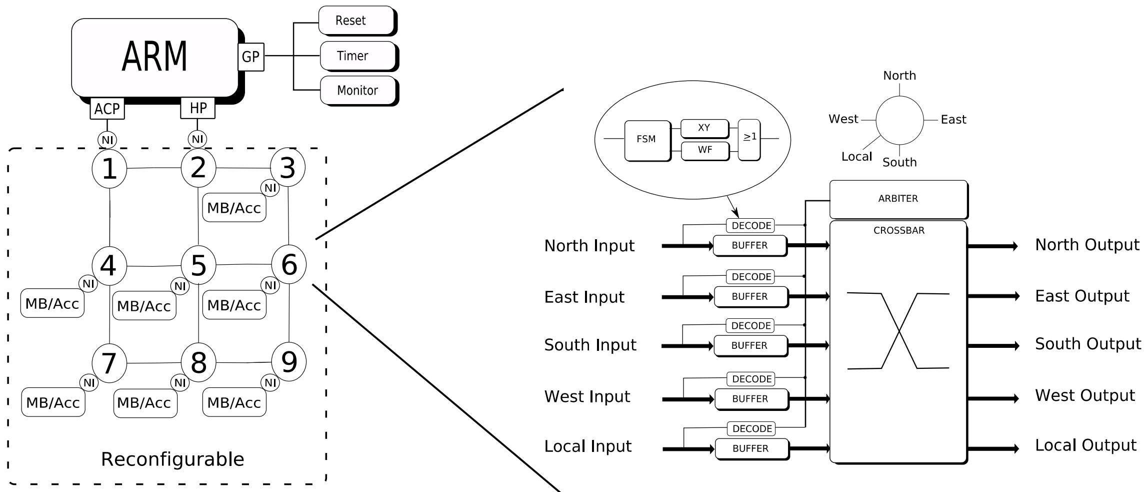


Figure 3.12: On the left, the RAR-NoC MPSoC; on the right, the configurable router architecture [RG14].

higher operating frequencies.

Multiple clock domains may be used, to support different application needs. However, bus-based systems are not designed to support this feature efficiently. The increasing amount of PEs sharing the same bus will lead to unfeasible high operating frequencies. Therefore, a different solution should be used. The Network-On-Chips are proposed as a solution to the highlighted issues. In our work, similar issues are expected, due to the amount of PEs, as well as the massive amount of data to be transported intra-chip. In Section 12.0.3 there are more details about the communication issues in the design of our architecture.

In [SFP13], the utilisation of an MPSoC based on a NoC architecture is discussed for a real-time IP/CV application. The MPSoC is based on heterogeneous Tiles, composed of distributed memories, GPPs, and hardware blocks dedicated to specific functions (motion estimation, discrete transforms, and filtering). Each Tile has also a Network Interface (NI) which is connected to a NoC Router. This architecture does not use a spatial pixel distribution, like in a pixel array, however, it is configurable to work as a cascade of processing elements accordingly to the application. This type of architecture takes advantage of the NoC structure to trade flexibility with communication delay. Another advantage is the possibility of re-using the same PE in different parts of the image processing chain since the NoC topology allows for

data flow loops.

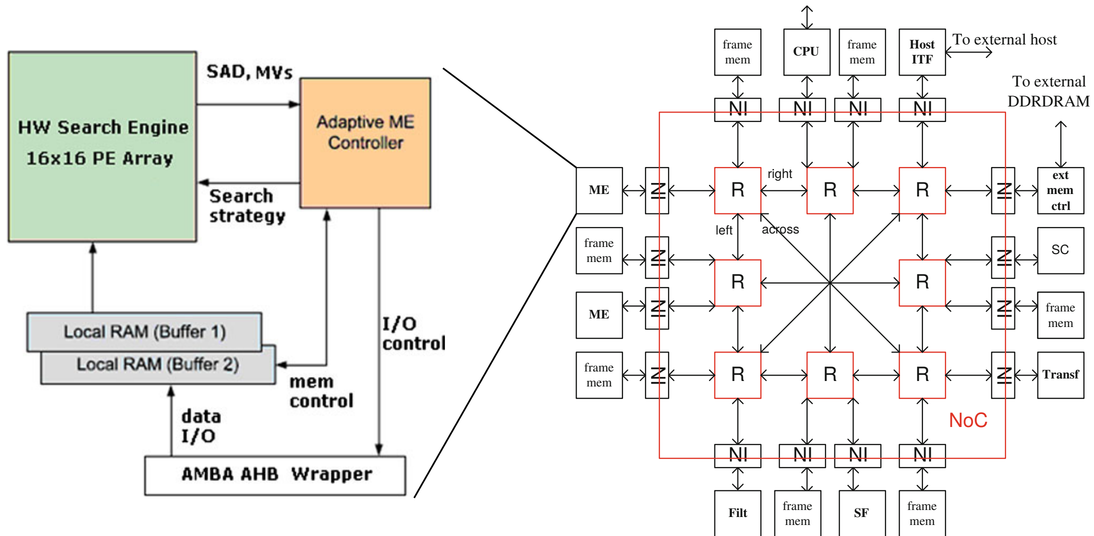


Figure 3.13: On the left, the ME tile diagram; on the right, the NoC-based MPSoC architecture [SFP13].

A Coarse-Grained Processor Array (CGA) developed over a NoC is proposed in [Pha+13]. The architecture is organised as a 2D mesh, 4-connected, with each Tile composed by the Router, a RISC processor, Data, and Program Memories. An interesting aspect here is the use of independent Program Memory in each Tile. This feature is implemented to allow for different applications running simultaneously. In our case, considering an image region allocated to each Tile, the same application should be running, and the replication of Program Memories could be considered a waste of area. However, we highlight here that the synchronisation among the PEs can be done by data dependencies, instead of being performed by a global controller. This feature can also allow for faster execution of pixel-based branches, where different Tiles can be in different states simultaneously.

The work of [KKK13] explores partially the design space of a many-core architecture used for image segmentation. The authors propose a layered architecture composed of a CMOS photo-detector array at the top of the processor layer. The processor layer is based on a 2D array of PEs, 4-connected in a point-to-point configuration. Each PE is responsible for a region of the image and composed by an ALU, Register File, Local Memory, and a Communication Unit. In this work, the

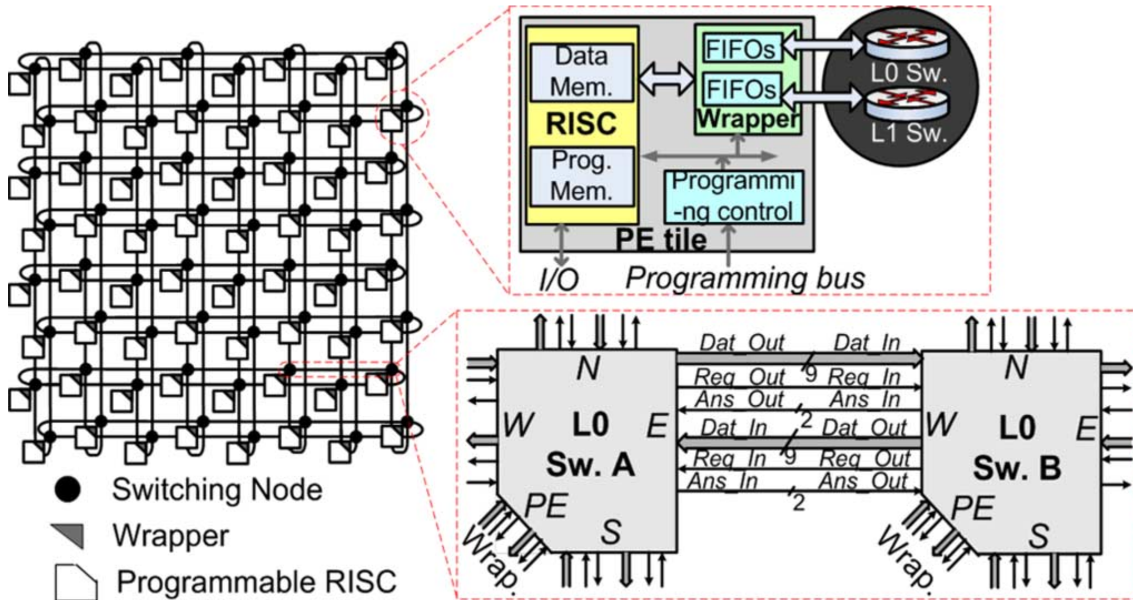


Figure 3.14: The NoC-based MPSoC proposed by [Pha+13].

authors analysed the impact of varying the number of PEs with fixed image size. In our work, we extended this analysis to different types of PEs, several communication structures, and memory organisation.

Several commercial and free tools capable of system-level design can be found in the market. Diverse features as simulation, profiling, HDL code generators, virtual design platforms and so on have been developed to deal with the vast design space for application-specific processing systems. Most of the tools and methodologies found in the literature focus on few aspects in the design space, and there are no methodologies focused on the specific problems a hardware engineer would find when developing a vision processor. In this work we performed a simple analysis to bridge this gap, allowing the designer to have more detailed information from the application domain, by analysing the processing needs from the outputs to the inputs.

It can be identified in the literature, that the main bottleneck in IP/CV architectures is the data access. The parallelism exploration must be extended from the instruction level to the memory access level. This can be done by exploring the focal-plane approach and the spatial parallelism of IP/CV algorithms. Another critical point is the migration from analogue to digital processing, focusing the analogue

Table 3.4: My caption

Reference	PE Type	Communication Structure	Mapping Method
[Iss+12]	heterogeneous	Bus	Application-specific SoC
[KOA05]	VLIW PE – SIMD	Bus	Memory access patterns
[Han+05]	WPPE	Switching matrix	Sub-word parallelism
[C+08]	independent	independent	Mapping of KPNs
[TTH13]	RISC PE – SIMD	Switching matrix	Loop parallelisation
[SFP13]	PE Array	NoC	Cascade of Pes

parts to acquisition only, leaving the processing tasks to the digital parts, which can have more flexibility and lower costs. Considering these points, the general architecture suggested in Figure 1.7 is envisioned to be able to provide the aimed advantages: an application-specific many-core system with spatially distributed pixel registers, integrated by an intrachip communication structure. The following topics show a bibliographical review directed to the development of this architecture.

In the current work, we are developing a mix of the presented techniques, to identify the application's needs and refine the processor's model since the beginning of the design flow. It is important to highlight that not only the processing part but also the communication needs are considered in our design.

Chapter 4

Image Processing Concepts

In this chapter, we review some of the basic concepts related to the Image Processing and Computer Vision (IP/CV) area. We provide only a non-exhaustive overview which should be enough for most readers to understand the development of this thesis. [3]

4.1 Image Processing and Computer Vision Chain

IP/CV applications appear in several different areas, such as safety monitoring, biometric devices, industrial quality control, driving assistance systems, medical diagnostics, remote sensing, underwater inspection, and consumer market. These applications require different algorithm types, from simple contrast correction to complex pattern recognition. In this section, we highlight the IP/CV main characteristics used in this work.

Most of the image processing algorithms are used to generate a new image from the original image, performing operations such as noise filtering, feature enhancement (borders, lines, regions, objects) and feature detection (vectors of characteristics describing the features). Another type of algorithms, like pattern recognition, works analysing the vectors of characteristics to identify objects. This work is fo-

[3]This chapter is based on excerpts and adaptations from previously published papers [MH14], [YLH17].

cused on the algorithms, mentioned at first, which transform the original image into another image.

Figure 4.1 shows an example of image processing stages for an application. Most of the works found in the literature about Vision Processors are focused on offering processing solutions for the first and the second stages [KG06]. The approach of the present work extends the analysis also to intermediate-level operations (stages 3 and 4).

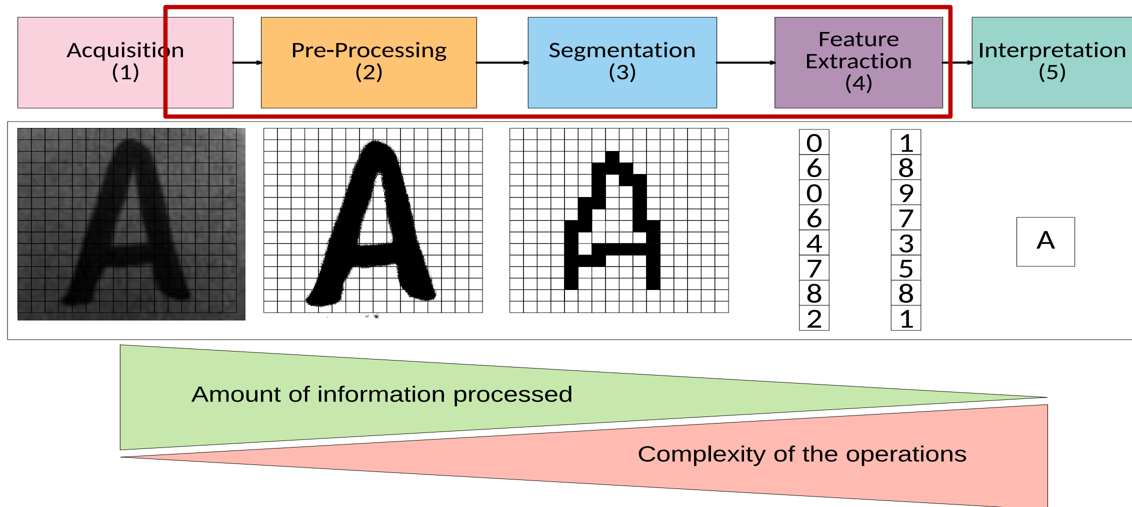


Figure 4.1: A general IP/CV processing chain [KG06; MH14; MK16; YLH17].

A typical IP/CV system contains one or more of the stages shown in Figure 4.1:

1. *Acquisition*: It encompasses the CMOS pixel array and its ADCs. Its output is the original image.

2. *Pre-Processing*: It is responsible mainly for noise removal through filtering. Its outputs are intermediary images.

3. *Segmentation*: It localizes the Regions of Interest (ROI) in an image (ROI definition depends on the algorithm used). Its outputs are intermediary labeled images.

4. *Feature Extraction*: Scans the ROIs to extract their characteristics (features). The outputs are feature vectors representing these characteristics.

5. *Interpretation*: Extracts useful information from the feature vectors. The outputs are the information extracted. In the example, could be the identification of the letter A.

Each state of the processing chain has different characteristics of complexity and amount of operations. The first stages use low-level pixel operations, mostly based on trivial arithmetics. However, the amount of information to be processed is enormous. In contrast, stages close to the end of the processing chain need high-level, sophisticated algorithms, but operate over less information.

It is important to observe in Figure 4.1 that the amount of data to be processed, in general, decreases from one step to the next. On the other hand, the complexity of the functions used in each step increases. This means that at the beginning of the chain there are data-intensive simple operations, and at the end of the chain complex operations over few data [KG06], [Brä+13], as in Figure 4.1.

Also, it is common to find a considerable amount of Instruction-Level Parallelism at the beginning and more sequential algorithms at the end of the chain. As a result, for the beginning of the chain, one can propose architectures such as Vector Processors, Superscalar and VLIW architectures, as well as direct hardware implementations, for example using embedded reconfigurable fabric. For the end of the chain, one can propose more general-purpose processors, to allow for more complex algorithms implementation. This information can be used by the chip designer as a starting point on which architectures can better explore algorithm characteristics.

4.2 Operation Types

In Stages 2 and 3 (Figure 4.1), the operations work in the pixel level, transforming a given image in another one. The IP/CV operations at the pixel level can be classified in three types, as shown in Fig.4.2 ([KG06], [MH14], [Zar11]):

- Pixel operation: Gets a single pixel as input to generate a single output pixel.

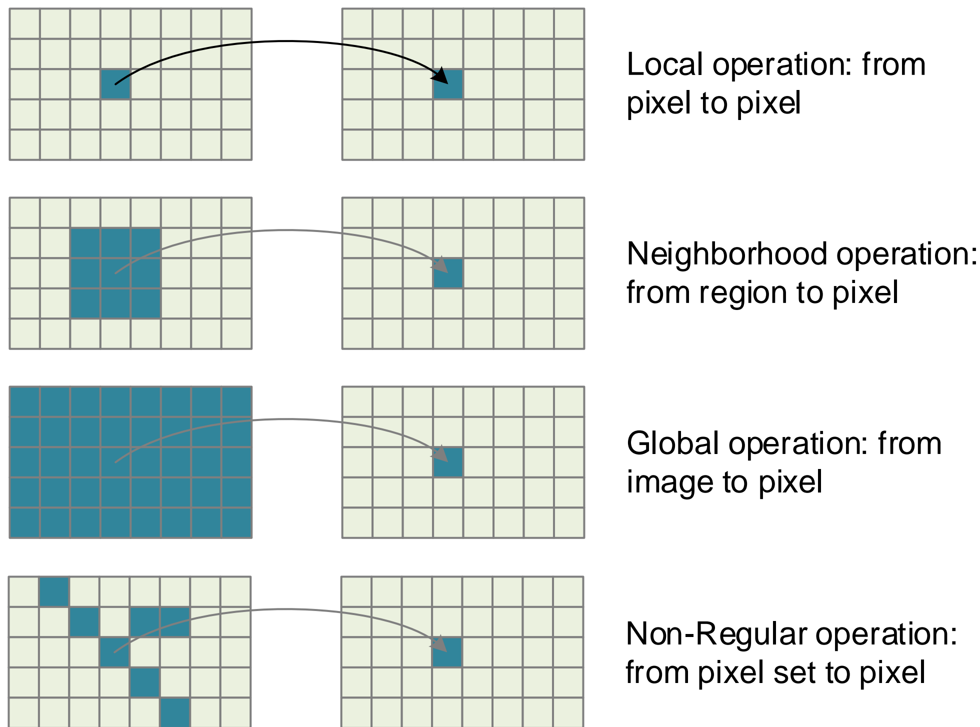


Figure 4.2: Basic pixel-level IP/CV operations [KG06].

- Neighborhood operation: Gets as input a regular group of pixels to produce a single output pixel.
- Global operation: Gets as input the whole image to generate a single output pixel.
- Non-Regular operation: Gets as input a set of pixels which share a common property to produce a single output pixel.

We must highlight that Pixel and Global operations are just particular cases (lower and upper limits) of Neighborhood operations. Several different IP/CV applications can be described using combinations of those three types of operations.

4.3 The OpenVX Project

In the parallel processing domain, one of the most important goals is to identify and explore the maximum amount of parallelism possible [KG06]. Looking at the IP/CV algorithms, and considering the spatial distribution of PEs over the image area, we

can identify coarse-grained parallelism. In the IP/CV domain, the OpenCV library [tea17] is one of the most used collections of algorithms. It is used for educational, industrial and scientific purposes, and can be considered as an informal standard.

With the increasing number of complex IP/CV commercial applications, the industry identified the need for an IP/CV standard *de facto*. The Khronos Group [Edi17] released in 2014 the first version of the OpenVX standard. OpenVX is a set of rules and design patterns created to describe IP/CV applications. Similar to other standards, like OpenCL and OpenGL, the OpenVX actuates as a *frontend* for application's description. The *backend* should be created by each hardware manufacturer, accordingly to its architecture's characteristics [Edi17].

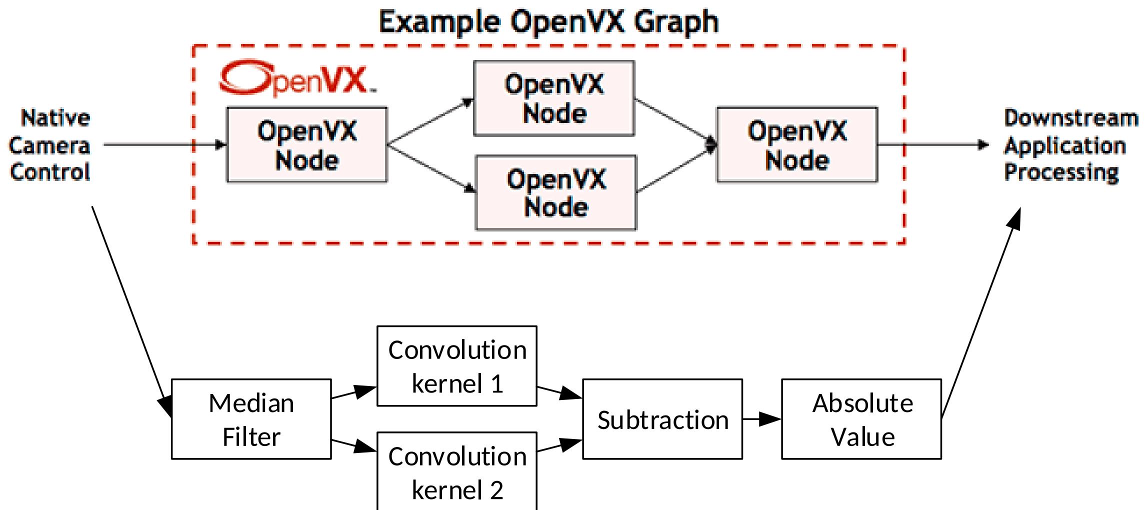


Figure 4.3: OpenVX graph of a simple application [Edi17],[Mor+16b]

OpenVX defines a programming model based on graphs, composed of nodes and links. Each node is a complete IP/CV algorithm (filtering, motion detection, arithmetic operations among images, and so on), as shown in Fig.4.3. In general, both input and output of a node are images. The OpenVX functions/blocks/nodes were defined with the following main criteria [Edi17]:

- **Applicable to Acceleration Hardware**: The vision functions most feasible to be implemented as hardware accelerators were chosen.
- **Very Common Usage**: The most common vision functions found in academy and industry were chosen.

- **Encumbrance Free**: Algorithms, methods, and techniques used in OpenVX are free of property/intellectual claims.

Considering the broad application of other Khronos Group standards, we decided to develop our architecture to implement most of the OpenVX function blocks. Only the functionalities of the function blocks were analysed. To be compliant with the standard, some features like garbage collection and kernel management were not considered.

Chapter 5

Application-Specific Hardware/Software Co-Design

In this chapter, we perform an analysis of the application-specific features which can determine hardware/software design templates. We start this chapter by analysing different pixel acquisition architectures, to highlight their main advantages and drawbacks. Afterwards, we check some parallelism exploration possibilities and define the basic programming model of our architectures. In the end, we show the specification of a general architecture template which will be used as a guide for further development in this thesis [4].

5.1 Image Acquisition Schemes

In [Zar11], different CMOS sensor acquisition schemes are discussed. This design choice reflects directly on the amount of parallelism that can be explored, but also in the size and complexity of the analogue part of the system. As described in Figure 1.4, there are three basic acquisition schemes, each one with its advantages and disadvantages (*Single-Pixel Stream*, *Region-Based Stream*, and *One Stream per Pixel*). In this section, we analyse each scheme, to provide a comparison among

[4] This chapter is based on excerpts and adaptations from the previously published papers [MH14] [YLH17]

them. Let's first consider the following parameters, essential to define the pixel acquisition scheme:

- **Resolution**: M rows and N columns.
- **Throughput**: Thr : represents the amount of pixels generated per time unit.
- **Neighborhood**: a pixel array of size $L \times K$, needed in the first processing stage.
- **Region**: an image region of size $W \times Z$, used only in the *Region-Based* acquisition scheme.
- **Initial Delay**: $\triangle_{init}$: the amount of cycles until the first neighborhood is filled.

5.1.1 Throughput

The throughput measures the acquisition speed: how many pixels are acquired and made available at each clock cycle (pixels per cycle - *ppc*). Table 5.1 shows the equations used to determine the throughput of each acquisition scheme.

Table 5.1: Throughput for each acquisition scheme.

Acquisition Scheme	Thr (*ppc*)
Single Pixel	1
One per Pixel	$M \times N$
Parallel Rows	M
Parallel Columns	N
Region-Based	$\frac{M \times N}{W \times Z}$

Figure 5.1 plots a specific case of the throughputs for a fixed size image ($M = N = 1024$), as a function of $W = Z$. The acquisition scheme which has more flexibility is the *Region-Based Stream*, which provides more possibilities for the throughput choice. Intuitively, the *Single Pixel Stream* and *One Stream per Pixel* schemes are the cheapest and the most expensive (considering the silicon area), respectively. Therefore, it is not naive to say that the *Parallel Rows/Columns* and *Region-Based*

Stream schemes should have the best cost/benefit ratio. In the graph, the highlighted point $(32, 1024)$ shows the case when both schemes have the same amount of ADCs (which roughly corresponds to the same cost in silicon area).

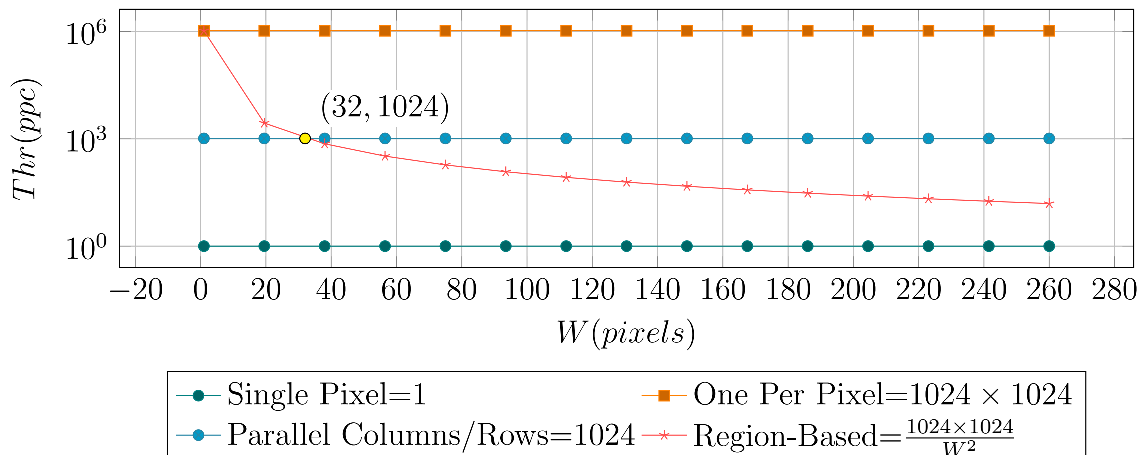


Figure 5.1: Throughput considering: $M = N = 1024$, $W = Z = \{variable\}$.

5.1.2 Initial Latency

In general, the first algorithms applied to the acquired image are based on groups of pixels (neighbourhoods). These neighbourhoods can have any size or shape. However, square shapes are the most common ones and therefore used for the analysis in this section. Table 5.2 shows how to determine the Initial Delay for each acquisition scheme.

Table 5.2: Initial Delay for each acquisition scheme.

Acquisition Scheme	$\triangle_{init}(cycles)$
Single Pixel	$N \times (L - 1) + K$
One per Pixel	1
Parallel Rows	K
Parallel Columns	L
Region-Based	$W \times (L - 1) + K$

Figure 5.2 shows the Initial Delays for each acquisition scheme, considering different parameter's values. Observing the plotted functions, we can see that the *Region-Based* scheme provides a variable range which encompasses all the spectrum from the *Parallel Columns/Rows* to the *Single Pixel* schemes.

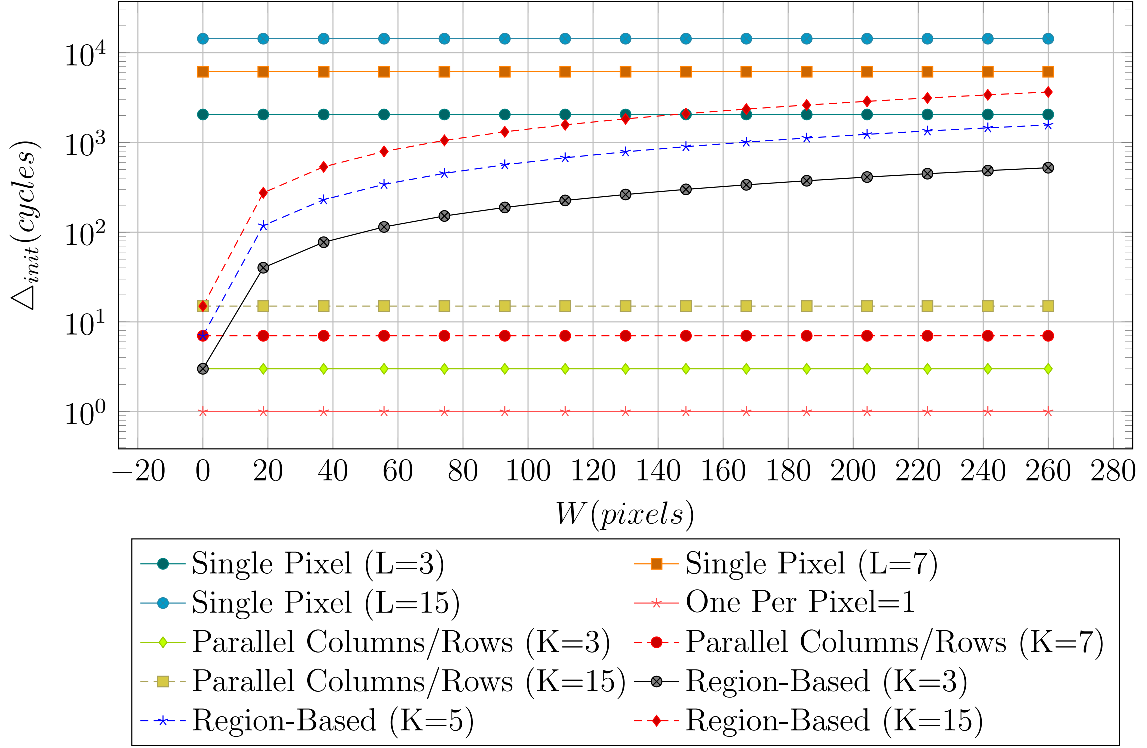


Figure 5.2: Initial Delay considering: $M = N = 1024$, $L = K = \{3, 7, 15\}$, $W = Z = \{variable\}$.

5.1.3 Image Delay

The *Total Delay* is the total amount of cycles to load the complete image in each acquisition scheme. Equation 5.1 shows how to determine the *Total Delay* for any acquisition scheme.

$$TotalDelay = \triangle_{init} + \frac{M \times N}{Thr} \tag{5.1}$$

The *Total Delay* is a combination of the *Initial Delay* with the total amount of pixels in the image, adjusted with the *Throughput*. The graphs in Figure 5.3 show the *Total Delay* of each acquisition scheme. Once more we can observe that the *Region-Based* acquisition scheme provides more design flexibility, which is an essential characteristic in the scope of this work.

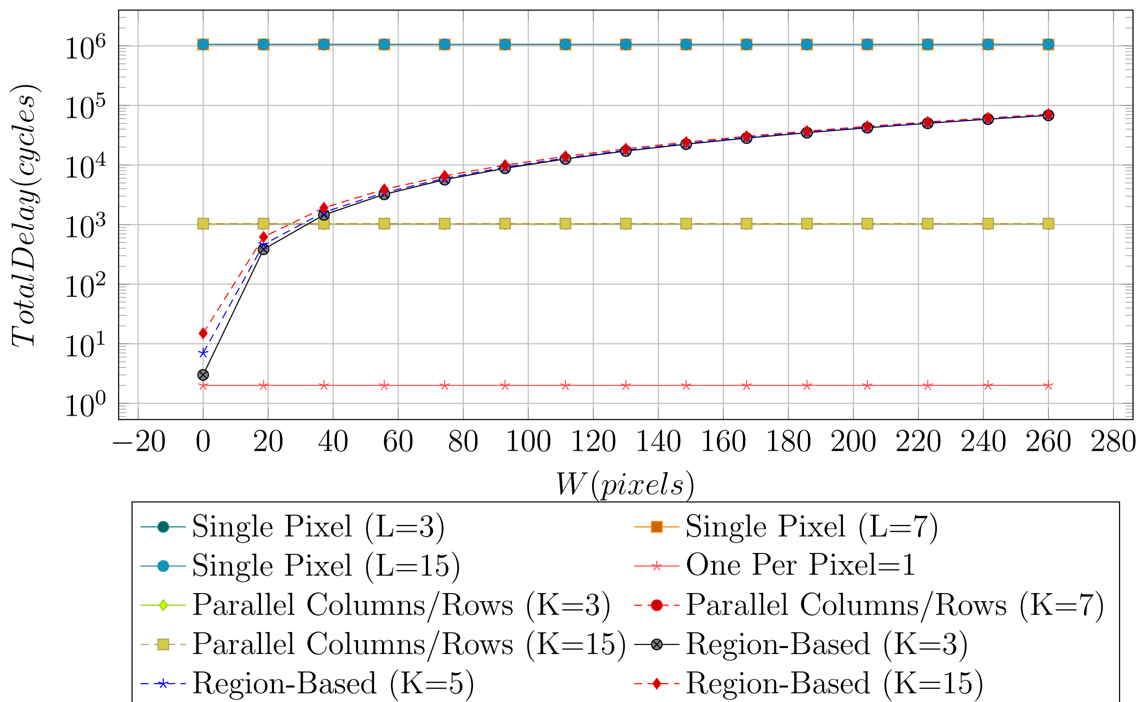


Figure 5.3: Total Delay considering: $M = N = 1024$, $L = K = \{3, 7, 15\}$, $W = Z = \{variable\}$.

5.1.4 Overview

The discussion made until now in this section was focused on the acquisition schemes from the processing part. From an analogue design perspective, there is also some interesting analysis to be done. The focus of this work is on the processing architecture possibilities (here considered purely digital), not in the analogue design. However, we can make a qualitative analysis of the analogue design issues related to the vision processor architecture:

Scalability The focus of this work is the design possibilities of a multi/many-core vision processor. Therefore, the scalability refers to changes in the image resolution and the number of processors. Considering *Throughput*, *Initial Delay* and *Total Delay*, the most flexible acquisition scheme is the *Region-Based* since it provides more opportunities for design changes based on the resolution and number of processors. Also, a stackable architecture (like in Figure 1.7) would have equal propagation delays from the ADCs to the Pixel Memories responsible for image buffering. In the other acquisition schemes, several different wire lengths would be used, resulting in

less deterministic behaviour and more noise sensitivity.

Image Quality A pixel array is composed of: a light sensitive area (photodiode) in each pixel, amplifiers, and connection through the switching/transfer bus to the ADCs. The quality of an acquired image is directly related to the amount of light it can acquire from the environment. In a Focal-Plane approach, the sensitive area is reduced due to the addition of processing/communication/storage elements to the pixel sensor area, reducing the *fill-factor*. The stackable architecture shown in Figure 1.7 (with a *Region-Based* scheme) optimizes the sensitive area by putting non-acquisition related elements in another chip layer. This separation could also be done in the other acquisition schemes. However, the outputs of the pixels need to be routed to the die borders of the pixel-array, also occupying the sensitive area with the data transfer wires.

Design Flexibility The design of mixed-signal (analogue and digital) integrated circuits has several issues. As a *thumb rule*, analogue designs are less sensitive to noise and manufacturing problems when designed/manufactured in older foundry technologies. However, the digital design benefits from smaller power consumption and higher operating frequencies. The 3D chip shown in Figure 5.4 can be manufactured using different node sizes, allowing both analogue and digital parts to benefit from it.

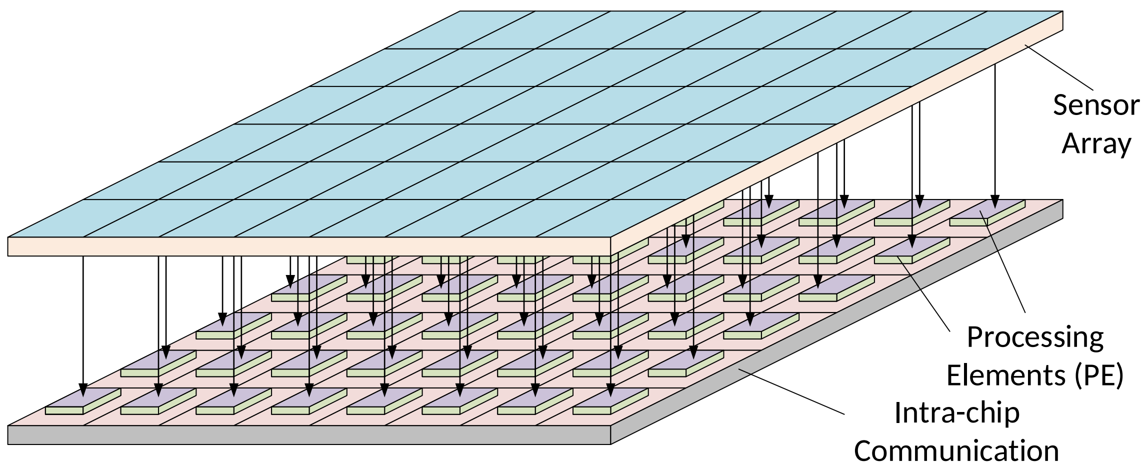


Figure 5.4: *Focal-Plane/Near-Sensor* Image Processing concept [Fos98] [MH14] [MKH15a].

Programmability One of the significant challenges of using a multi/many-core chip is to solve the problem of programming several cores efficiently. By observing the different acquisition schemes, the *Region-Based* one preserves the same characteristics as the *Single Pixel* one, with the difference on the number of pixels to be processed. This favours the utilisation of legacy codes and algorithms with minimal modifications, in comparison with the other acquisition schemes, as will be detailed in Section 5.2.

Considering the analysis performed in this section, we can consider the *Region-Based* acquisition scheme as the most suitable for the design-space exploration, because it is the most flexible, offering a balance among the parameters analysed.

5.2 Parallelism Analysis

To analyse the parallelism exploration opportunities, we will consider the sample IP/CV processing chain shown in Figure 5.5. The application shown is an *Edge Detection* composed by simple blocks. A first type of parallelism appears in the IP/CV processing chain as *Function-Level* (or *Block-Level*). Looking to Figure 5.5, we can observe that some blocks have no data dependencies with others: *SobelX* can be executed in parallel to *SobelY*, as well as the pair of *ABS* blocks among themselves.

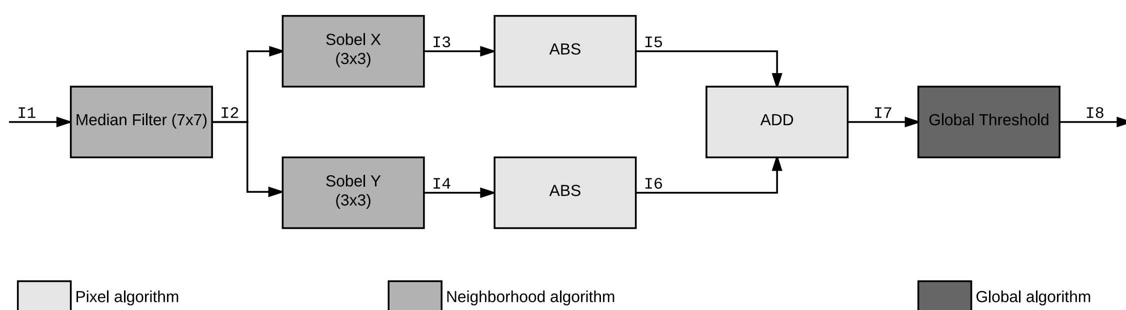


Figure 5.5: Simple Edge Detection application.

5.2.1 Macro-Pipeline

In a real-time IP/CV system, not a single, but a sequence of images is processed (in general, a continuous stream of images). A simple type of parallelism appears among the different stages and the image stream. Figure 5.6 shows a *Macro-Pipeline* for the processing chain of Figure 5.5, in which the processing stages can be applied in parallel to an image sequence, in a similar fashion as a pipeline in a common processor [MH16].

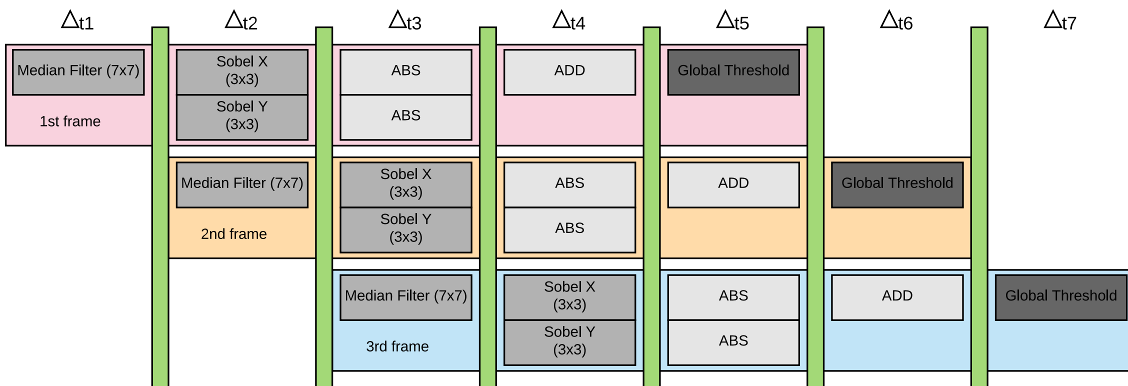


Figure 5.6: Macro-Pipeline: exploring the parallelism in image sequences [MH16].

5.2.2 Operation-Level Parallelism

Some IP/CV algorithms are composed by several operations, e.g. the *Convolution* in Figure 5.7. This algorithm is used to perform different functions, like the *SobelX, SobelY, Median Filter* blocks from Figure 5.5. The *Convolution* in Figure 5.7 operates over a 3×3 neighborhood, by multiplying each pixel to a coefficient (k_n) and summing up the results. We can observe that each multiplication is independent of the other, as well as some of the additions. Therefore, these operations could be performed in parallel. In a VLIW processor, this type of parallelism would be called *Instrucion-Level Parallelism* (ILP).

5.2.3 Loop-Tiling / Spatial Parallelism

With the choice for the *Region-Based* acquisition scheme (Section 1.4), a type of Spatial Parallelism can be explored. The left side of Figure 5.8 shows how a generic

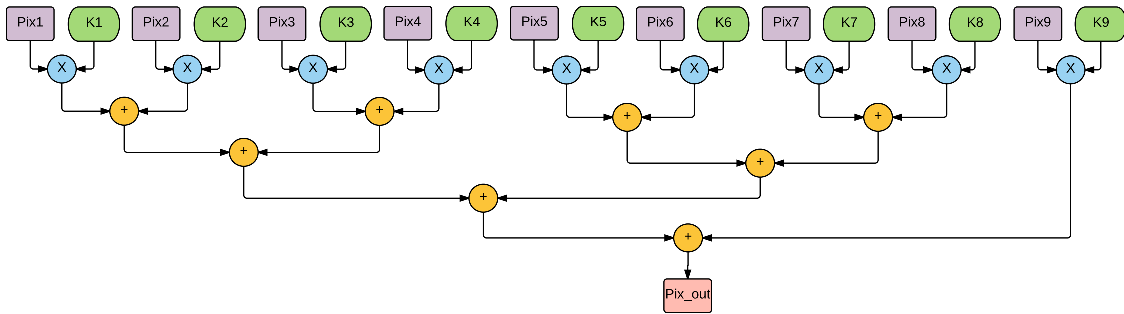


Figure 5.7: Operation level Parallelism: 3×3 Convolution example [MH16].

IP/CV algorithm is commonly implemented: for each pixel in the image the processing chain blocks are applied. Due to data independence, the nested loops can be both unrolled, and each loop body could be computed in parallel. This would match with the *One per Pixel* acquisition scheme. This scheme would provide the highest throughput, at least for the first stage in the Image Processing and Computer Vision (IP/CV) processing chain. However, the cost and technical implementation issues of having one processor per pixel are not suitable for practice.

An alternative, which matches the *Region-Based* acquisition scheme, is to divide the image into regions with the same size of the acquired regions. In this way, the two image-sized nested loops would be divided into smaller nested loops with the size of each region. This process is also known as *Loop-Tiling* [YLH17].

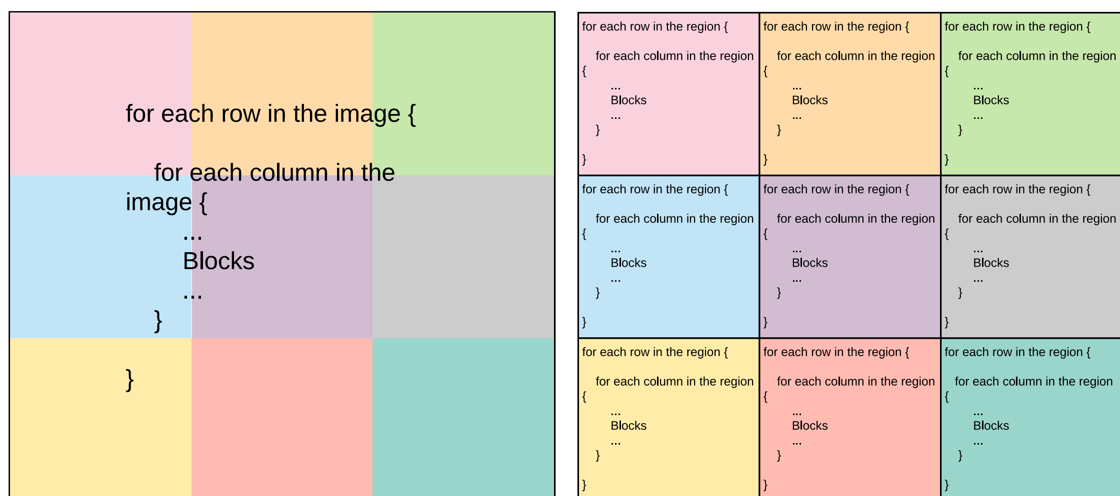


Figure 5.8: Loop-Tiling example for a generic IP/CV block, as defined in [MKH15a].

The same operations will be performed, however in parallel, with each region allocated to a different PE, as shown in Figure 5.9. This algorithm/data mapping

favours the scalability of the system since a resolution increase can be easily solved by inserting new PEs, or by changing the number of pixels per Region/PE.

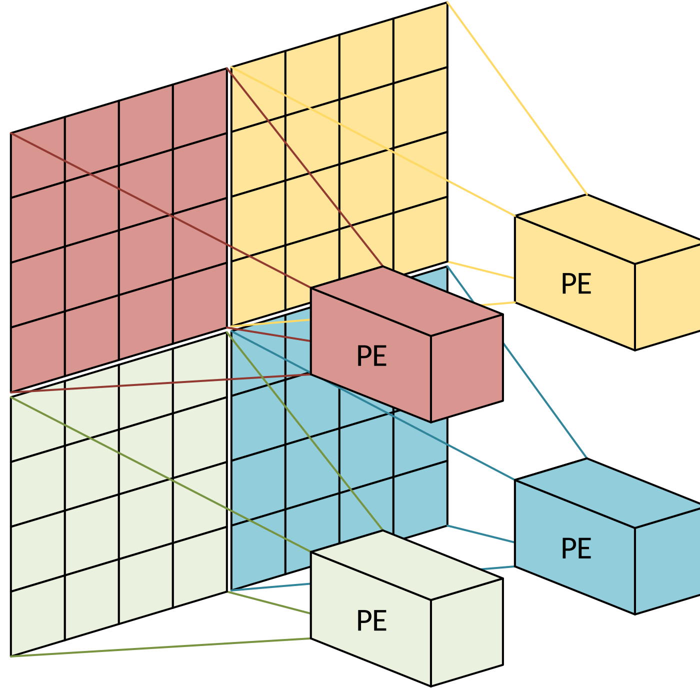


Figure 5.9: Exploring the Spatial Parallelism using a *Region-based* configuration: each PE is responsible for a region of the image [MH16].

In the next section, we show the definition of a conceptual template for our many-core vision processor.

5.3 Tiled Parallel Architecture

As discussed in section 5.1, the *Region-Based* acquisition scheme offers interesting characteristics for the type of problem we want to solve. It is straightforward to see that the *Loop-Tiling* presented in section 5.2.3 matches with the *Region-Based* acquisition scheme. Considering the ideas highlighted, we propose a general architecture description which shall have these features.

In Figure 5.10, we show the *General Tiled Architecture*, where each tile is composed by a *Pixel Memory*, a *Processing Element*, and a *Communication Interface*. Besides, the tiles are integrated through an *Inter-Tile Communication Structure*. Each tile is responsible for processing the pixels within its image's acquired region.

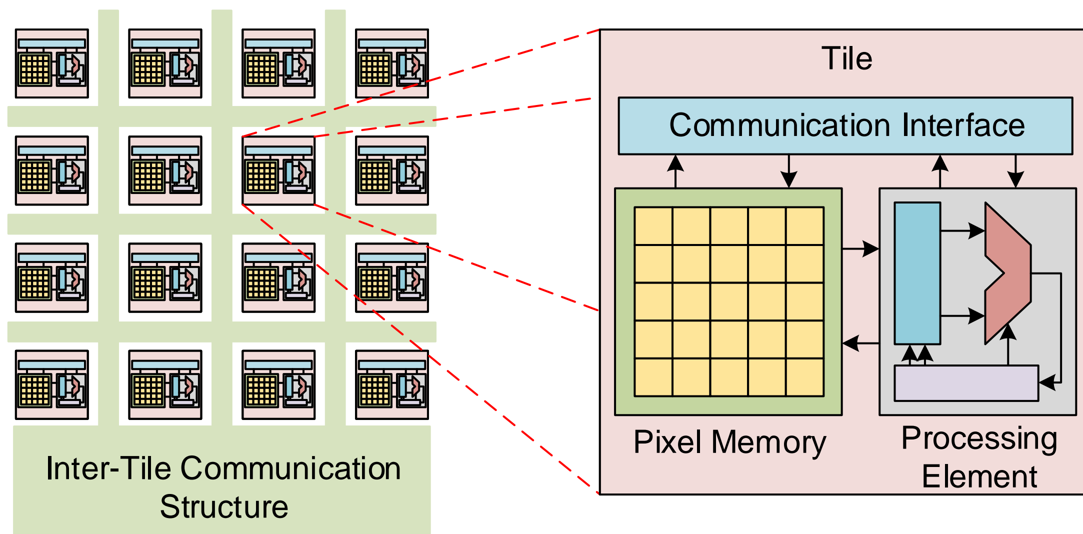


Figure 5.10: General Tiled Architecture

To simplify the nomenclature for the rest of the thesis, we define in this section the concept of a *Tile-Code*. The *Tile-Code* is the algorithm/program/code needed to generate a **single** output pixel. This is equivalent to say that the *Tile-Code* is the set of operations in the loop-body of Figure 5.8, originated from a complete loop-unroll. The *Tile-Code* has some characteristics that can be used in this project:

Programming Model: First of all, from its definition, the *Tile-Code* is implemented in each Processing Element (PE) independently. If we program the *Tile-Code* in a parameterizable way, the same program can be just replicated in each PE and repeated for each pixel. This offers an easy and low-complex solution for mapping the IP/CV algorithms to the many-core architecture.

Modularity: A single *Tile-Code* can be used to represent a complete IP/CV processing chain. However, each block of an IP/CV processing chain can also be represented by a *Tile-Code*, if we consider the intermediary images as outputs/inputs of the *Tile-Codes*.

Abstract Representation: The *Tile-Code* must be seen not strictly as a source-code (e.g. C, C++, Java) but as an abstract representation of the operations needed to generate a single output pixel. The IP/CV applications considered in this work

can be broken down into the four basic pixel-level operations shown in Figure 4.2. This abstraction comes from the fact that the PEs are not defined yet. As we will show in this thesis, the *PE* can be a simple programmable processor (in this case, the *Tile-Code* would be a source-code), a systolic array (in this case, the *Tile-Code* would be represented in the array), or even a direct hardware implementation (in this case, the PE would be a result of a high-level synthesis of a *Tile-Code*).

Part III

High-Level approach

Chapter 6

The high-level methodology

In this part of the thesis, we propose a methodology to design and program many-core vision processors from a high-level perspective: we use the desired application as first input to determine the underlying hardware/software architecture.[5]

6.1 Introduction

Considering the concepts, results and ideas from the bibliography, some key points must be highlighted in this work. An important aspect is a need for an application's development layer which can make the hardware transparent to the programmer. Another important aspect is the spatiotemporal characteristics of IP/CV algorithms. A digital image is a spatial distribution of pixels and the IP/CV techniques should be considered as spatial problems [Ung58]. This means that the hardware/software architecture should also be thought in a spatially distributed way, to better deal with the IP/CV techniques.

Task graphs represent the dependencies between the components of an algorithm and are used in many areas. They represent a set of tasks and are used to distribute them in a particular order among a defined number of execution units so that they can be processed as efficiently as possible. The complexity of the individual tasks is

[5]The chapters presented in this part are based on excerpts from two Master Theses supervised by the author - [Wer15] [Fri15] - as well as two a previously published papers: [MKH15a] and [Mor+16b].

not uniform but depends on the task to be solved. In this way, a task can correspond to a single machine instruction, but also to a complete complex program.

Figure 6.1 depicts an example of task-graph mapping and scheduling. A generic task-graph with eight tasks is provided. The mapping task starts by assigning a cost for each node and each edge of the task-graph. The node's cost is known as processing cost, and the edge's cost is known as communication cost.

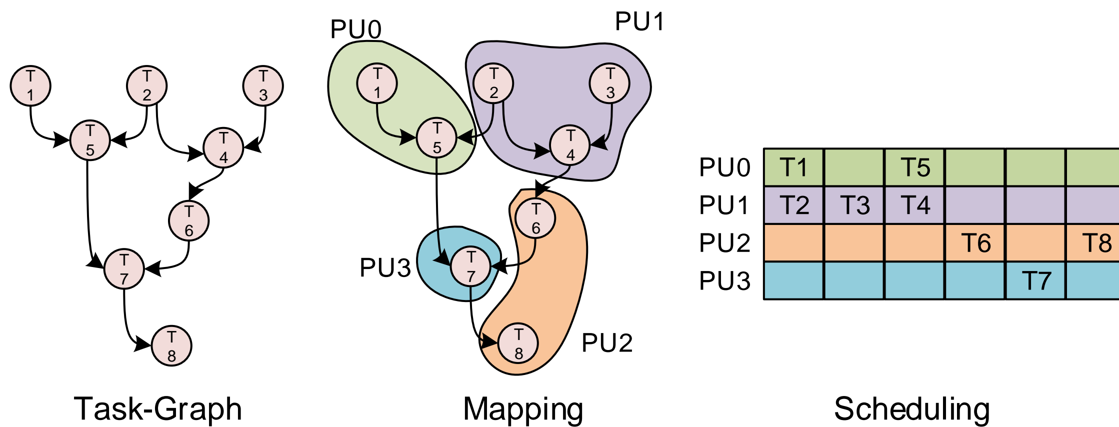


Figure 6.1: An example of task-graph mapping and scheduling.

The processing cost depends on different factors:

- The task itself, its complexity and particularities.
- The processing unit responsible for the task. In a heterogeneous architecture, different types of processing units could be used.
- The matching among the task functionality and the processing unit.

The communication cost depends on some other factors:

- The amount of data to be transfered.
- The channel bandwidth.
- The communication structure (Bus, Point-to-Point (P2P), Network-on-Chip (NoC),etc).

The goal of the mapping process is to optimise the task assignment to the processing units. This optimisation can be for speed, power consumption, fault-tolerance, or any design constraint (even for combinations of them). In a higher abstraction level, in the area of graph analysis, the task mapping problem can be viewed as a *Clustering* problem. The clustering process tries to classify the tasks into groups called *clusters*. In Figure 6.1, the clustering/mapping was performed to four different Processing Unit (PU).

The scheduling process takes the already clustered task-graph and defines when each task should be executed, to maintain the algorithm's semantic. This process is based on data dependencies and availability. In Figure 6.1, the scheduling resulted in six time slots to process the complete algorithm.

6.2 Methodology's Overview

The proposed methodology can be seen in Figure 6.2. Starting from application's description (the *Tile-Code*), techniques for static code analysis, together with spatial considerations, are used to extract the parallelism and to create Task-Graphs and intermediary source code.

A Task-Graph Clustering technique, based on a library of hardware models, is used to create a SystemC/TLM2.0 description of a many-core architecture for each IP/CV algorithm. If an application needs to use different algorithms, simultaneous Task-Graphs are analysed, to generate a more suitable and flexible architecture. Simulations with different combinations of the TLM timing models are used to refine the SystemC/TLM2.0 description into more detailed models of the PEs and the communication structure, aiming to specify the Register Transfer Level (RTL) description.

The rest of the chapters in this part of the thesis are dedicated to explaining the development of the following tools:

- **Task-Graph Creator** - it is responsible by parsing the input C program and

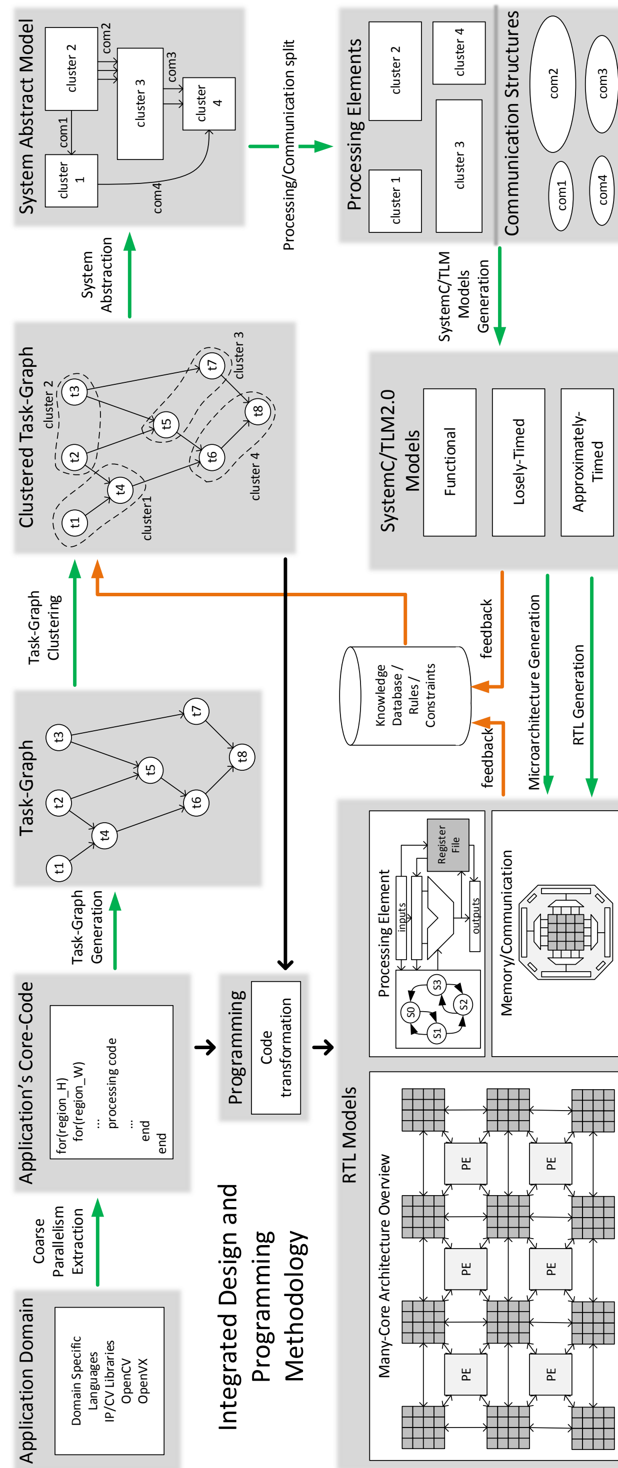


Figure 6.2: The High-Level methodology [MKH15a].

creating the *Task-Graph*.

- **Task-Graph Cluster-Generator** - it is responsible by clustering the *Task-Graph* based on a database of hardware models.

- **Task-Graph Simulator** - it is responsible for scheduling the execution of each

task, depending on the hardware models desired. It can work with both **LT** and **AT** timing models. It also monitors all the events during the simulation.

Chapter 7

Task-Graph Generator

In this chapter, we show the development of the *Task-Graph Generator*, a tool which aims to analyse a program's source-code and generate a task-graph.

7.1 Introduction

The goal of this thesis is to provide a methodology able to help to define a complex hardware/software architecture. To have enough details and be able to determine the architecture, we need to analyse the algorithms in a low abstraction level. We do not have the hardware architecture defined at this moment, which implies that it is not possible to perform an analysis using *Basic Blocks* at a processor's instruction level. We must analyse an algorithm description, a program [6].

Program analysis is a vastly studied area, which includes algorithms, compilers, programming languages, among others. For the sake of simplicity, we have chosen to describe the *Tile-Codes* in the C programming language. Our choice is based on the fact that C is still the most used programming language, with a lot of legacy and knowledge available. Besides, it is also supported by most of the high-level synthesis tools, which can be useful in our project, as we will show later on.

To analyse the *Tile-Codes*, we chose an abstraction called *Task-Graphs*, which

[6] We are considering *Program* as the standard description of an algorithm, e.g. source-code, Domain-Specific Language (DSL)s, diagrams, and so on

has several interesting properties. The basic principle is to distribute parts of a task-graph to be processed in separated Tiles. This partition is performed by a clustering algorithm, as explained in Section 6.2.

A task-graph is composed of nodes (the tasks) and edges (connections among the nodes). These edges represent the data and the control flow of an algorithm. We can classify the edges into two types:

- **Internal Edges:** they represent the internal processor communications.
- **External Edges:** they represent the exchange of information among two Processing Unit (PU)s.

The manual generation of task-graphs is complicated and error-prone, even for simple programs. Despite being a well-studied area, we could not find in the literature an up-to-date and open-source tool able to extract task-graphs from programs written in high-level structured programming languages, e.g. C/C++/Pascal. On the other hand, there are several tools able to generate random task-graphs. Due to the lack of a suitable tool, it was decided to develop a task-graph generator tool from scratch.

Even though the C language was developed more than 40 years ago, it is still one of the most used programming languages, with plenty of legacy codes in different application domains. Also, it is a quite large language, and due to its complexity, we reduced our scope to just a set of the language. The main restrictions are that we do not support pointers or unconditional jumps (*goto* statements). Loops are supported only for the analysis of data dependencies.

The *Task-Graph Generator* was developed as functionality within the LLVM compiler framework, with the Clang C-Compiler as front-end [LA04b]. We must highlight that, at this point, we are not developing a compiler, but a program analysis tool, which shares several structures with common compilers. We use a specific library called *libTooling*, which simplifies the implementation of programs that need a compiler front-end.

Besides LLVM and Clang, we used the Boost Graph Library (BGL), which is a well-known library with support for different graph-based computing [Jer].

7.2 Indexing of Pixels and Processing Elements

The Image Processing and Computer Vision (IP/CV) algorithms studied in this thesis work on image pixels. Indexing the pixels is of high importance to maintain the semantic of the algorithms and provide correct processing results. On the other hand, considering the *General Tiled Architecture*, depicted in Section 5.3, the tiles are organised in a 2D array. To force the coherence of the addressing scheme (and, consequently, all memory accesses), we use three addressing modes simultaneously.

The pixel addressing uses Cartesian coordinates, with the starting pixel at the upper-left corner and the last pixel at the lower-right position. The tiles are indexed in the same way, with a significant difference: each tile has the same index as the starting pixel of its region.

For an image with $M \times N$ resolution, the absolute coordinates are in the range from 0 to $(M-1)$ and from 0 to $(N-1)$, where M is the number of pixels in width and N is the number of pixels in height.

Also, to facilitate the memory management within a tile, the pixels are addressed in the same way: to the upper left corner we assign the relative coordinates (0.0), and to the lower right corner we assign the coordinates $((m-1),(n-1))$. To separate the global and the local pixel addressing, we denote by m and n the width and height, respectively, of a tile image in pixels.

Figure 7.1 shows an example of the addressing modes, considering each tile with 2×2 pixels of resolution. Highlighted in the picture is the starting pixel of each tile.

From an algorithmic perspective, another addressing mode should be used: by the definition we gave in section 5.3, the *Tile-Code* is the program which generates a single pixel as output. The third addressing mode is relative to the position of the pixel of interest. Like in the other modes, we use a Cartesian coordinate system with the center in the pixel of interest and all the other image pixels have their

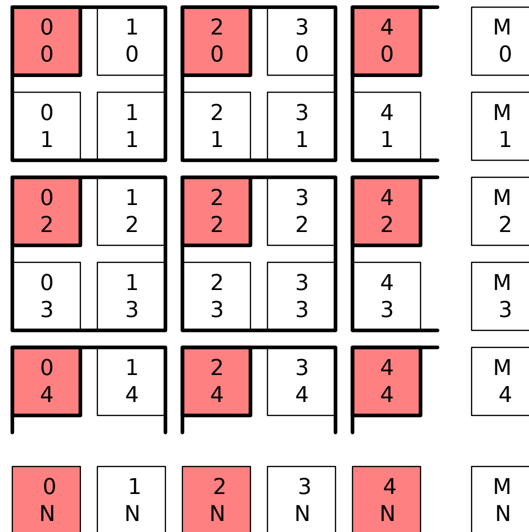


Figure 7.1: Display of absolute numbering of pixels and tiles [Fri15].

coordinates determined relative to the centre pixel. This means that to some pixels we will assign negative coordinate values. Figure 7.2 shows an example with 4×4 tiles, with the pixel of interest at the upper-left corner of the highlighted tile.

7.3 Tools and Libraries used

7.3.1 The Boost Graph Library

Since we intend to work with graph manipulation, we have searched for tools offering classes, structures and methods to ease handling the task-graph analysis. The BGL is a complete C++ library able to create, store and manipulate graphs. Furthermore, the BGL also offers several data structures for graph storage, common search algorithms, and complex graph-based methods (e.g. the identification of sections, trees, rivers, and so on). The BGL is supported by a large community, providing extensive documentation and it is strongly based on templates, which can be adapted and customised for our purposes. In addition, it provides the visualization of the graphs based on standards like *Graphviz* [GN00] and *GraphML* [Bra+02]. Figure 7.3 shows a basic class hierarchy for graph creation using BGL.

-m -n	...	-2 -n	-1 -n	0 -n	1 -n	...	m-1 -n	m -n	m+1 -n	...	2m-1 -n
...	...	...	...	...	...	...	...	...	...	...	...
-m -2	...	-2 -2	-1 -2	0 -2	1 -2	...	m-1 -2	m -2	m+1 -2	...	2m-1 -2
-m -1	...	-2 -1	-1 1	0 -1	1 -1	...	m-1 -1	m -1	m+1 -2	...	2m-1 -1
-m 0	...	-2 0	-1 0	0 0	1 0	...	m-1 0	m 0	m+1 0	...	2m-1 0
-m 1	...	-2 1	-1 1	0 1	1 1	...	m-1 1	m 1	m+1 1	...	2m-1 1
...	...	...	...	...	...	...	...	...	...	...	...
-m n-1	...	-2 n-1	-1 n-1	0 n-1	1 n-1	...	m-1 n-1	m n-1	m+1 n-1	...	2m-1 n-1
-m n	...	-2 n	-1 n	0 n	1 n	...	m-1 n	m n	m+1 n	...	2m-1 n
-m n+1	...	-2 n+1	-1 n+1	0 n+1	1 n+1	...	m-1 n+1	m n+1	m+1 n+1	...	2m-1 n+1
...	...	...	...	...	...	...	...	...	...	...	...
-m 2n-1	...	-2 2n-1	-1 2n-1	0 2n-1	1 2n-1	...	m-1 2n-1	m 2n-1	m+1 2n-1	...	2m-1 2n-1

Figure 7.2: Representation of the relative numbering of pixels and arithmetic units [Fri15].

7.3.2 The Clang compiler Front-End

In the task-graph creation process, we assume that the application's description was already functionally verified, which means that the code is free of lexical and/or semantic errors. To create the task-graph, we use the front-end of the Clang compiler, mainly its Abstract Syntax Tree (AST) parser.

Clang provides AST generation from C language family (C, C++, ObjectiveC), which fits our desire to use a subset of the C language. The Clang parser actuates identifying source-code templates to create a tree-shaped graph with data dependencies, statements, expressions and so on.

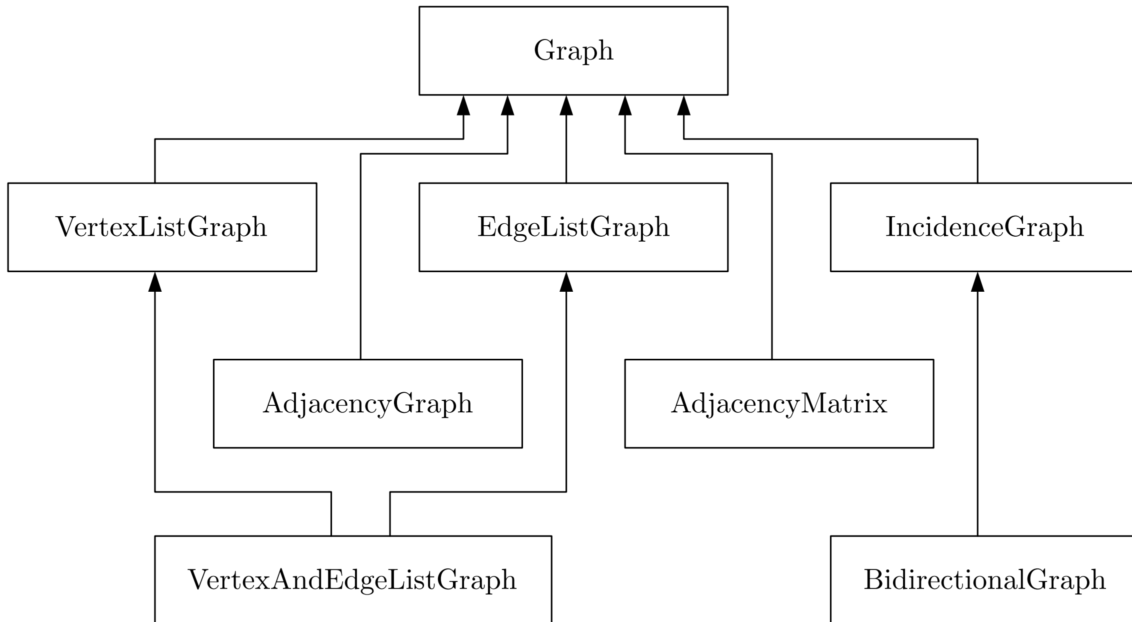


Figure 7.3: Hierarchy of the recipes in the BGL [Fri15].

7.4 Generating the Task-Graph

In this section, we show sample codes for C language structures, illustrating the AST (generated automatically by the Clang front-end) and the *Task-Graph* (generated automatically by our *Task-Graph Creator*).

7.4.1 Literals

A literal is a string used to represent fixed values. In an AST, the node types for literals are Integer−Literal, Floating−Literal, Character−Literal and String−Literal. A literal must be always a leaf in the AST. As an example, let's take a look at Listing 7.1. The code shows a *void* function ex_Literals, which has three literals. The AST is depicted in Figure 7.4, and the Task-Graph (TG) is shown in Figure 7.5. We can highlight that the TG shows clearly that each task is independent from the others.

Listing 7.1: Sample code for literals

```
void ex_Literals()
{
    'f';
    172;
    19.85;
}
```

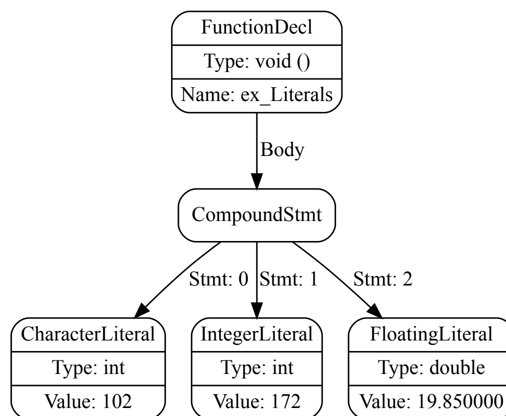


Figure 7.4: AST for the code example 7.1 of Literals [Fri15].

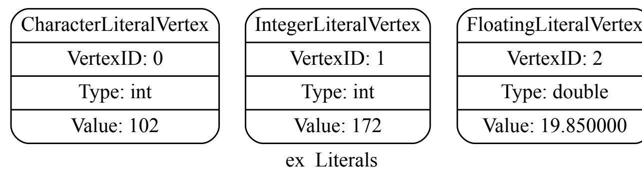


Figure 7.5: Task-Graph for the code example 7.1 of Literals [Fri15].

7.4.2 Variable Access

In Listing 7.2, two functions are shown. The first function, ex_ParmVarDeclRef, illustrates a function with a parameter alpha which is returned by the function. The second function, ex_VarDeclRef, illustrates a function without parameters, but with an internal variable alpha which is returned by the function. Looking into Figures 7.6 and 7.7, we can see that the AST are partially similar (the return statement of variable alpha). On the other hand, when we look to the TG in Figures 7.8 and 7.9, we can see that both representations are almost the same, with the difference in the

type of the of vertex, denoting that in the first case it is a literal, and in the second case, it is an input. For the analysis we intend to perform, the TG representation is more clean and straightforward.

Listing 7.2: Example code for variable retrieval

```
int ex_ParmVarDeclRef(int alpha)
{
    return alpha;
}
int ex_VarDeclRef()
{
    int alpha = 25;
    return alpha;
}
```

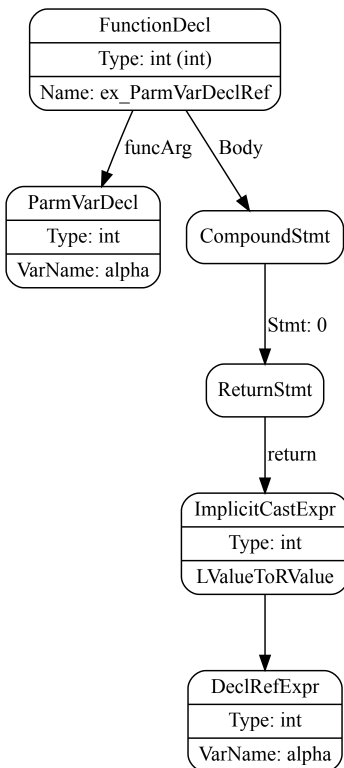


Figure 7.6: AST for function parameters access [Fri15].

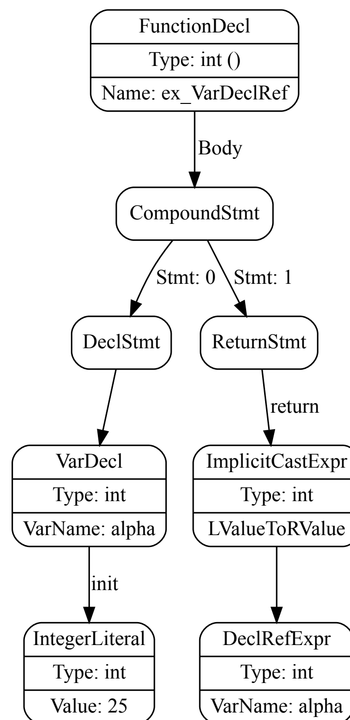


Figure 7.7: AST for variable access [Fri15].

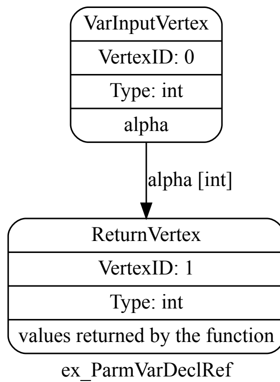


Figure 7.8: TG for function parameters access [Fri15].

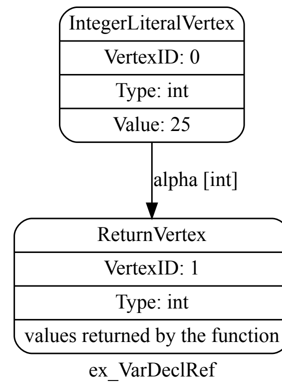


Figure 7.9: TG for variable access [Fri15].

7.4.3 Unary operators

Listing 7.3 shows two versions of an increment operation (pre and post increment), as well as a unitary subtraction, all examples of *unary operators*. In addition, a *Binary Operator* (explained in the next topic) is used in the return statement. When we compare the AST (Figure 7.10) with the TG (Figure 7.11), we can see that the TG representation is easier to interpret, due to the concise information presented. An important aspect to point out is the clear parallelism that can be extracted from the TG. The two ++ statements in Listing 7.3 are independent, thus computed in parallel, receiving the input vertex values. Then the – is computed in parallel to the ∗ operation, which generates the data needed in the ReturnVertex.

Listing 7.3: Sample code for unary operators

```
1 int ex_UnaryOp(int alpha, int bravo)
2 {
3     -(++alpha);
4
5     bravo++;
6
7     return alpha * bravo;
8 }
```

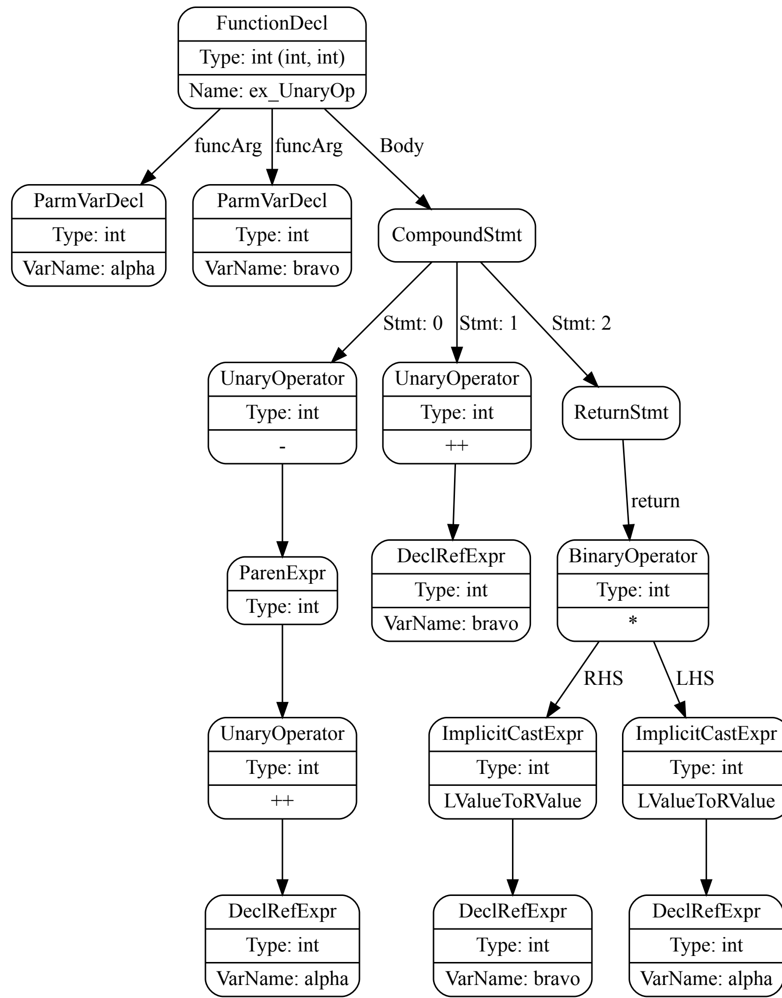


Figure 7.10: AST for the code example 7.3 for unary operators [Fri15].

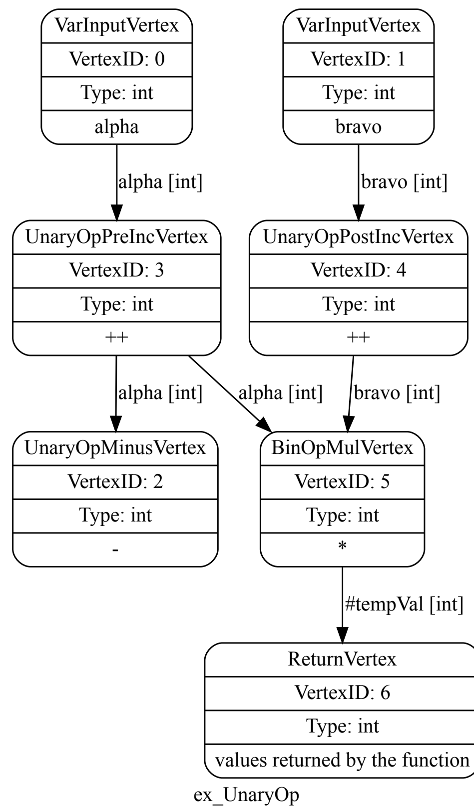


Figure 7.11: Task-Graph for the code example 7.3 for unary operators [Fri15].

7.4.4 Binary operators

Listing 7.4 shows the code example for three binary operators, using three function parameters as inputs. The function ex_BinaryOp does not return any value. The first statement, line 3, shows an operation whose output is not assigned to any variable (probably it would be later eliminated by compiler optimisations). What is important to highlight here, is that in the TG representation, we cannot identify completely line 5 of the code. It is not clear that to the variable charlie is assigned the & operation between alpha and bravo. This happens mainly because of the absence of pointers, or a return statement. Observing the code, we can see that it operates locally over the variables, but all data is then lost.

Listing 7.4: Sample code for binary operators

```
void ex_BinaryOp(int alpha, int bravo, int charlie)
{
    alpha + bravo;
    bravo *= charlie;
    charlie = alpha & bravo;
}
```

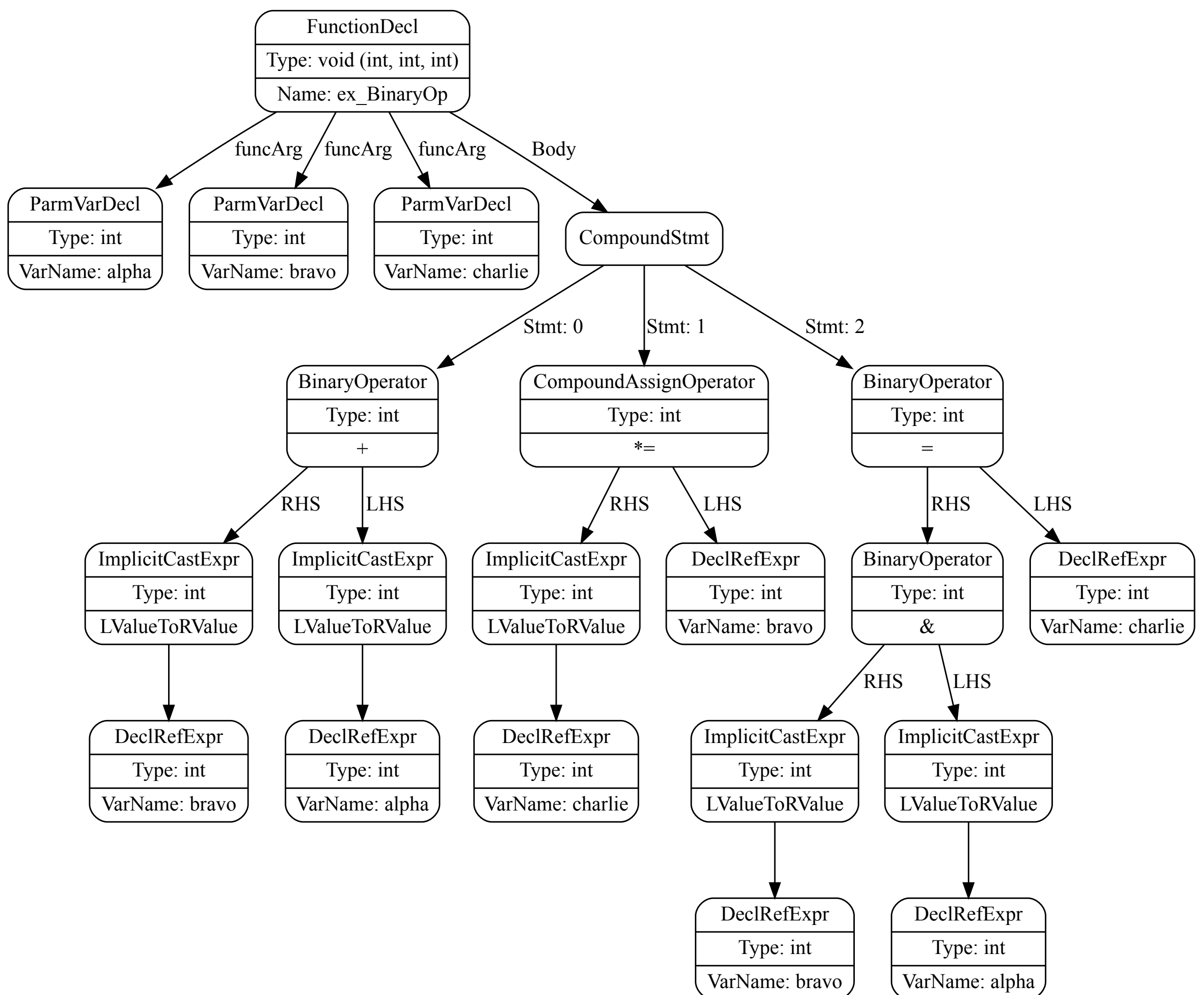


Figure 7.12: AST for the code example 7.4 to binary operators [Fri15].

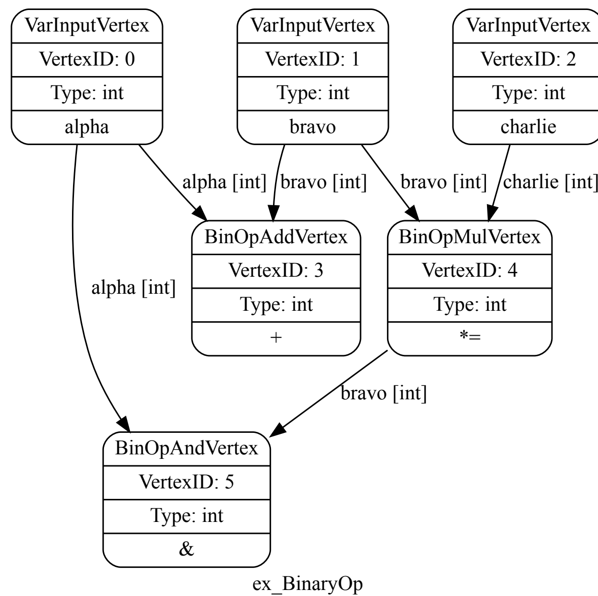


Figure 7.13: task graph for the code example 7.4 to binary operators [Fri15].

7.4.5 Ternary operators

The representation of the code example for the conditional expression (Listing 7.5) in AST is illustrated in Figure 7.14, the resulting task graph in Figure 7.15. The child nodes in the AST become dependencies in the task graph. The node of the conditional operator in the task graph serves as the source for the value generated, depending on the condition.

Listing 7.5: Example code for conditional expression

```
int ex_CondOp(int alpha, int bravo, int charlie)
{
    return alpha > 5 ? bravo : charlie;
}
```

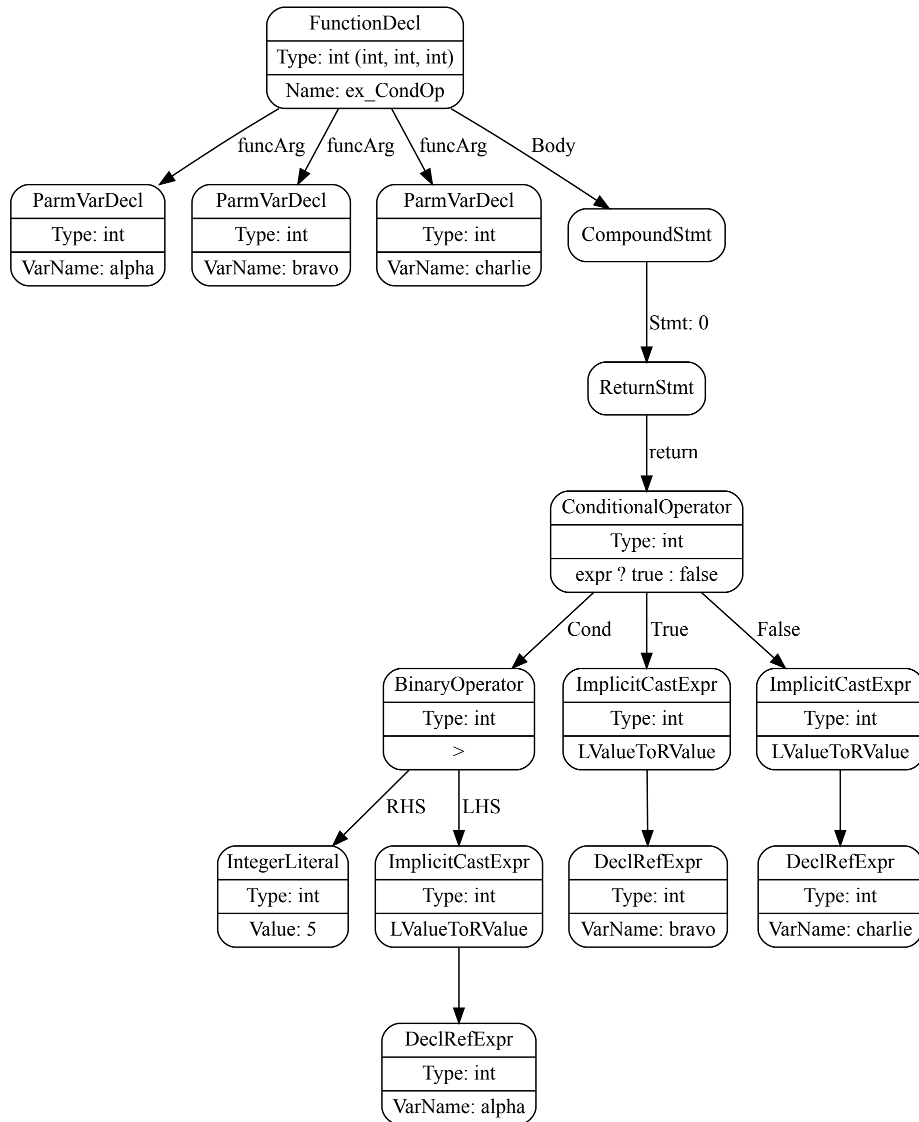


Figure 7.14: AST for the code example 7.5 to conditional expressions [Fri15].

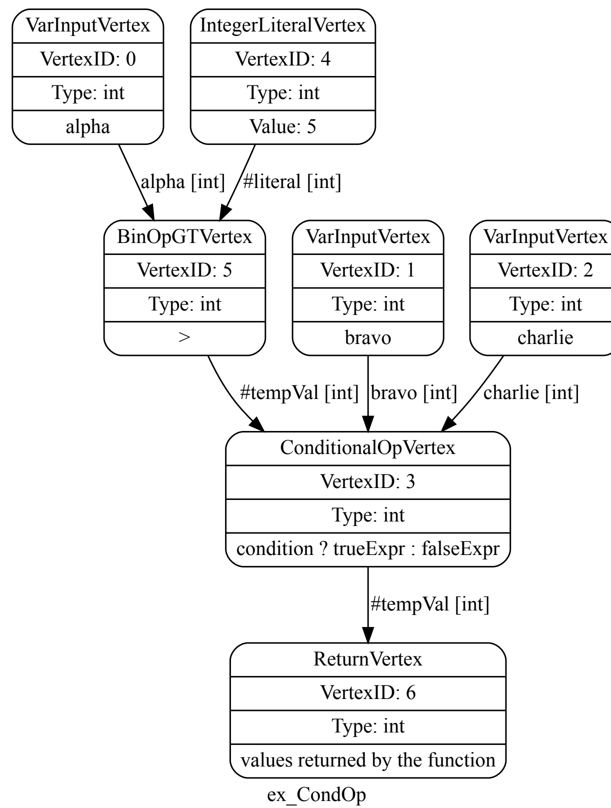


Figure 7.15: Task graph for the code example 7.5 to conditional expressions [Fri15].

7.4.6 Function calls

The comparison of the representation of function calls can be seen in the listing 7.5, the AST in Figure 7.16 and the task graph shown in Figure 7.17. The function arguments become dependencies of the function call at whose node the return value is available in the task graph.

Listing 7.6: Sample code for function call

```
int funnyFunc(int alpha, int bravo, int charlie);

int ex_FuncCall(int delta, int echo, int foxtrott)
{
    return funnyFunc(delta, echo, foxtrott);
}
```

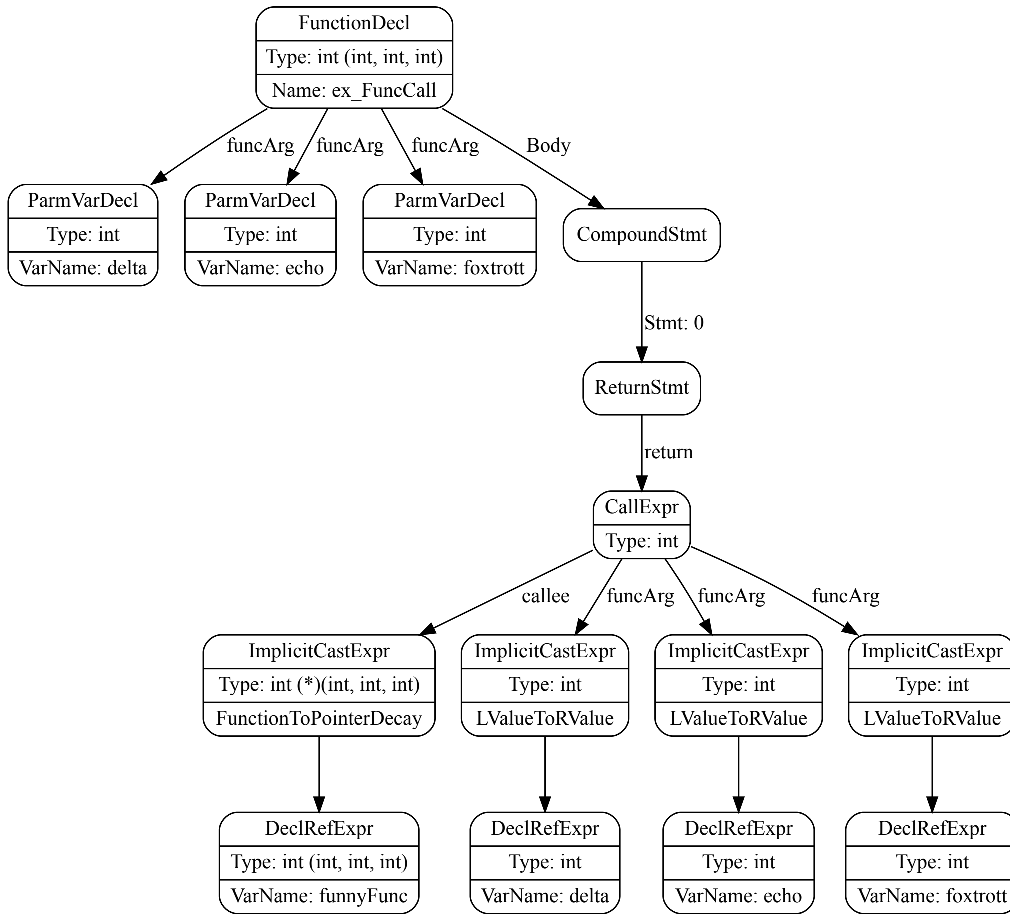


Figure 7.16: AST for the code example 7.5 to call the function [Fri15].

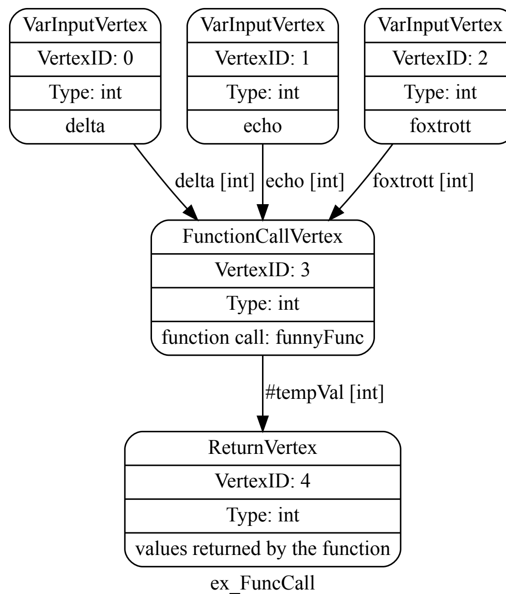


Figure 7.17: Task graph for the code example 7.5 to the function call [Fri15].

7.4.7 Control Flow

Table 7.1 shows the control flow constructions available in the C language. In this chapter, we are performing a static code analysis. Therefore, control constructions must have all its possibilities analysed since there is no dynamic execution path. The vertex in the TG related to the control flow is dependent on all values that are read within the control flow block. When we use such representation, it allows for a hierarchical simulation, which is important for the clustering process.

Table 7.1: Control flow constructions.

if-then-else	do-while loop
switch-case	for loop
while loop	return statement

For the sake of simplicity, we will show here only the cases for the *if-then-else* and *return statement*.

if-then-else Listing 7.7 shows the sample code of the *if-then-else* branch construction. The variable bravo can be returned with its original value (5, in the code) or a value dependent on the input parameter alpha. In Figure 7.19 we can see that the condition is analyzed before the IfBeginVertex, which receives only the comparison result. The two branch possibilities, *then* and *else* are derived from vertex 6 and converge to vertex 7, which represents the end of the branch construction.

Listing 7.7: Sample code for if branching

```
int ex_IfStmt(int alpha)
{
    int bravo = 5;
    int charlie;

    if ((charlie = alpha >> 3) != 0)
        bravo = bravo + alpha;

    return bravo;
}
```

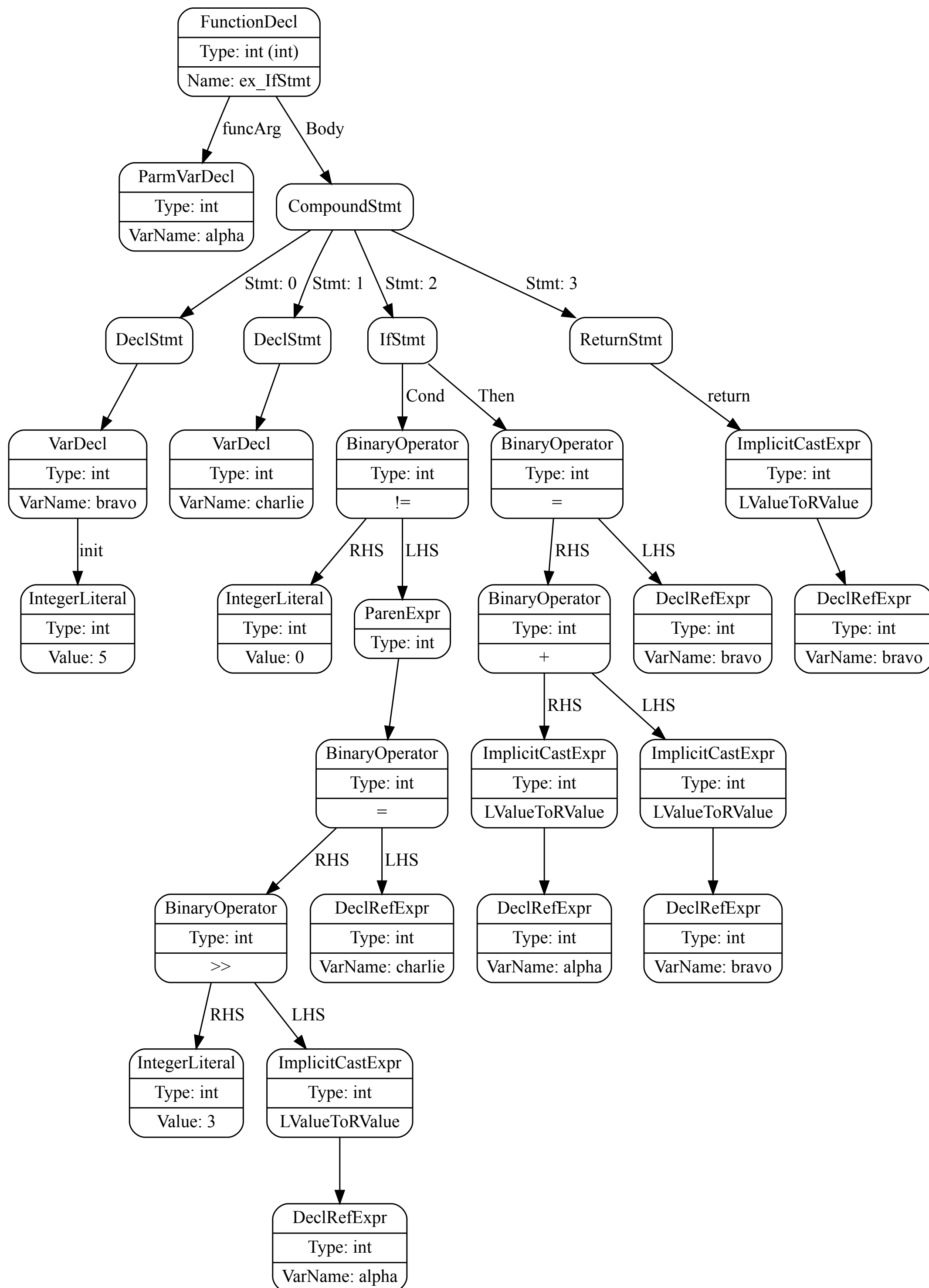


Figure 7.18: AST for the code example 7.7 to the if branching [Fri15].

.

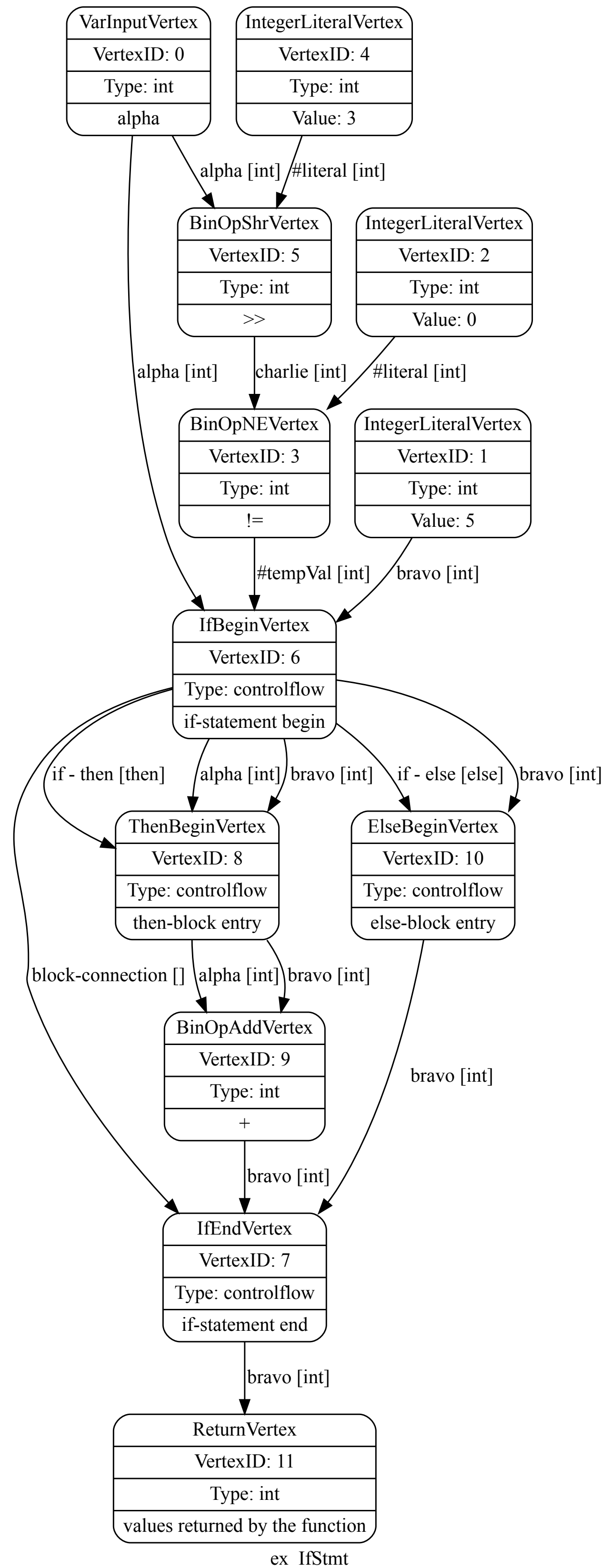


Figure 7.19: Task-Graph for the code example 7.7 to the if branching [Fri15].

.

Return statement In the C programming language, the *return* statement is used to exit a function, with both a result or an error identification flag. A single function can have an arbitrary number of return statements. However, they must exclusive. In our implementation, some constraints must be observed:

- Each function may have only one return statement.
- The return statement must be the last statement in the code.

Chapter 8

Task-Graph Clustering

In this chapter, we develop the *Cluster Generator*, which is responsible by clustering the *Task-Graph* based on a database of hardware models.

8.1 Introduction

The primary utilisation of clustering methods is on data classification. These methods are used in several areas, and they can use different classification methods [Sch07]. We must highlight that graph clustering is only one of this methods since data can be represented in different ways and still be clustered/classified. In our approach, we selected the *Force-Directed Clustering* because it can be viewed as a geometric problem, as our goal of allocating tasks to spatially distributed Processing Unit (PU)s, and it has been used for task allocation by different researchers in the past decade. It is not our goal to discuss/find the best or the most efficient clustering solution, but to determine the importance of clustering/classification methods in the methodology we are developing here.

8.2 Force-Directed Clustering

As a basic reference for the Force-Directed Clustering (FDC) method, we have followed the approach described in [PS08]. In this section, we describe the method,

using as an example the program in Listing 8.1, whose Task-Graph (TG) is depicted in Figure 8.1. Parameters and literals are the task-graph inputs, and the vertexes are basic C language operations. The edges represent dependencies, data-flow, and control-flow. For the control-flow, there is a hierarchy of defined entry and exit points. The graph output is the return value.

Listing 8.1: Sample code for if branching

```
int ex_clustering(int a, int b, int c)
{
    return (a + b) / c;
}
```

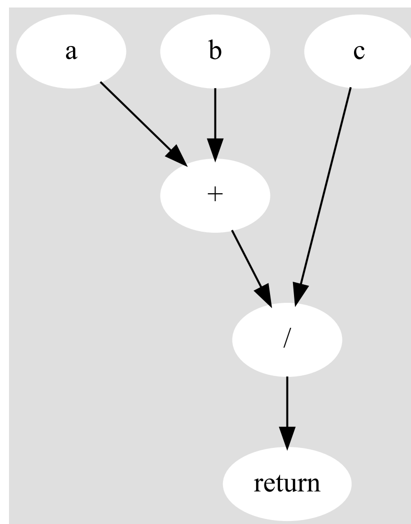


Figure 8.1: TG for the program in Listing 8.1 [Fri15].

The FDC method is specially used in algorithms for drawing graphs. It considers all nodes of a graph as charged particles, and all edges as springs, as shown in Figure 8.2. The model assumes that attraction and repulsion forces are acting over the graph nodes. Repulsion is based on the charge and distance of each node, and attraction is based on the spring forces. The primary goal of this method is to find a state of minimal energy in the physical system.

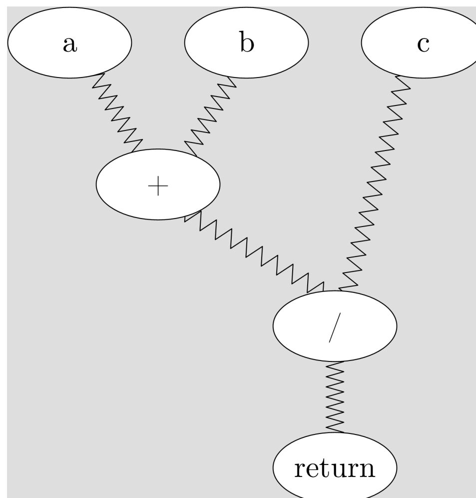


Figure 8.2: A graph represented using charged particles and springs [Fri15].

The algorithm is based on the following steps:

1. Assign to each graph's node a starting position.

2. Based on the current position, the forces acting on each node are then determined.

3. The new position of each node is then calculated, and the nodes are repositioned.

An important aspect is that this is a static method. Therefore, we do not consider dynamic effects such as vibrations and rotations. To speed-up the algorithm, a simplification is done, avoiding the use of differential equations.

To combine the strength of repulsion and attraction forces with the properties of the graph, edge weights are used to specify each spring elasticity. The weight of each node determines the charge of the respective particle. Using this model, it follows that edges with high weight values represent connections (springs) hard to stretch, meaning that the connected nodes are more likely to be placed close to each other.

In the case of the TG, high edge weight values mean that the nodes connected would be more likely to be assigned to the same cluster. On the other hand, nodes

with high charge values lead to strong repulsion forces. Therefore, these nodes are less likely to be assigned to the same cluster.

The position determination algorithm consists of the following steps:

1. determine the repulsion and attraction forces from the valid positions.
2. calculate the new positions from the forces determined.
3. repeat the steps until the threshold value for the energy is not exceeded, or the maximum number of iterations is reached.

The maximum number of iterations and the threshold value can be set by the user. The system energy in an iteration step is determined by summing the amounts of all forces used to change the positions.

After the positions have been determined, the clustering of the graph is then created, in the following steps:

1. Arrange the edges according to their length (the spatial distance of connected nodes).
2. Starting from the shortest to the longest edge, determine if the two nodes should be clustered together or not, based on a force-threshold.
3. Merge(or not) the nodes.
4. Calculate the sum of communication costs (energy) for the entire graph.
5. If the new energy value is smaller than in the previous iteration, we have a new cluster set. Otherwise, repeat the steps for the next edge.

8.3 Clustering in the multi-core architecture

Considering the *General Tiled Architecture* discussed in Section 5.3, we can now determine some constraints to the clustering method, to adapt it to our analysis.

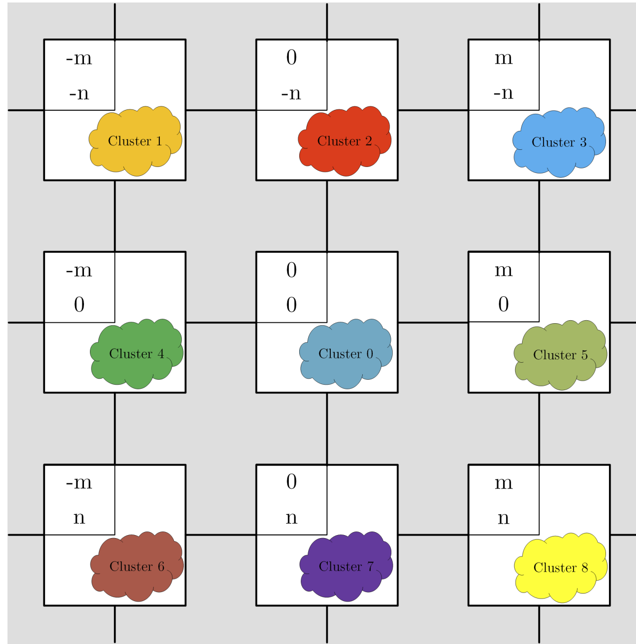


Figure 8.3: Cluster assignment for the *General Tiled Architecture* [Fri15].

Figure 8.3 shows that we assign a cluster to each tile. In this case, we are analysing the clustering method with a single *Tile-Code* to be assigned to the architecture. *Cluster 0* represents the cluster where the pixel of interest is located.

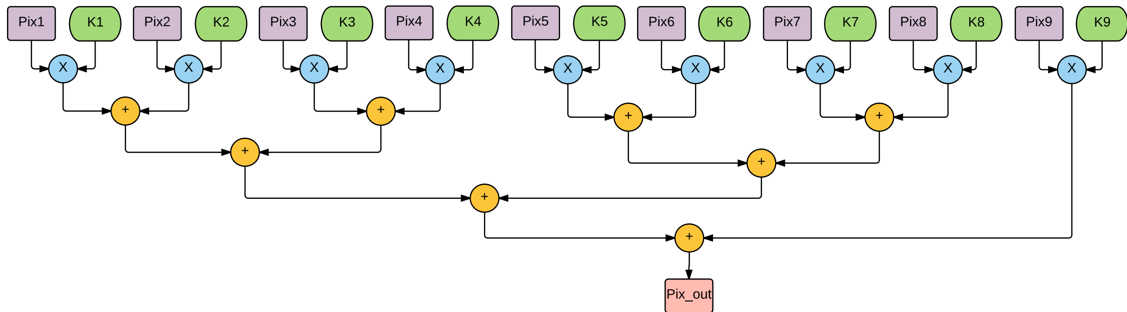


Figure 8.4: 3×3 convolution example .

Let's consider a simple Image Processing and Computer Vision (IP/CV) algorithm, like the 3×3 convolution from Figure 5.7, replicated in Figure 8.4, and an architecture with 2×2 pixels per tile. Figure 8.5 shows the pixel of interest (0,0) and the surrounding neighbourhood needed to perform the algorithm. We can see that four pixels are in the same tile of the central pixel, and five other pixels are in other tiles.

The constraints applied to the clustering algorithm, considering the underlying

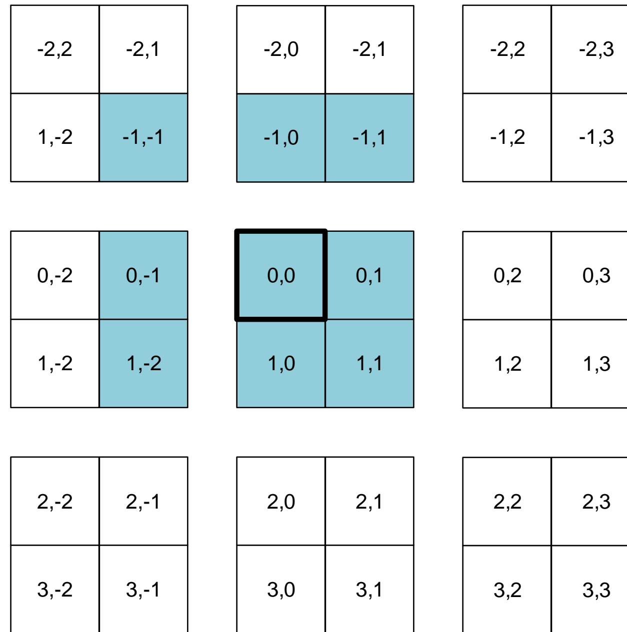


Figure 8.5: Basic pixel assignment for the sample architecture.

architecture are listed as follows:

- The input pixels are in fixed allocated to their original tiles. This means that the corresponding nodes do not move during the clustering process.
- The other inputs, coefficient values, are movable within the clustering.

8.4 Clustering Results

This section uses the *Task-Graph Creator* and the *Task-Graph Cluster Generator* tools, to perform an analysis of two IP/CV algorithms. A discussion about the results obtained and further enhancements are provided at the end of the section.

8.4.1 Convolution algorithm

We start the analysis with a 5×5 convolution algorithm, which resembles the algorithm in Figure 8.4, but with a larger neighbourhood. We have chosen this example because of its simplicity and representativeness of a larger family of IP/CV algorithms. Furthermore, this algorithm requires 25 distributed pixels, so that the

effects of the connection structure and programming styles can be well-represented here.

One important aspect to be highlighted is that the quality of the source code has a strong influence on the clustering results since the non-optimized code will generate larger task-graphs. This is not directly related, however, to the number of lines the code has. For example, the algorithm in Figure 8.4 has eight additions. If these additions are written as a single line in the source-code, the generated *TG* will have a tree of eight concatenated addition blocks, with a long critical path. The exploration of associativity and commutativity of operations are not automated in our tool and must be formulated by the programmer/user.

In the algorithm created for convolution, the products of the pixel values are first calculated with their weights. The results of the products are then added up. In the algorithm used for the experiments, the sums were formulated according to the structure represented in Figure 8.6. The red arrows represent the transmission of the calculated data. In this example, we considered the existence of horizontal and vertical connections among the tiles, with no diagonal connections.

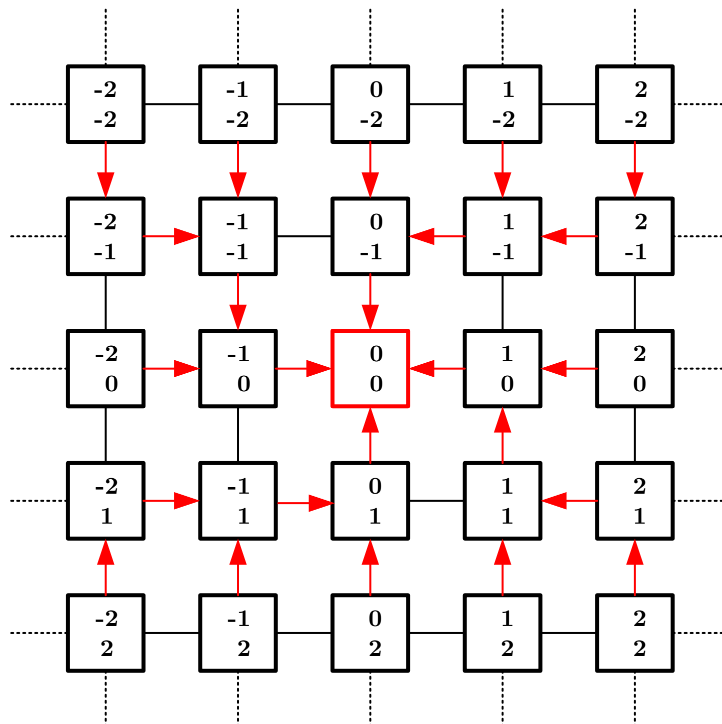


Figure 8.6: Communication structure of the computing units with manually implemented convolution [Fri15].

The use of the shown communication structure is based on the fact that precise communication paths, which in each case only require directly accessible neighbours, are given. This makes it possible to test whether the clustering algorithm combines useful nodes into clusters. It is also necessary to find out to which extent the boundaries of the clusters are bordered by nodes that are connected to other clusters via favourable edges that only require one communication step. To obtain a reference for the clustering results, clustering has been created manually, where the arithmetic operations are distributed as shown in Figure 8.6. The nodes with incoming arrows add the values they receive to their value and pass on the overall result.

Based on the same graph, clusters were subsequently created using the *Force-Directed Clustering* technique. The step size, which parameters are needed for the weighting of the edge weights, the spring length and the termination conditions were changed. The resulting structures are in all cases of worse quality than the manually created pattern. The positioning step produces comprehensible results in which even changed weights of nodes and edges can be recognised.

The actual clustering step and in particular the related quality measure still have considerable potential for improvement for the selected application. In particular, the fact that once assigned clusters remain unchangeable and that each new clustering is assumed when the quality value improves means that no optimal solutions are found. In this case, the use of an optimisation method is recommended, especially to improve the allocation of nodes at the cluster boundaries.

The results generated by the current program can be chosen as a starting point for such an optimisation process since significant improvements can often be achieved in the clusters created by small changes. The following example in Figure 8.7 shows the communication structure in automatic clustering. This shows that in many cases non-optimal paths are chosen.

It makes sense to revise the quality criterion in such a way that paths to nearby arithmetic units, as well as paths that do not move away from the target processor, are more preferred. The task graphs for the discussed examples are shown in Figures

8.9 and 8.8.

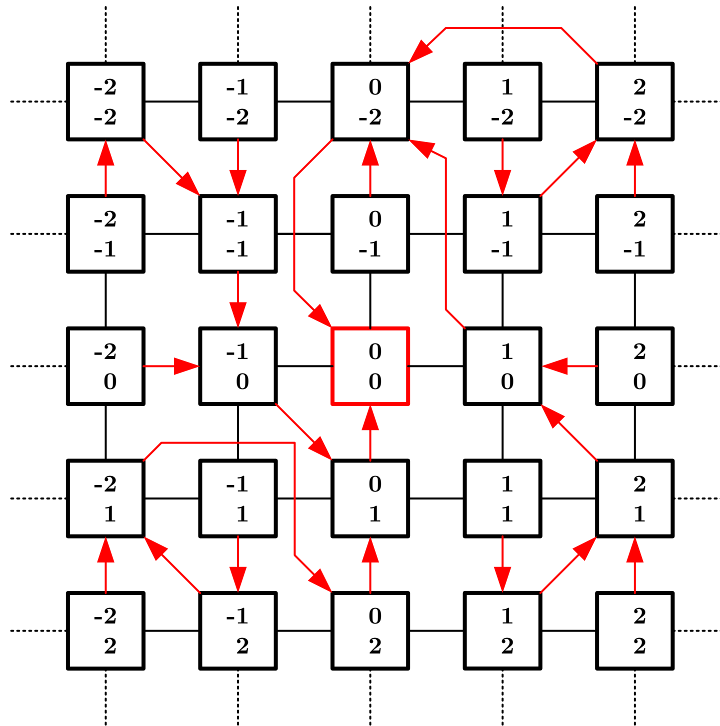


Figure 8.7: Communication structure of the computing units with automatically clustered convolution [Fri15].

To investigate the effects of the parameters in the clustering process, the example shown above was primarily used. The effects of the spring length on the clustering result are not particularly strong. This is because the initial spring length is the same for all edges and the relative distances from which the sorting is carried out do not change significantly.

The parameters for the step size and the weighting of the nodes and edges have a strong influence on the result. It should be noted that the relation of these parameters has a strong influence on the stability of the algorithm. High step widths require very high damping of the weights of the nodes and edges and often lead to unacceptable clustering results.

If the edge weights are over-emphasised, the clusters are created almost exclusively from the connection structure, because the rejection of the nodes has almost no effect. However, this behaviour is more desirable for clustering the task graph in this case of application than a minimal emphasis on the edges or even a strong

emphasis on the node weights.

This behaviour leads to significantly worse clustering results, especially in those cases where it may be desirable that the algorithm is not distributed evenly so that it can be processed at the lowest possible communication effort and high speed. This is not necessarily the case if the processors are working at a constant load because the same algorithm is executed on each processor in the assumed model.

Performing pre-calculations for a neighbour can prove to be quite useful if this reduces dependencies and communication effort. However, it is not desirable to distribute the calculation effort equally among the participating processing units.

8.4.2 Median-Filter

The second IP/CV algorithm analysed is the *Median Filter*. It is also a *Neighborhood Operation*, which sorts the pixel values within the neighbourhood, outputting the median value.

In contrast with the convolution algorithm, the TG in the median filter is strongly connected, with several data exchanges among the tasks. This a result of the C-coding style used but serves as a good example of the clustering technique.

The *TG* is structured in layers with data dependencies among them, in a quite homogeneous flow. For this example, different clusters were created with the clustering technique, as well as a manually partitioned version. Figure 8.10 shows one of the clustering results. We can see that there is a dominance in one of the clusters, which has the highest task count. However, in this example, as well as in the other tested examples (with other weights assigned), the dominant cluster was not the one with the pixel of interest.

Here the question arises whether a change of the algorithm for generating the initial clustering should include a path from the pixels assigned to the computational unit (relative coordinates (0.0)) to the return node to avoid this behaviour.

The parameters have a similar effect on the structure of the graph in this case: due to the very homogeneous network structure, however, no clear groups are formed

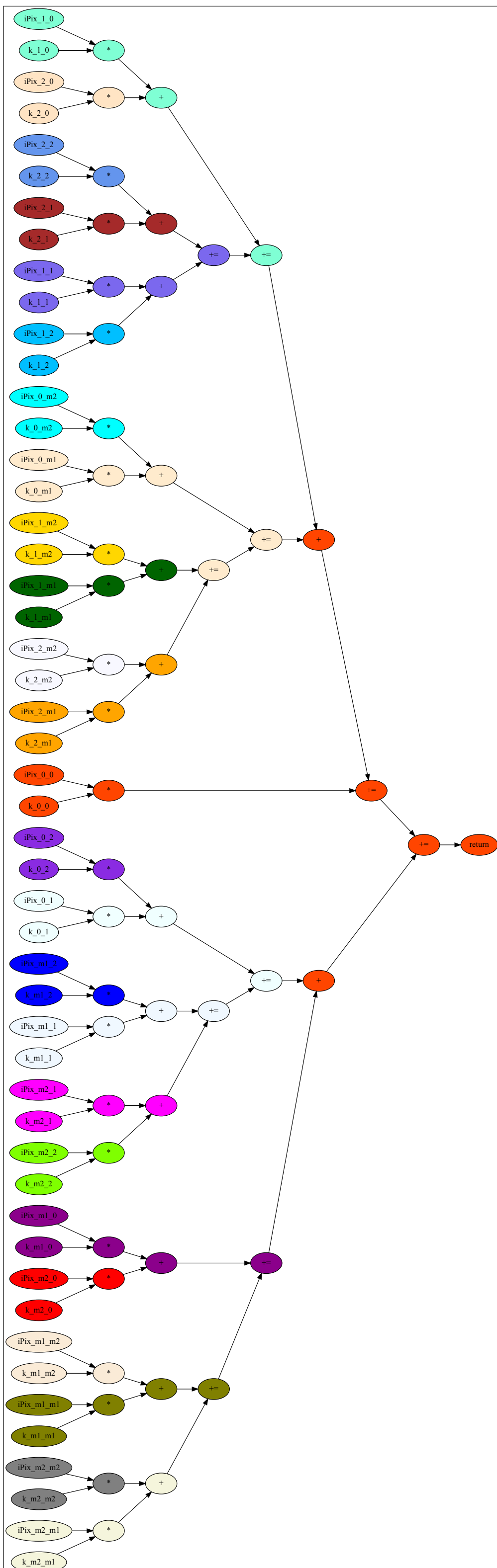


Figure 8.8: Manual Clustering [Fri15].

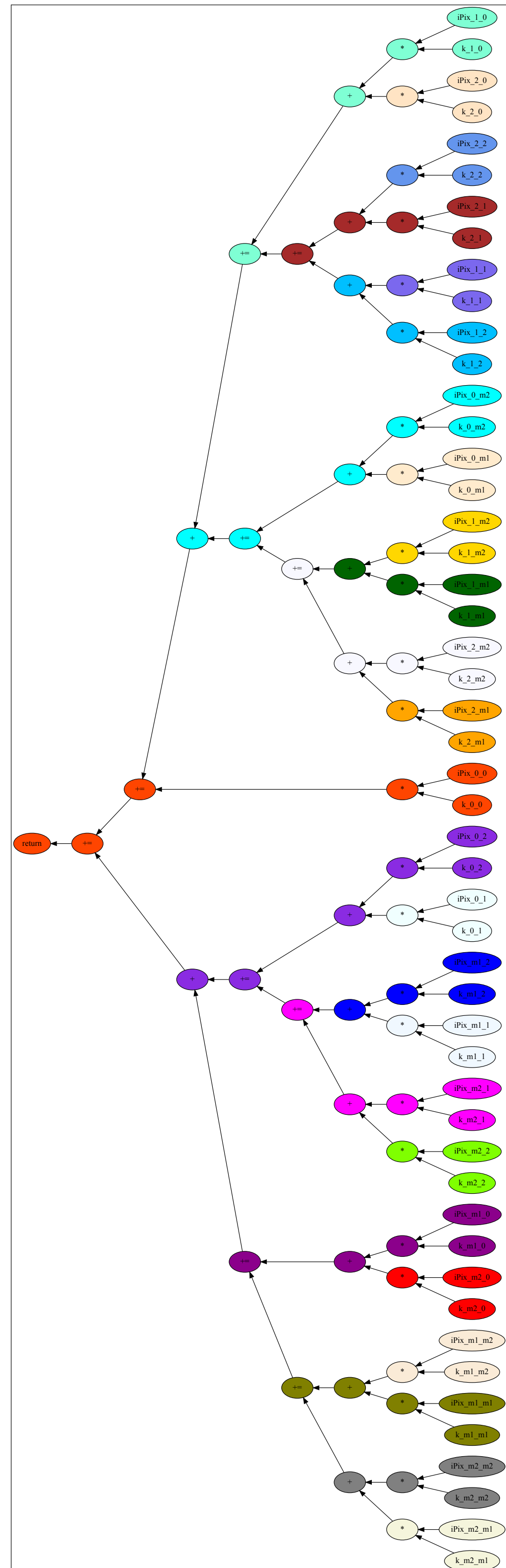


Figure 8.9: Automatic Clustering [Fri15].

as with tree structures. Only the nodes on the extremes (roots and leaves) are strongly influenced by the change of the parameters. The nodes inside the network

are placed similarly in all cases, and the cluster allocation does not differ significantly.

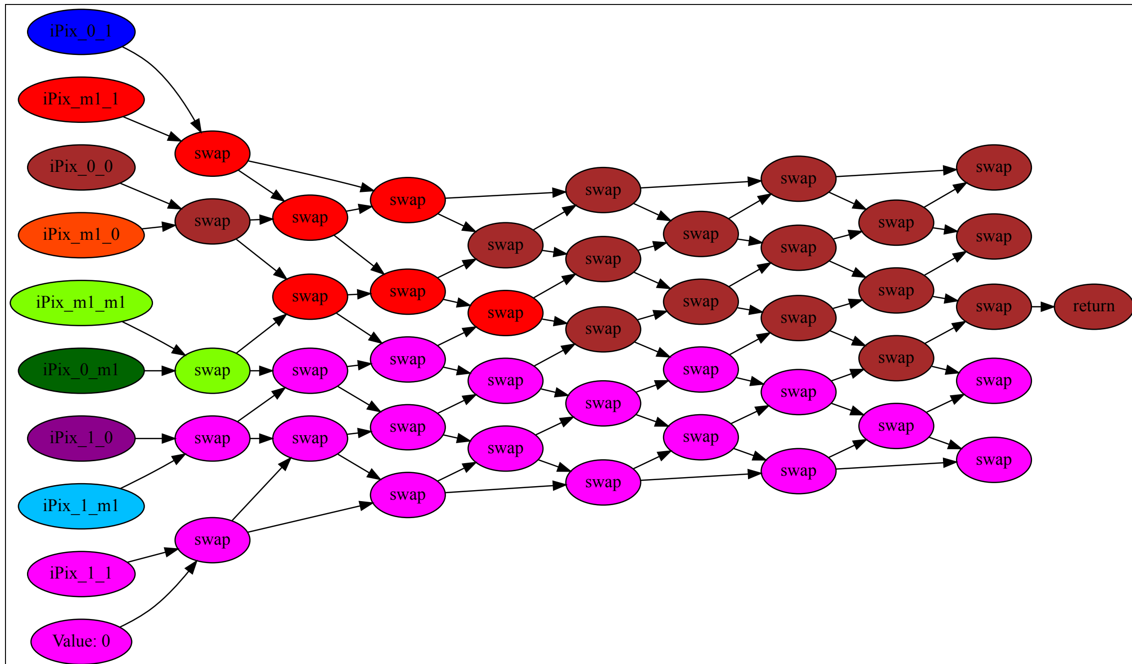


Figure 8.10: Task-Graph for the Rank-Order-Filter with cluster assignment [Fri15].

8.5 Analysis of the clustering method

- The quality of the results is currently still very much dependent on the quality of the generated core code. A useful enhancement is the automatic optimisation of the given source code concerning the processor architecture.
- Within the scope of this work, only one clustering algorithm was investigated using quality criteria which, for simplicity, only considered the communication costs. Inserting the processing costs is a trivial task, and our choice was based merely due to the absence of a complete database of hardware models. The results obtained can be used, but in many cases can be optimised. However, the programs created allow one to easily replace both the clustering algorithm and the costing functions to test other algorithms and the results that can be achieved with them.
- If algorithms are investigated that use input parameters other than input pixels, such as the values of a filter mask, the *TaskGraphCreator* also displays

them as nodes, regardless of whether they are defined as local variables or as function parameters. In clustering, these must be assigned to exactly one cluster, even if they are used in calculations on several processing units. This behaviour occurs when a processing unit has more than one pixel. In this case, communication costs occur which are not necessary because algorithm-specific values, unlike pixels, do not have to be transmitted. A solution to this problem is to create the values for each pixel to be calculated as a separate variable or parameter, which complicates the creation of the algorithms on the one hand and leads to unnecessary use of memory on the other.

- As explained in the description of algorithms and the section for clustering task graphs, all input pixels required for a function are passed in such a structure whose size depends on the number of pixels per processing unit and the number of pixels needed to calculate an output pixel. During the experiments, the idea arose to use only one structure containing the pixels present on a processing unit and as many parameters as neighbouring processors. The structure described above has particular advantages when functions should be concatenated because the return value always contains the pixels of a processing unit. Another advantage is that the information about the relative positions of the related arithmetic units could be used as parameter names. These would then no longer have to be calculated for each pixel from the coordinates. The structure described above has not been implemented in the project but requires only changes in the functions for preprocessing the task graphs in the clustering program and for automatic creation of the model if their use is considered in other projects. In this case, the *TaskGraphCreator* and the clustering algorithm do not have to be adapted.

- The clustering was performed in this chapter using unitary weights assigned to the task links/edges. In the future, we plan to have a database of hardware models, to use costs for area, power consumption and speed to assign the weights. In this direction, we would perform a multi-objective optimisation.

The simulator developed in the next chapters is directly related to the weight assignment. It receives as input the clustered task-graph and creates architecture models based on the clusters. After the simulation, feedback is given to the *Cluster Creator* to reassign weights and perform new clustering. This interaction among the tools is essential to achieve the optimal design desired.

Chapter 9

Task-Graph Simulator

For the investigation of image processing algorithms and their processing on different multi-core topologies, a parameterizable simulator is described in this chapter. The simulator was developed using SystemC and Tlm 2.0, as well as the concepts discussed in the thesis, like the *Tile-Codes* and the *General Tiled Architecture*. It can generate an executable simulation from the *Task-Graphs* created in the previous chapters.

9.1 Introduction

Figure 9.1 shows the basic idea of the simulator presented in this chapter. A task-graph, as discussed previously, is the representation of a *Tile-Code*. Given a clustered task-graph, it could be that the resulting assignment is like shown in the picture. The tile-code is replicated in each tile. Therefore, after the clustering, each tile executes parts (tasks) of different task-graph instances.

This type of mapping is useful for both homogeneous or heterogeneous processing architectures. The simulator accesses a database with different hardware models, to estimate the processing costs. By simulating with different hardware possibilities, the simulator can generate several solutions, which can then be fed back to the clustering part and enhance the cluster results.

In this chapter, we show how we modelled the scheduling of each task, the

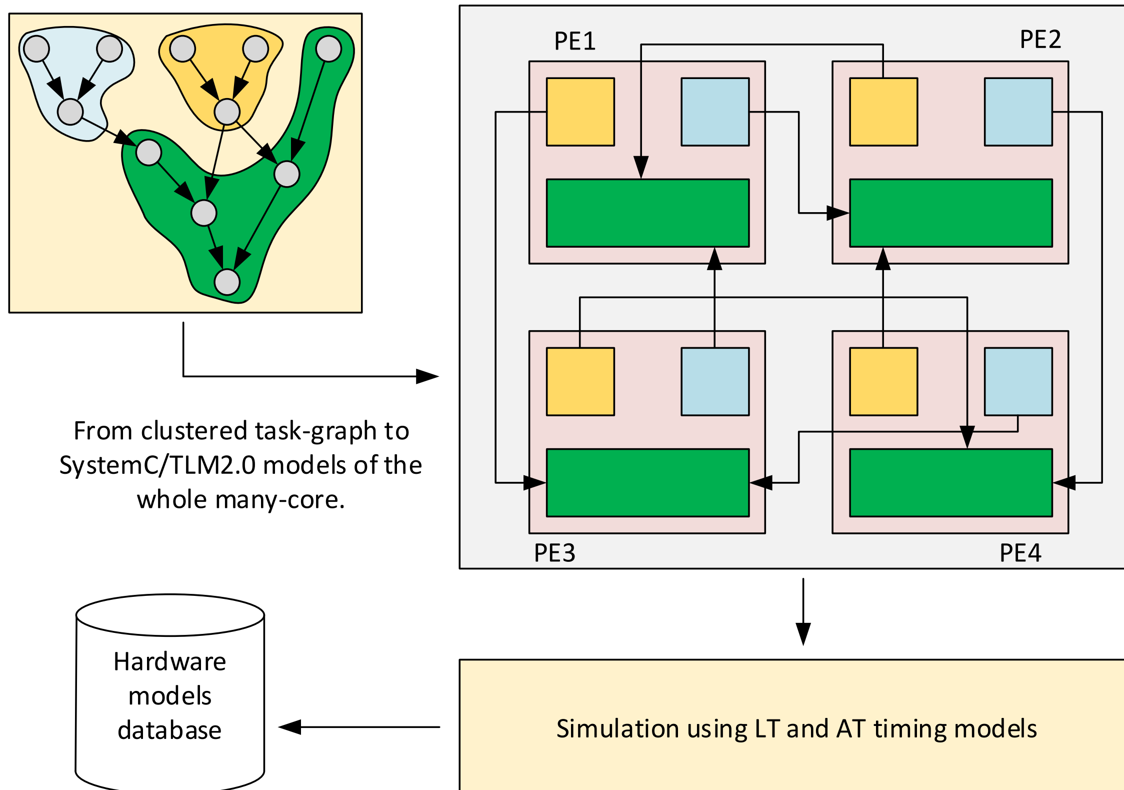


Figure 9.1: From the clustered task-graph to a many-core model

communication processes, and the parameterisation of the simulator.

9.2 Task-Graph interpretation

One of the goals of our simulator is to resemble, as close as possible, the behaviour of a real hardware implementation [7]. Therefore, the simulator must be able to handle sequential and parallel execution of the tasks, like would be the case in real hardware. This section shows how this is achieved by the simulator.

[7]At the TLM level, it is mainly a functional simulation with some timing annotations, depending if Loosely-Timed (LT) or Approximately-Timed (AT) is used.

Listing 9.1: Simple clustered task-graph example

```
int ex_tginterpretation(int p1, int p2, int p3, int k1, int k2, int
    k3)
{
    int tmp1 = k1 * p1;
    int tmp2 = k2 * p2;
    int tmp3 = k3 * p3;
    int pout  = tmp1 + tmp2 + tmp3;
    return pout;
}
```

Figure 9.2 shows the task-graph for the code in Listing 9.1. The Task-Graph (TG) in the picture is already partitioned into three different clusters. Let's define now some concepts and constraints which help the simulator to interpret/understand the TG.

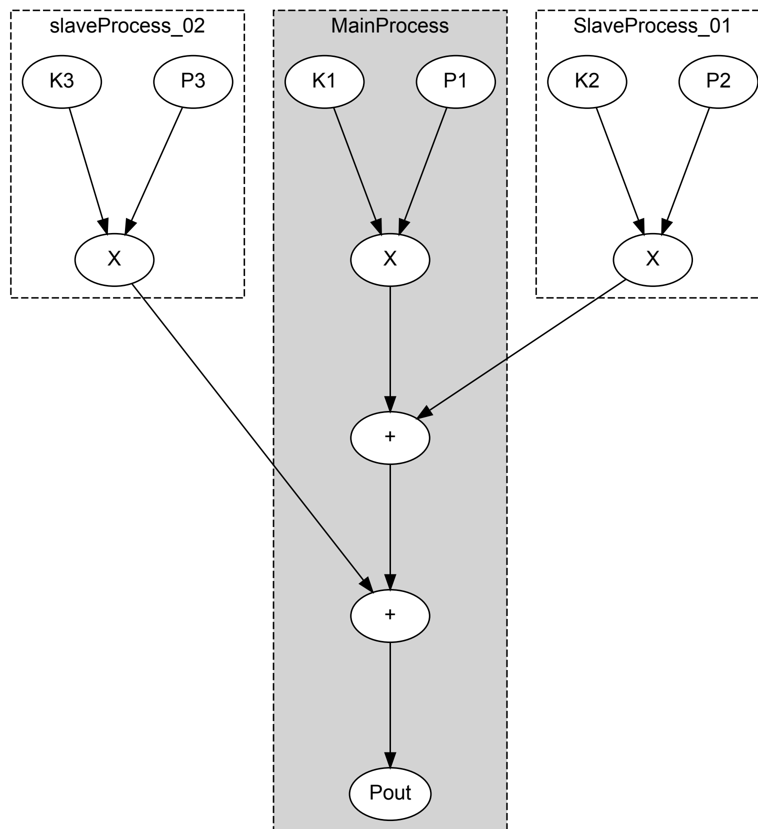


Figure 9.2: Example for a *task graph* with clustering [Wer15].

The cluster defined as "*MainProcess*", describes the part of the image algorithm

computed on the tile that produces the result for the pixel of interest. It is defined that for the calculation of an output pixel, all nodes and edges that are in the same cluster as the output pixel must belong to the "*MainProcess*" and the corresponding input pixel [8] must be included. This also applies to the calculation of more than one pixel of interest per tile.

The clusters called "*SlaveProcess*", are tasks that should be allocated to the neighbouring tiles, and the edges linking two clusters describe transactions between the tiles of the many-core architecture. These transactions should be mapped using the TLM library and the modelling styles defined there.

To generate the simulation model, the tasks in the graph nodes must first be correctly interpreted. The image processing algorithm has to be established by implementing the tasks and communicating with other process units. The necessary tasks for the neighbourhood must be implemented and made available to the right neighbours. After implementing the clusters in a processing unit, the defined network is to be established by TLM-connections and by instantiation of several process units to model the complete architecture.

9.3 Implementation Overview

The TLM models define a partition among processing and communication within an architecture, to allow for separate analysis for each part. A layer model that encapsulates a processing description in a structure description is used for this purpose.

Figure 9.3 shows a graphic representation of our model. The tiles consist of a *Core-Wrapper* and the *Tile-Code* contained in it. The *Core-Wrapper* has *Target-* and *Initiator-Sockets* and implements the functionalities for modelling the TLM communication styles.

The *Tile-Code* are the clusters of the *Task Graph*. The *MainProcess* (see Figure 9.2) is registered as the main process on the simulator's scheduler.

[8] The corresponding input pixel is the one in the input image which shares the same (x,y) coordinates with the pixel of interest.

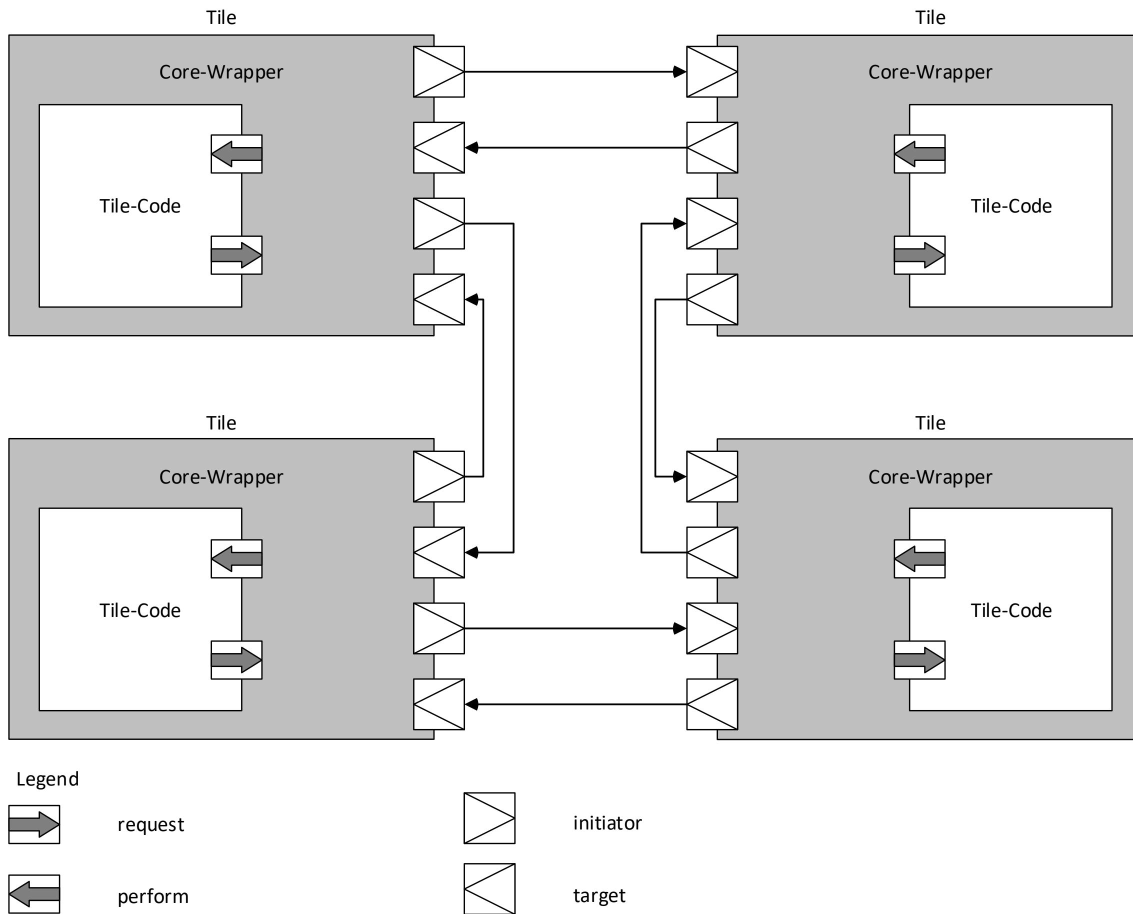


Figure 9.3: From the Task-Graph to a many-core model [Wer15].

The other clusters form auxiliary functions and receive their *Tile-Code* methods. The connection between *Wrapper* and its *Tile-Code* is established via the interfaces for calls from the *Tile-Code* on the *Wrapper* and for calls from the *Wrapper* to the *Tile-Code*.

The *Wrapper* forms, together with *Sockets*, a hardware structure of the model, while the clustered algorithm is distributed on *MainProcess* and auxiliary functions. The model of a tile is thus created by instantiating a TLM-*Wrapper* with corresponding *Sockets* and a *Tile-Code* object, which is then connected to each other via interfaces.

9.4 Functional Sequence

When starting the simulation, the main process of a *Tile-Code* of a processing unit is started. The processing of the code runs to a point where an edge connects two nodes of different clusters in the *Task Graph*. This is interpreted as a communication between two different tiles and generates the call of a data request from the *Tile-Code*. This initiates a TLM transaction in the *Wrapper* and places it to the addressed neighbours.

After that, the main process is halted until the *wrapper* successfully receive the data. The communication partner of the current transaction deals with this according to the chosen modelling style and for his part directs the enquiry to his *Tile Code* implementation. The address of the transaction object is used to differentiate between the individual auxiliary tasks. For example, an address can select a particular auxiliary task or address an exclusive data of the tile. To model this situation as well, the *Tile-Code* has a *Queue* that collects requests while the processing unit is busy.

The orders in the *Queue* are then processed as in a First-In First-Out (FIFO). Thus, if a tile halts its *MainProcess* because a value request is running, the processing unit is free to execute auxiliary functions for its neighbourhood. Processing requires computing time. Each cluster knows its processing time and suspends the auxiliary process for that time before returning the data and releasing the tile.

The complexity of the processing unit can be changed in such a way that the number of parallel processes can be adjusted or that the auxiliary functions can be distributed to hardware units, being able to emulate different processor models, as Very-Large Instruction Word (VLIW) ones.

9.5 Simulation Example

In this section, we show the results of a simple simulation of the 3×3 Convolution discussed in the last chapter. Different architectural possibilities were considered, to

show the flexibility of the developed simulator. We examined an image of resolution 12×12 pixels, varying the number of pixels per tile, and the number of tiles.

9.5.1 Clustering

Table 9.1 shows the parameters used to configure the clustering tool. We can see, for example, the *Node cost* is $10\times$ the *Edge cost*, what means that the communication costs are much smaller than the processing costs.

Table 9.1: Clustering parameters used in the example.

Name	Value
Number of iterations	3000
Node cost	1
Edge cost	0.01

The TG generated is shown in Figure 9.4, and the clustering result is shown in Figure 9.5. The colours represent the clusters. Therefore, edges connecting nodes of different colours symbolise external communication requests.

Figure 9.6 shows the tiled architecture with the communication pattern generated by the clustering process.

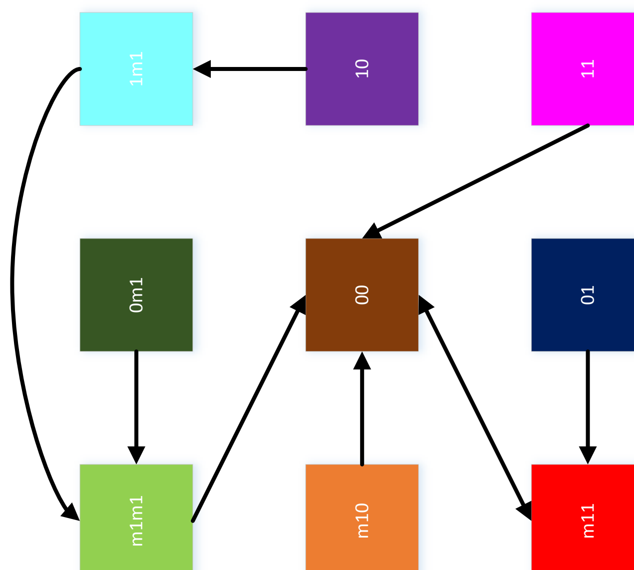


Figure 9.6: The tiles and the communication pattern after clustering [Wer15].

9.5.2 Simulation results

In Table 9.2, the simulation results are listed. As expected, the simulated time increased when we decreased the number of tiles. Some important aspects should be considered here, as the communication structure. The tasks are scheduled when they arrive, following a queue in the core wrapper, however, the communication channels are not blocked. Therefore, if a *Tile A* wants to request a pixel from *Tile B* and vice-versa, both messages can be exchanged at the same time. This is not what would happen in practice. In a real hardware implementation, the communication channels would act as bottlenecks, limiting the total bandwidth. This means that the model created still needs refinements to approximate the behaviour of real hardware better.

Name	**Pixel per PE**	**time (ns)**
conv_1_3	1	96
conv_2_3	4	333
conv_3_3	9	767
conv_4_3	16	1368
conv_6_3	36	3062

Table 9.2: Simulation results for the convolution example.

Another drawback identified in this simulation model is the fine granularity of the implementation. In the SystemC library we created, each task is a different class. A single TG will then be represented by several classes instances (objects) which consumes much memory in the computer hosting the simulation. This happens because by default the SystemC execution allocates the created objects in the stack. Each communication transaction is an object's instance, and the minimal support for garbage collection can lead the system to an *stack overflow*. A solution would be to use the *heap*, a garbage collector, or to modify the simulation model.

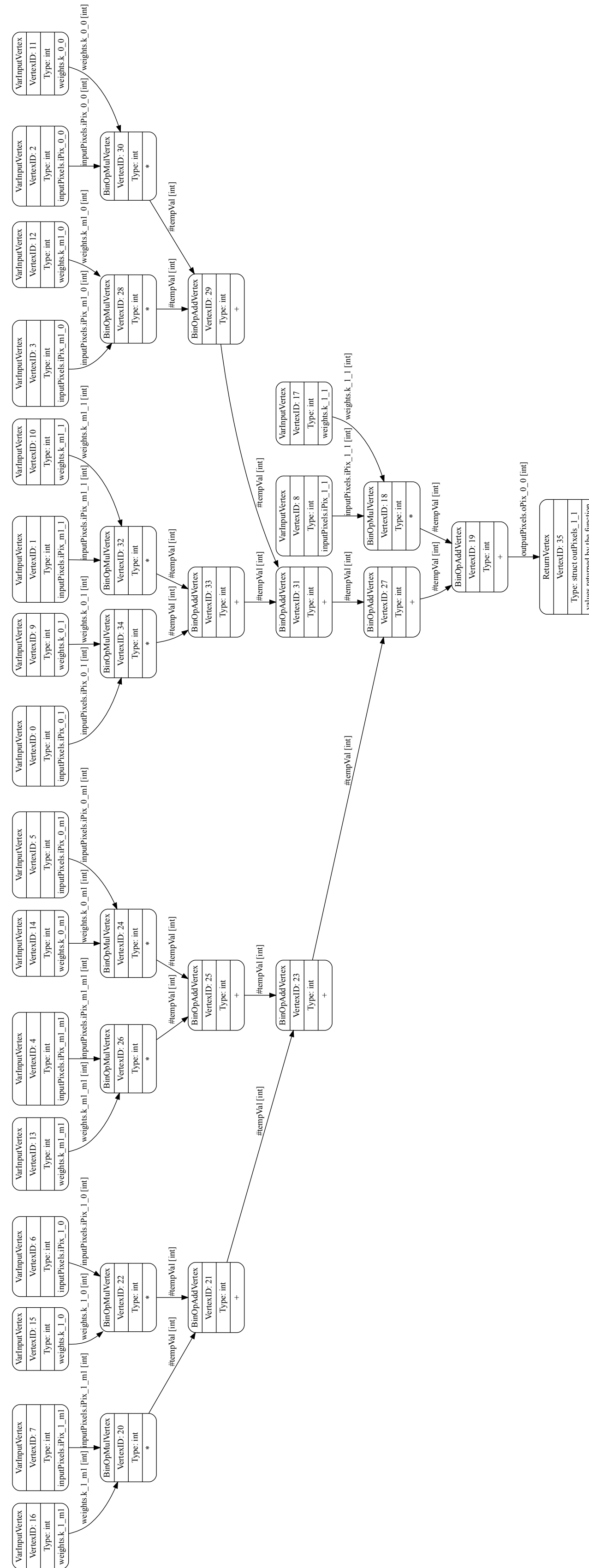


Figure 9.4: *Task Graph* for the example [Wer15]..

Figure 9.5: *Task Graph* for convolution after the clustering [Wer15].

Chapter 10

Discussion

In the previous chapters, we focused on developing tools to help the analysis of Image Processing and Computer Vision (IP/CV) *Tile-Codes*, using *Task-Graph* simulations. We developed a tool to extract the *Task-Graph* from an ANSI-C description, using the LLVM framework and the Clang compiler front-end. Also, we developed another tool which applies the *Force-Directed Clustering* technique to determine the optimal distribution of tasks, given costs for communication and processing. A SystemC/TLM2.0 simulator was also developed. This simulator takes as input a *Clustered Task-Graph* and some design parameters. Based on a library of nodes, the simulator assemblies a many-core system, where each Processing Element (PE) is implemented as a *Task-Graph.* Both Loosely-Timed (LT) and Approximately-Timed (AT) abstraction levels can be used in the simulator, which can also schedule the execution of every task and packet transmission.

The general idea of the *Task-Graph Clustering* flow is to perform a multi-goal optimisation, based on the application (represented by the *Task-Graph*) and a database of hardware models. At this point of the thesis, the database is not yet available. Therefore, the tests were performed only to show the techniques used.

The simulator developed can assembly a many-core system based on the clustered task-graph. The simulation to be performed is dependent on both the hardware database and the clustering results. Some drawbacks of this simulation model were discussed in the previous chapter.

With the tests performed, the simulator could provide correct results when compared to the original program in the *Tile-Code*, even when distributed among different tiles. This shows that the tool-flow developed here works as planned and present the expected behaviour.

In this part of the thesis, we have developed the desired tools and analysed whether they are useful or not in our approach. We can see now the need for a different simulation model, which shall be able to behave more like a physical hardware implementation. The communication structure must be taken into account, resembling Point-to-Point (P2P), buses, Network-on-Chip (NoC), or any other intrachip communication technology. The PE model used here is based on the task-graph execution and can be used to determine the final PE model.

In the next part of the thesis, we extend the current simulator to a lower-level of abstraction, to be more close to the real hardware. To do so, we discuss the design space of such architectures to reduce the number of possibilities to be explored.

Part IV

Intermediate-Level approach

Chapter 11

Introduction

In the previous part of this thesis, we have developed a high-level simulator, which uses a task-graph as the basis to determine the processing system performance. We identified the need for a lower level of abstraction to shrink the design space to a more focused set of possibilities. In this part, we extend the *Task-Graph Simulator* with different structures to resemble real hardware implementations.

In the first chapter, we discuss the design space and show the development of the new simulator modules, as well as the inclusion of a tool to help to estimate timing, power consumption, and silicon area of the hardware architectures analysed.

The second chapter shows simulation results and estimations for different hardware configurations, by running a more complex set of *Tile-Codes* in a complete Image Processing and Computer Vision (IP/CV) processing chain.

Chapter 12

The MPSoC Simulator

In this chapter, we use several concepts extracted from the literature, the parallelism analysis, and the application domain (IP/CV), to determine which characteristics should be explored to help to design many-core vision processors. With this information, we describe the implementation of a simulation tool with specific features and parameters.

The use of SystemC and TLM2.0 allows dividing the analysis into processing and communication [MH14]. We can take advantage of this separation to analyse the design characteristics of the PEs and the communication structure individually. Also, the use of Loosely-Timed (LT) and Approximately-Timed (AT) models allows the analysis of timing issues, parallelism, and resource sharing problems [MH14]. The *Generic Payload* is used in all communications among the modules, abstracting complex protocol implementation [Bla+10].

The simulator's main modules are shown in Figure 12.1. The basic architectural organisation assumes the *Region-Based* acquisition scheme sending parallel pixel streams to the *Tiles*. Each *Tile* is just a wrapper to encapsulate the *Processing Element* and the *Pixel Memory* blocks. Its main parameters are the image resolution, the region resolution and the region's coordinates. The other parts of the simulator are detailed explained in the next sections.

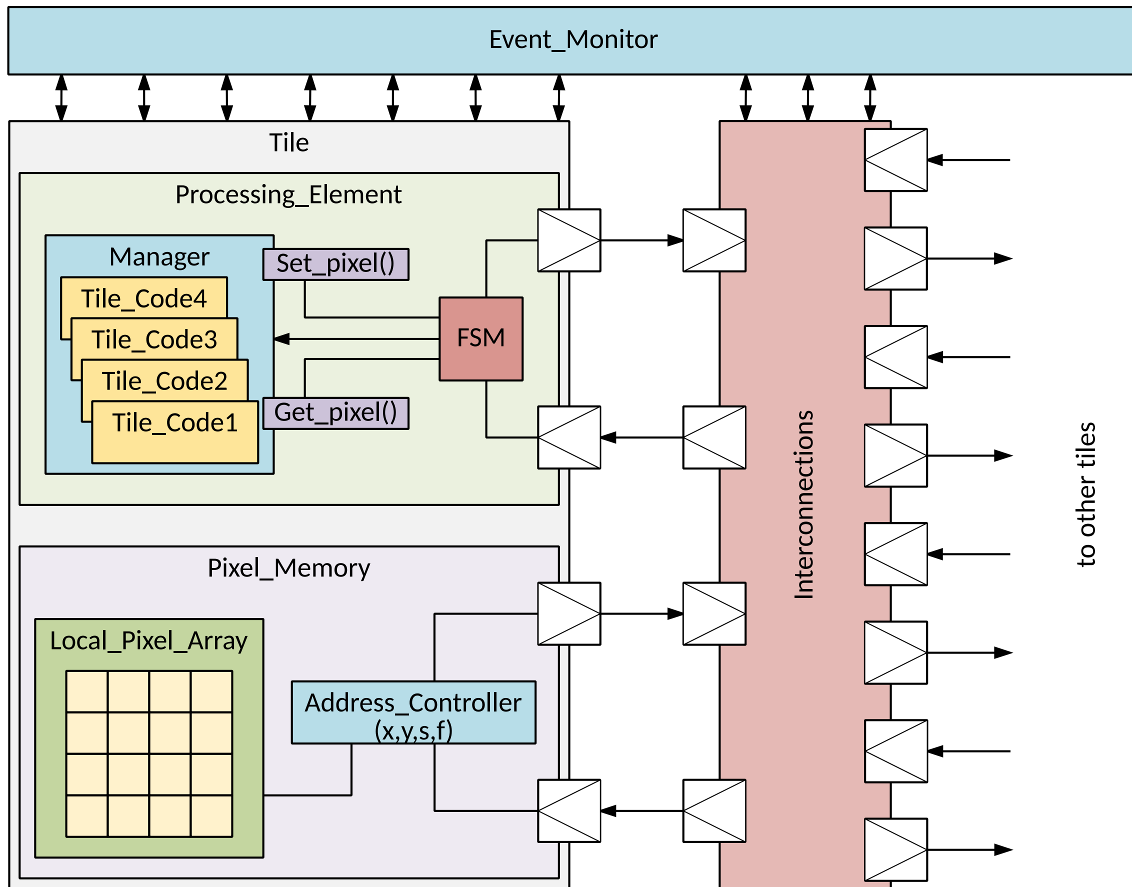


Figure 12.1: The simulation tool's organization.

12.0.1 Processing Elements

As shown in Figures 5.8 and 5.9, each PE is responsible for processing the pixels in its region. Therefore, each PE executes the *Tile-Code* for every pixel in its region.

An efficient PE architecture for IP/CV applications should be able to explore the Operation Level Parallelism [KG06] [Won+11]. This fine-grained parallelism can be considered a superset of the well-known Instruction Level Parallelism (ILP). VLIW (Very Large Instruction Word) and Superscalar processors are capable of exploring ILP by replicating in its microarchitecture several parallel execution units (multiple ALUs, Multiplication units, and so on). Recent works have shown that for common IP/CV algorithms, VLIW processors can perform more than 3× faster than a standard RISC processor [HWA15a]. The VLIW architectures, however, have a scalability problem: the switching module in their microarchitecture and the Register File grow exponentially in complexity and area when we add more parallel

units. This problem limits the amount of ILP to be parallelised and should be explored in more details.

In the simulator, each PE implements a *Finite State Machine* (FSM) which controls the execution of each *Tile-Code* and manages the data dependencies. In this module, it is possible to emulate the behaviour of sequential and parallel execution of different parts of the algorithm, by scheduling the SystemC execution threads to be simulated as parallel or sequential. Figure 12.2 illustrates the concept:

- In the first column, the operations are scheduled to be executed in parallel as soon as the data dependencies are solved. This organization emulates a VLIW/Superscalar architecture, with two multiplying units.
- In the second column, the operations are scheduled depending on hardware resource assumptions. This organization is more similar to a common RISC processor.
- In the third column, the scheduling is similar to the previous one. However, it schedules the input $J_y(i, j)$ to not be read in parallel to the first multiplication.

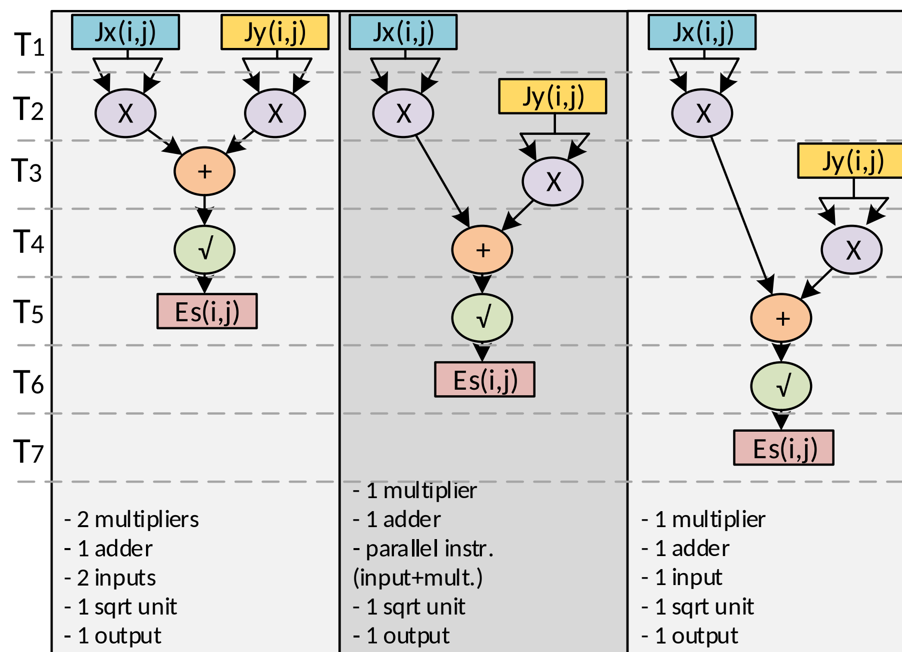


Figure 12.2: Example of how the simulator implements different PE types.

For the sake of simplicity, in this work we considered only two PE types:

- *RISC*: in this model, there is one multiplication unit and one ALU. No parallel operations can be scheduled. It corresponds to the third type in Figure 12.2.

- *VLIW4*: this model has four multiplication units and four ALUs, allowing for full parallelism: the operations are scheduled as soon as the data dependencies are fulfilled. It corresponds to the first type in Figure 12.2.

12.0.2 Pixel Memory Organization

In Figures 5.5 and 5.6, we can see that several intermediary images are generated. Each PE is responsible for processing a distinct region of all images, however, in *Neighborhood* and *Global* operations pixels from other regions are needed as inputs. Figure 12.3 shows this issue for 3×3 neighbourhoods: (A) only pixels in the same region are used; (B) pixels from two regions are used; (C) pixels from three regions are used. Most of the blocks in an IP/CV processing chain are based on region algorithms, which means that we must search for an efficient way to transfer pixels from one Tile to the other.

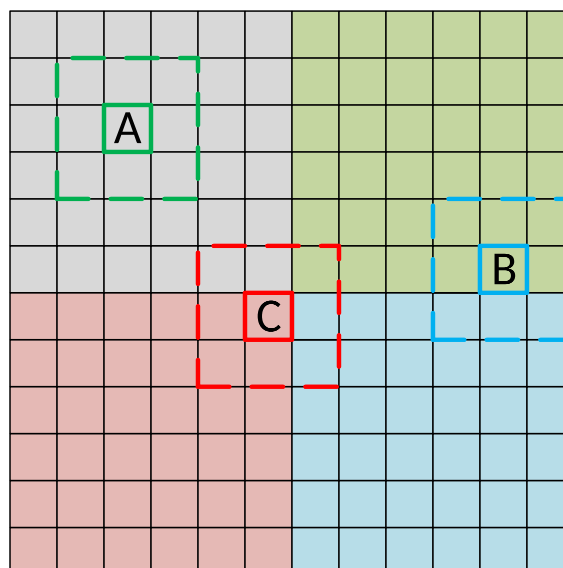


Figure 12.3: Example of pixel exchange needs among different image regions [MH14].

To maintain the *Pixel Memory* data coherency, each PE is allowed to write values only to the *Pixel Memory* of its *Tile*. Read access is allowed without any restriction.

We identified two possibilities for the *Pixel Memory* organization: (a) private, (b) shared, as shown in Figure 12.4.

In the private mode, the *Pixel Memory* access, for both read or write, is exclusive to the local PE. To have access to *Pixel Memories* in other *Tiles*, a PE makes a request to another PE in the corresponding *Tile*. This scheme works like interruptions in a common processor, where the interrupted PE stops its own algorithm's execution to perform the requested operation.

In the shared mode, the PEs requests are managed by the Pixel Memory itself, without the interference of the local PE. This scheme allows the local PE to execute its algorithms without interruptions, enhancing the overall performance. However, the *Pixel Memory* must be able to handle multiple requests, while maintaining its performance. Besides, the datapaths showed in Figure 12.4 have considerably different access delay, as can be inferred from the number of steps needed to access pixel values from other *Tiles*.

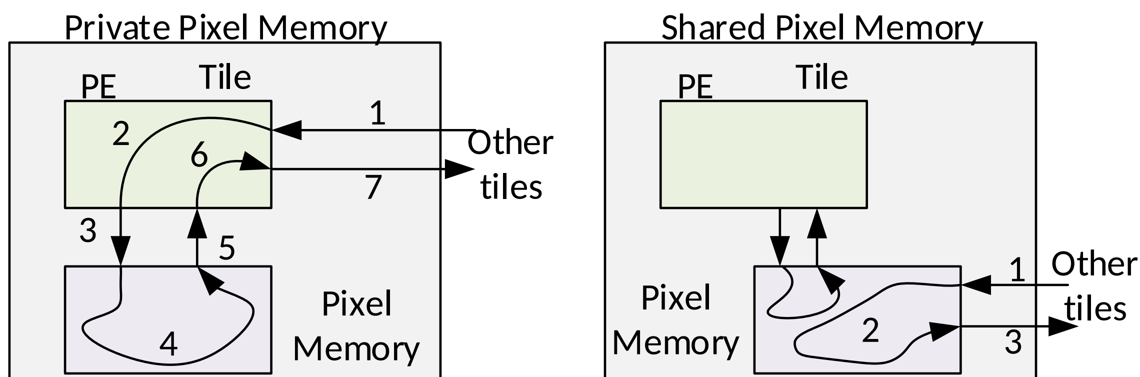


Figure 12.4: Private vs. Shared Pixel Memory for external accesses.

In the simulator developed, both models (private and shared) were implemented. The private model works as a common memory with single access (single rw port). The shared model was implemented with two rw ports: one for the local PE, and the other one connected to the Tile's communication interface.

12.0.3 Communication Structure

In the last topic, we show an example of pixel exchange among image regions. In addition to pixels, the PEs also need to exchange messages for synchronisation, status information, stalls and so on. There are several solutions for intrachip communication, like point-to-point (P2P), bus, cross-bar, and Networks-on-Chip (NoCs). Each type of communication structure offers advantages and disadvantages in throughput, traffic control, fault tolerance, silicon area and energy consumption. The right choice of the communication structure is one of the key points to meet the design constraints.

The *Communication Structure* is responsible for distributing the pixel values (from the input and intermediary images) and synchronization messages among the tiles. Let's consider a general neighbourhood operation which takes a square region to perform some computation. All the transactions shown could occur at the same time, due to the data dependencies. The PEs are simultaneously requesting pixels among them, which generates much traffic and conflicts, and the structure choice can determine the success of the design in meeting the constraints.

In Point-to-Point communication, the PE is responsible for the communication tasks, besides the processing ones. It is also responsible for transferring data among its neighbours, so the PE located between two communicating tiles would be interrupted to perform message and data transfers [DT03]. To the time spent in processing, it is added the time needed for the communication tasks and the context change inside the PE (e.g. interruption handling). Also, some IP/CV algorithms have unpredictable communication needs (content-based algorithms), which means that the amount of context changes is not previously known.

For a bus-based approach, all PEs should be considered bus masters, which requires an arbiter (extra hardware/energy consumption). Also, bus architectures do not scale well for systems with dozens to hundreds of Tiles, which can be a drawback for implementing systems with high image resolutions.

The state-of-art solution, which presents good scalability and more efficient ways

to handle simultaneous communication issues are the Networks-on-Chip [Hes+16]. The main characteristic of a NoC is the separation of processing and communication by the addition of specialised hardware for communication. This strategy frees the PE to be optimised for the processing tasks only, increasing the system's performance. However, a NoC includes intrinsic message propagation delays, due to the presence of the Routers. The Routers are specialised architectures for packet switching transmission, being responsible for data transfer among *Tiles*.

In our architecture, the *Tiles* are physically placed in a grid, to have equally distributed acquisition delays, as explained in Section 5.2.3. Also, because of most IP/CV algorithms, the *Tiles* would need more pixels from their surrounding neighbours. Because of these characteristics, we consider two general topologies for our system: *4-Connected* and *8-Connected* (Figure 12.5). The main difference between them is the number of connections each *Tile* has to the neighbouring ones. Each topology offers a different balance among area, power consumption, and throughput, which must be analysed carefully.

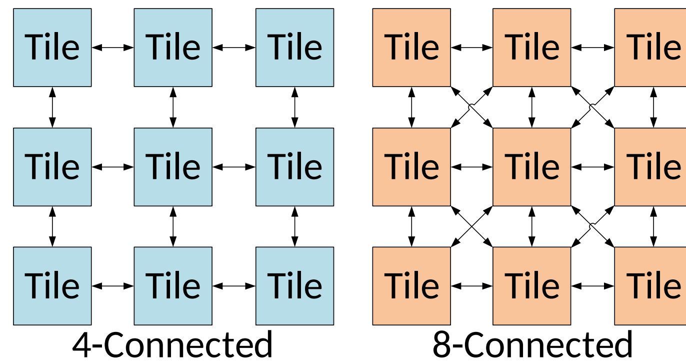


Figure 12.5: 4-Connected and 8-Connected topologies simulated.

In the simulator, the *Interconnections* module is configurable to behave like the different communication structures discussed in this Section. All *Tiles* connect to this module, which varies the number of I/O channels depending on the number of ports to be simulated. It manages the requests and message buffers, depending if LT or AT simulations are desired.

12.0.4 Programming Model

A common design mistake in the area of Multi/Many-Core Architectures is that most designers first develop the hardware part, postponing the software toolchain development until the architecture is finished. Several problems appear then since the application's developer is in general not the processing system designer. A good programming model should offer to the programmer an abstraction which allows him to focus on the application domain, without caring about code optimisation, performance and other constraints. We are considering here the programming model based on the *Tile-Codes*. Figure 12.6 shows a general partition of the programming model proposed. In this model, the applications are described using pre-defined blocks (IP/CV libraries), assembling a *Processing Chain*, as shown in Figure 5.5. Each block is mapped to a *Tile-code*. To *Get* and *Set* pixels, a *Communication Layer* provides special functions, and, for the sake of completeness, messages for synchronization among PEs.

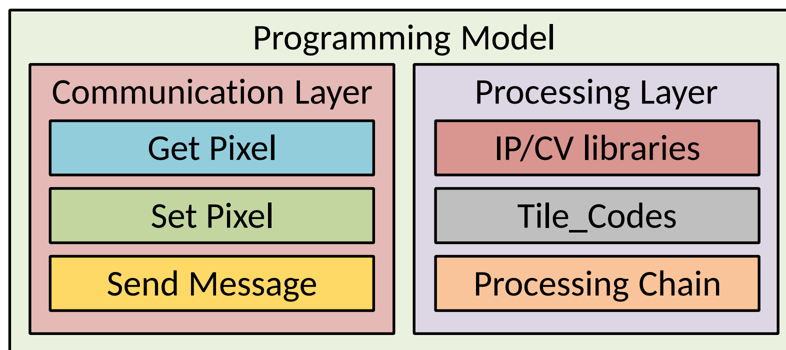


Figure 12.6: Programming Model overview.

Despite its simplicity, this programming model allows a programmer to write a vast number of IP/CV applications. An example of a custom Tile-Code is shown in Listing 12.1. This code generates as output the geometric mean of a 3×3 neighbourhood.

Listing 12.1: A custom *Tile-Code* to determine the geometric mean of a 3×3 neighborhood.

```
void geomean(int x, int y, int s, int f)
{
    int p1 = get_pixel(x-1,y-1,s,f);
    int p2 = get_pixel(x-1,y  ,s,f);
    int p3 = get_pixel(x-1,y+1,s,f);
    int p4 = get_pixel(x  ,y-1,s,f);
    int p5 = get_pixel(x  ,y  ,s,f);
    int p6 = get_pixel(x  ,y+1,s,f);
    int p7 = get_pixel(x+1,y-1,s,f);
    int p8 = get_pixel(x+1,y  ,s,f);
    int p9 = get_pixel(x+1,y+1,s,f);

    int pout = sqrt(p1*p2*p3*p4*p5*p6*p7*p8*p9);

    set_pixel(x,y,s+1,f,pout);
}
```

The parameters *(x,y,s,f)* in *get_pixel* and *set_pixel* functions can be explained as follows:

- x : the X coordinate of the desired pixel.
- y : the Y coordinate of the desired pixel.
- s : identifies the corresponding block in the IP/CV chain.
- f : identifies the acquired frame, used for image sequences.

12.0.5 The Event Monitor

In the simulator, the *Event Monitor* (Figure 12.1) traces all activities running in the simulations. It collects FSM status from the PEs with timing annotations, to identify when each processing step occurs. This module also registers *Pixel_Memory* accesses, useful to determine the reading/writing patterns. The Interconnections

module is also monitored. The buffers and message routing schemes can also be accessed by the *Event_Monitor* and all switching activities, transactions, and states are stored in a log file for late analysis.

12.0.6 Estimation of Power, Area and Timing

When designing a system at the Register-Transfer Level (RTL), the designer can perform the synthesis (for both FPGA or ASIC), and get estimations about the power consumption, the area, and the maximum operating frequency (directly related to the propagation delays). However, when the design is developed at the Electronic System Level (ESL), in general using high-level descriptions (Transaction Level, Specification Level, and so on), accurate estimations are complicated and still a challenge in the industry and academy [Ger+09].

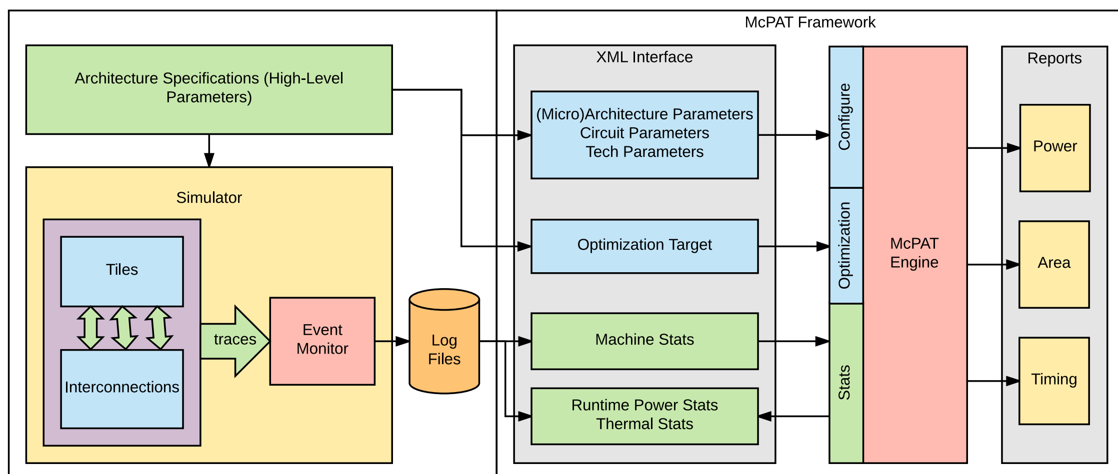


Figure 12.7: Integration of our simulator with the McPAT Framework (adapted from [Li+13a]).

In this work, we use McPAT (Multicore Power, Area, and Timing), a framework to estimate power consumption, silicon area, and timing [Li+13a]. McPAT receives as inputs a description of the architecture (processor descriptions, caches, memories, buses, NoCs, routers, register files, ALUs, multipliers, and so on) and a utilisation rate for each block. Each architectural block can be described with variable detail levels, and the estimations are as accurate as the details.

Figure 12.7 shows the integration of our simulator to the McPAT Framework.

The Architecture Specifications (pixels per tile, communication type, processing element configuration, and so on) are used to configure the Simulator and to fill a high-level description in the *XML Interface*. The Simulator's *Log Files* are also used in the *XML Interface*, to load the utilisation rate o each architectural block. McPAT reads the *XML Interface* file and provides reports about the Power, Area, and Timing estimations for the given architecture.

Chapter 13

Simulation Results

In this chapter, we show the simulation results for a simple edge detection application. We also include estimations obtained by an open-source tool for power, timing, and area analysis.

13.1 Simulated parameters

Using the simulation tool described in the last chapter, different configurations were profiled. Some of the underlying assumptions we used in the simulations are in the Table 13.1, where our configurations are compared to a recent work which implements an array of 8-bit processors integrated to the CMOS pixel sensor, in the focal plane.

Table 13.1: Overview of configurations in [Sch+16] and in this work.

Parameter	[Sch+16]	this work
Technology	$130nm$	$90nm$
Area (mm2)	~ 8.04	~ 10 for the (8×8)
Total resolution	(80×64)	(64×64)
Region (WxW)	(8×8)	$(4 \times 4), (8 \times 8), (16 \times 16), (32 \times 32), (64 \times 64)$
Power (mW)	36	~ 30 for the (8×8)

In Table 13.2 we can see the 16 architectural configurations used in our simulations. Also, each configuration was also parameterised with the *Region Size*, giving the total of 80 simulated architectures. The work in [Sch+16] uses an architecture

similar to configuration *A*.

Table 13.2: Simulated architectural configurations.

Config.	Topology	Communication	Pixel Memory	PE Type	Config.	Topology	Communication	Pixel Memory	PE Type
A	4-Connected	P2P	Private	RISC	I	8-Connected	P2P	Private	RISC
B				VLIW4	J				VLIW4
C			Shared	RISC	K			Shared	RISC
D				VLIW4	L				VLIW4
E		NoC	Private	RISC	M		NoC	Private	RISC
F				VLIW4	N				VLIW4
G			Shared	RISC	O			Shared	RISC
H				VLIW4	P				VLIW4

13.2 Area estimation

Figure 13.1 shows the Area estimation for the Table 13.2 configurations. When we have W=64, the region has the same size as the complete image, which means that there is negligible communication overhead, and almost all area is used for the *Processing Element* and the *Pixel Memory.* When we move in the graph to the left, reducing the Region's size, we increase the communication overhead and the number of *Tiles.* It is important to highlight that in the left side of the graph when we change the PE Type from RISC to VLIW4, the total area almost doubles (from A to B, from C to D, and so on). These characteristics show how sensible is the area to the PE Type selected.

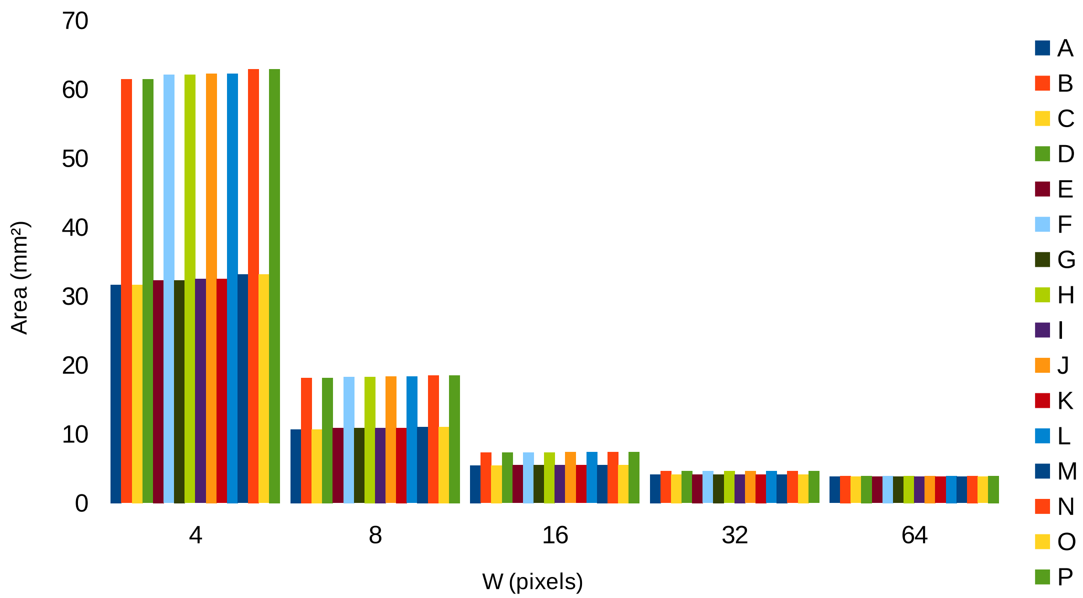


Figure 13.1: Estimated area for all configurations, considering $W = [4, 8, 16, 32, 64]$ pixels.

13.3 Power estimation

In Figure 13.2, the Power consumption estimation can be seen. It is straightforward to observe that the graph looks similar to the Area estimation (Figure 13.1), since the power consumption is directly related to the silicon area used. The Power consumption estimation was done only statically, considering the elements used in the composition of the architecture.

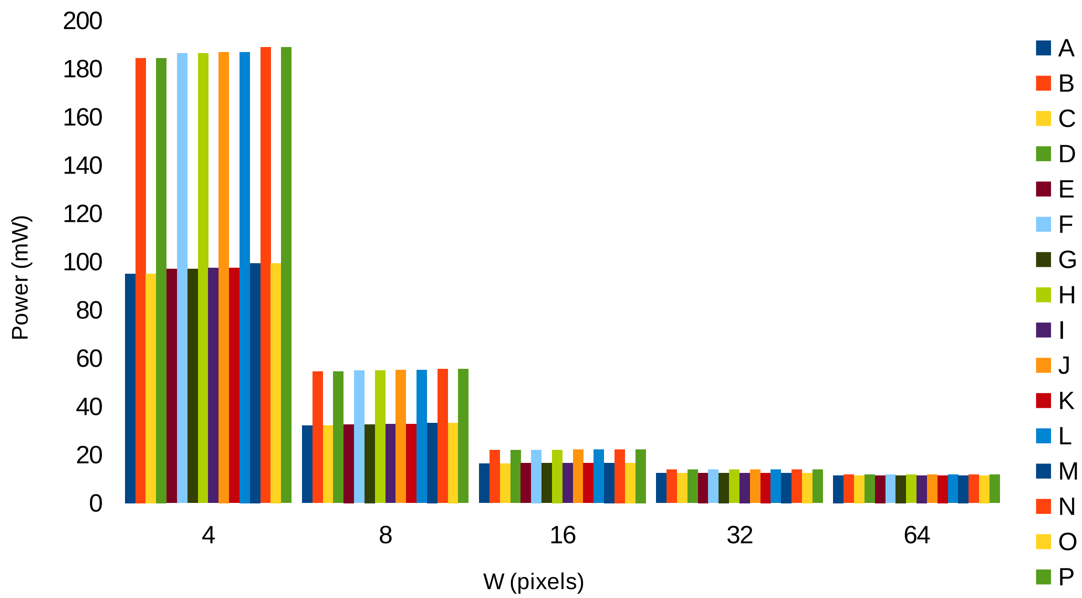


Figure 13.2: Estimated power for all configurations, considering $W = [4, 8, 16, 32, 64]$ pixels.

13.4 Performance estimation

The performance was estimated by the number of cycles needed to compute the application from Figure 5.5. The graph in Figure 13.3 shows a logarithmic graph for the number of cycles. We can observe that the choice of RISC or VLIW4 can have a strong impact on the performance (a bit less than one order of magnitude). The parallelism exploration is highlighted in this graph: for example, Configuration A for W=8 and W=16 present a difference in performance of 5 times.

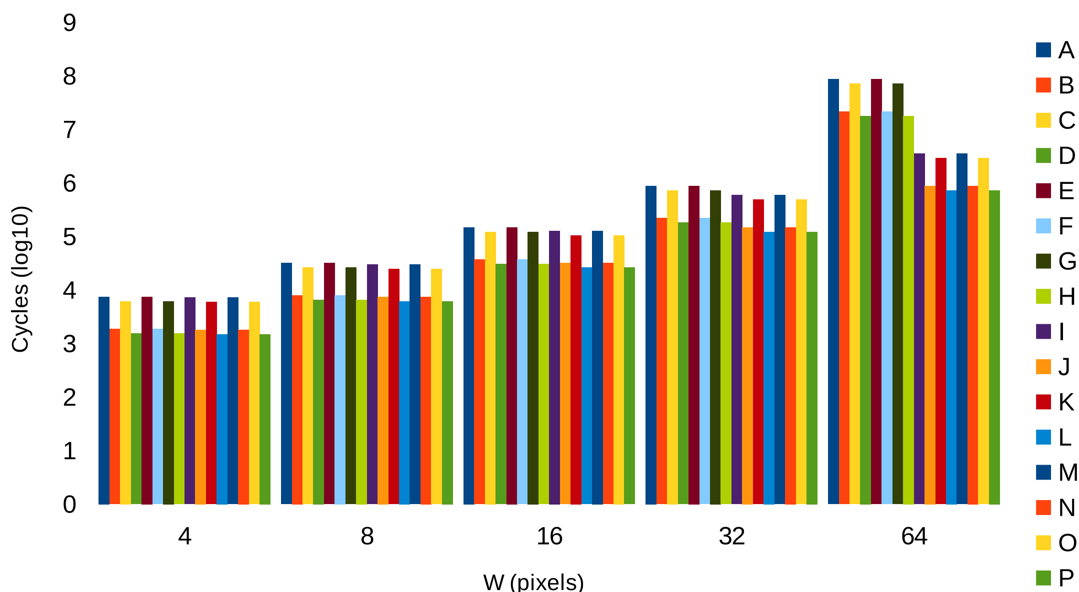


Figure 13.3: Estimated cycles for all configurations, considering $W = [4, 8, 16, 32, 64]$ pixels.

13.5 Results discussion

In this chapter, we performed several simulations with different architecture configurations, to show the flexibility of both the programming model (the library of *Tile-Codes*) and the simulator.

Besides, we were able to estimate the static power consumption, the silicon area, and the throughput per pixel in different architecture configurations.

We can highlight the high impact in performance observed when changing from the simple RISC model to the VLIW-like model, which came with an equivalent overhead in area.

80 architecture configuration possibilities were analysed in this chapter. However, there are several other design possibilities that could be explored. The exhaustive search for an optimal solution was not the goal of this chapter. Our objective was to show the design trade-off considering the selected design-space.

The main drawback at this point is that our simulation models are not cycle-accurate. To estimate precisely the power consumption, the area, and the timing

delays, it is necessary to perform the synthesis of the proposed architectures. This leads us to the next part of this thesis, which is dedicated to the Register Transfer Level (RTL) description of selected elements from the design space.

Part V

Low-Level approach

Chapter 14

The Baseline Architecture

Considering the necessity of synthesizable architecture models, as highlighted in the last chapter, we show in this part the Register Transfer Level (RTL) implementation of a baseline architecture for our many-core system.

As a proof-of-concept, we derived a many-core architecture suitable for the IP/CV algorithms, as shown in Figure 14.1. The architecture has the following characteristics: (a) the *Pixel_Memories* are accessed in *Shared Mode*, and (b) the communication is performed by a *4-Connected* NoC. Considering this model, the architecture was then described in VHDL. In this section, we show details of the hardware description and its implementation in an FPGA platform. In this work, we used the **Xilinx Kintex-7 fabric**, a mid-end modern FPGA technology.

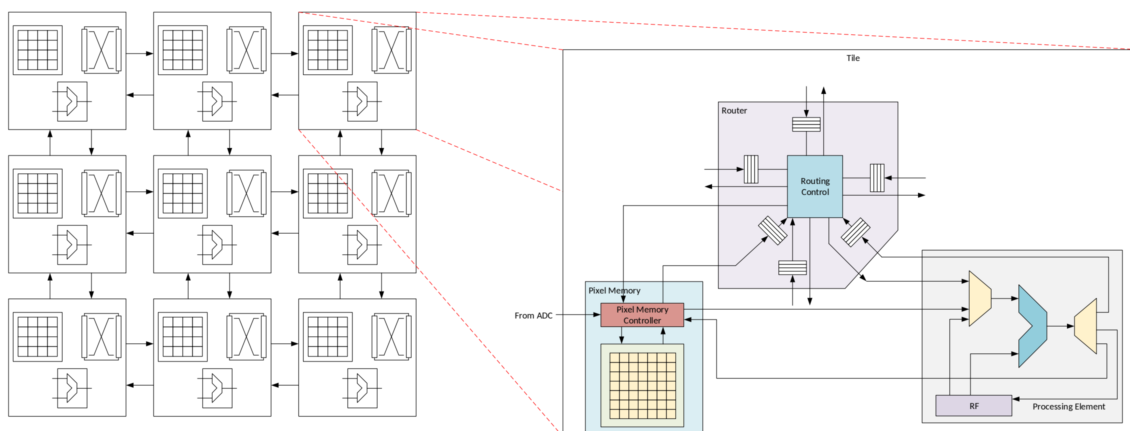


Figure 14.1: Many-Core Vision Processor and its Tile structure.

14.1 Pixel Memory

The *Pixel_Memory* block is responsible for storing pixels from the input image and all other intermediary images. It is also responsible for receiving the input pixels from the CMOS sensor array. In our architecture, each PE is in charge of a region of the image. This means that it is responsible for processing the region's pixels. The local PE and the local Router have read access to all pixels in their region. The write access to the addresses corresponding to the input image is restricted to the sensor's ADC. The remaining addresses (intermediary images) can only be written by the local PE. This organisation ensures the data coherence.

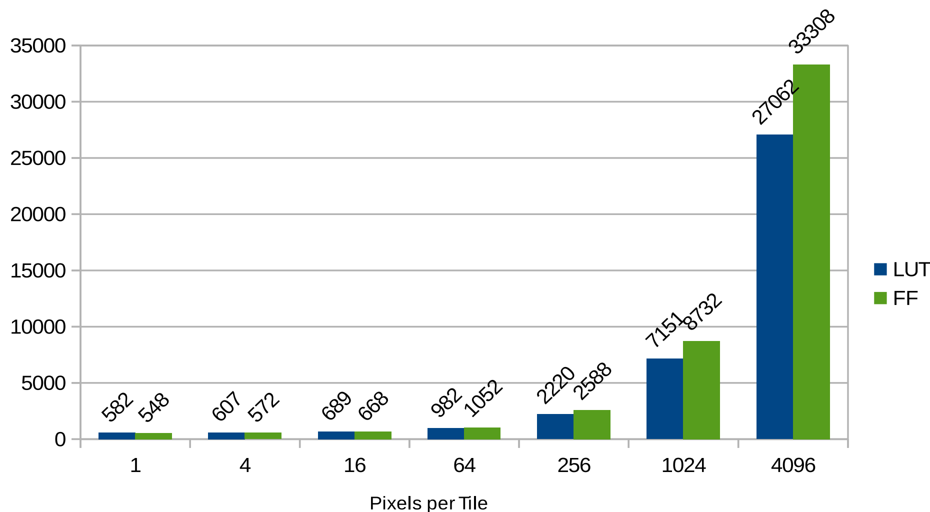


Figure 14.2: *Pixel_Memory* synthesis for different amounts of pixels per Tile (8 bits per pixel).

Figure 14.2 shows the synthesis results for varying amounts of region pixels (8 bits per pixel). There is no distinction among pixels from input or intermediary images. The only difference between them is the memory addresses used for each pixel.

14.2 Processing Element

The *Processing_Element* block is responsible for implementing the processing operations of the IP/CV algorithms. It was implemented as a programmable FSM

able to get and set pixel values to its local *Pixel_Memory* or through the NoC using its local *Router*. Figure 14.3 shows the PE internal organization. The ALU (Arithmetic-Logic Unit) performs 16-bit operations using integer arithmetic. The *Instruction Set* used is shown in Table 14.1. It is quite reduced and encompasses standard instructions (add, sub, mul, div, abs, jmp, lt, bt, eq), in addition to specific ones (getpx, setpx).

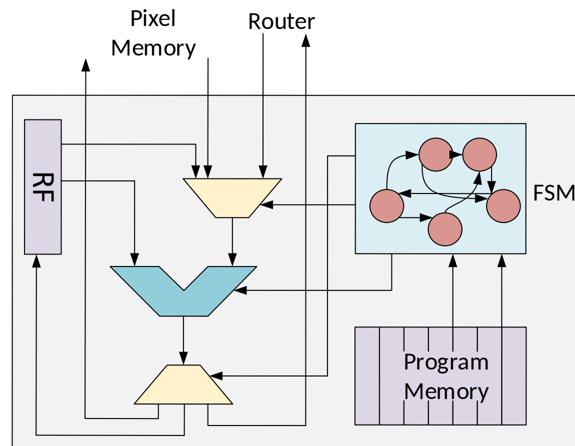


Figure 14.3: *Processing_Element* internal organization.

Table 14.1: Instruction Set used in the PEs (RISC processors).

Instruction	Description
ADD	Arithmetic addition of two integers.
SUB	Arithmetic subtraction of two integers.
MUL	Arithmetic multiplication of two integers.
DIV	Arithmetic division of two integers.
ABS	Arithmetic absolute value of the input integer
JMP	Jump to specified instruction address.
BLT	Branch if less than.
BGT	Branch if bigger than.
BEQ	Branch if equal to.
GETPX	Get a pixel value from the specified address.
SETPX	Set a pixel value to the specified address.

The *Program_Memory* and the *Register_File* can be configured at design time to have different sizes. Considering the *Processing_Element* without the *Program_Memory* and the *Register_File*, the synthesis results are: **245 LUTs** and **114 FFs**. The other two elements are just memories and their resource use is straightforward.

14.3 Router

The router, in our architecture, is an element responsible for receiving pixel requests (from the local PE or neighbour Tiles) and transmitting pixel values. In our implementation, the communication among the routers is using single-packets with all bits in parallel. A simple handshake is used to synchronise the communication in any router port. In total, the router has six inputs and six outputs, connected to the four neighbour *Tiles*, to the local *PE*, and to the local *Pixel_Memory*.

Figure 14.4 shows the *Router*'s internal structure. It is a pipeline, and a simple handshake is used to transfer the messages between each pair of stages. The first stage is an *Arbiter*, which uses a Round-Robin scheme to give the same priority all ports. The second stage is a *Decoder*, that determines the output port of each message. The third stage is the *Channel_Scheduler*, a shared message buffer, which stores each message until its destination port is free.

Table 14.2 shows the *Router*'s synthesis results with the architecture configured for a 256×256 image resolution, five steps, one frame and 12 messages in the *Channel Scheduler* buffer. The total sums of the elements are not the same as the *Complete_Router* values since each component was synthesised separately. The synthesis tool optimises differently when synthesising individual elements or complete systems.

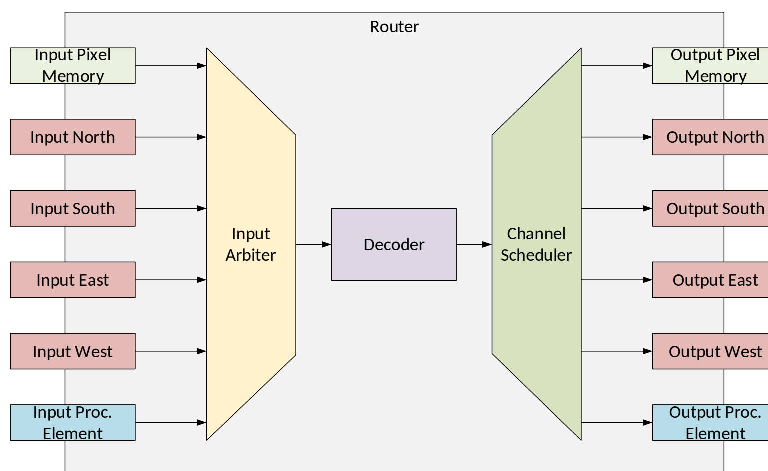


Figure 14.4: *Router*'s internal structure.

Table 14.2: Router synthesis results: 256×256 resolution, 5 steps, 1 frame, 12 messages.

Component	LUT	FF
Input Arbiter	18715	1105
Decoder	41	1046
Channel Scheduler	2758	9442
Output Controller	24	6252
Complete Router	20461	15663

14.4 Profiling Results

To illustrate the performance of our architecture, we show in this section the implementation of a complex IP/CV application composed of several algorithms organised in a chain: the Canny Edge Detector (CED). This algorithm is a well-known optimised edge detector published in [Can86], and detailed in [TV98]. The CED algorithm encompasses most of the general IP/CV processing chain shown in Figure 4.1. In the following topics, we detail each part of the algorithm and how it performs in our architecture, also making a comparison with state-of-art implementations from the literature.

14.4.1 Pre-Processing

The first three blocks of the CED algorithm are Neighborhood Operations (Figure 4.2). The *Gaussian Filter* is a *Pre-Processing*, and the gradients (G_x, G_y) are part of the *Segmentation*. These three algorithms are implemented using convolution operations. The first block uses a 5×5 filter, which requires a neighbourhood of the same size. The gradients are implemented using 3×3 neighbourhoods.

Communication through a NoC implies some level of uncertainty since it is not possible to determine precisely when a message will achieve its destination. The uncertainties come from the unpredictability of the arbiter, the routing algorithm, and the PE requests [Hes+16]. As shown in Fig.12.3, depending on its position, a pixel can have different amounts of local and external pixels.

The *Worst-Case Pixel* (WCP) is the pixel which presents the highest delay to

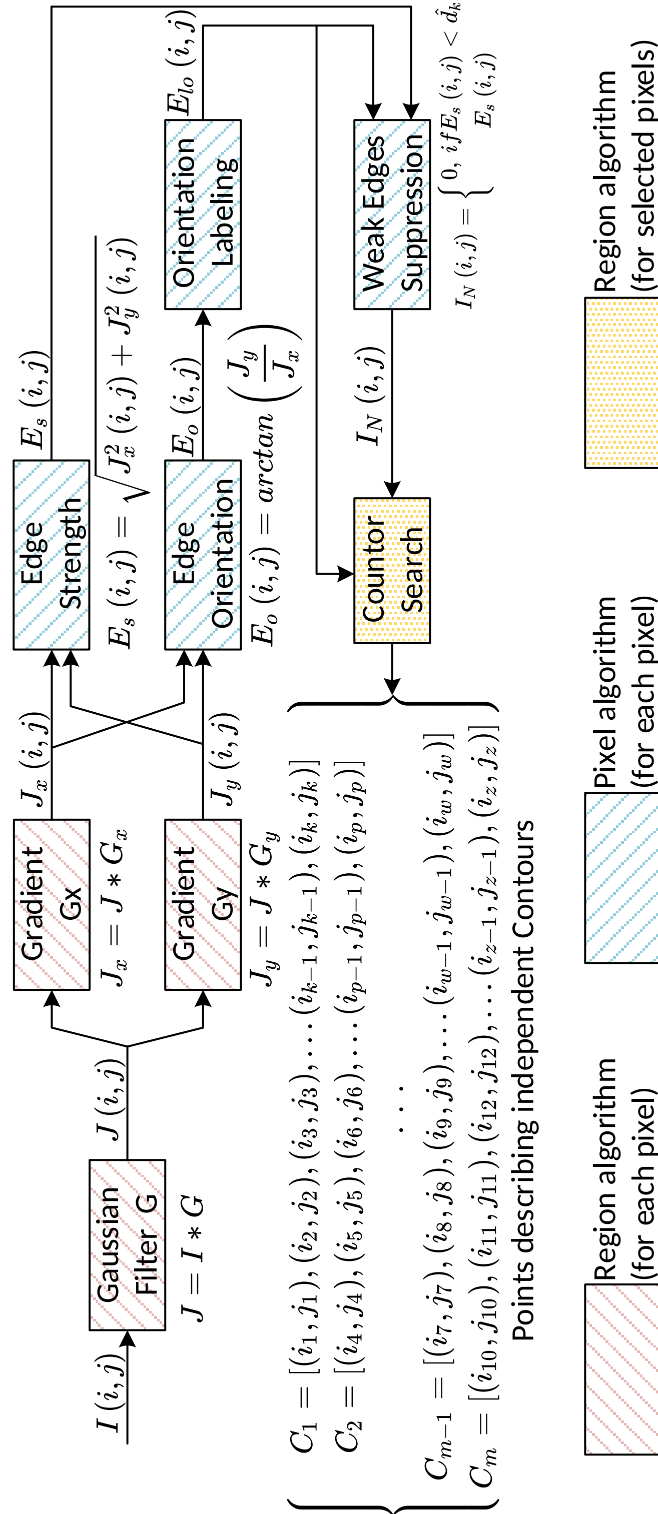


Figure 14.5: The Canny Edge Detector method: building blocks and operation's types.

be computed. Figure 14.6 shows how the Neighborhood Operations perform in our architecture, considering a different number of pixels per Tile. When we increase the number of pixels per Tile, the number of local pixels also increases, reducing

the accesses time. However, this also limits the amount of parallelism explored for a fixed image resolution. In this example, the image used has a resolution of $64 \times 64 = 4096$ pixels.

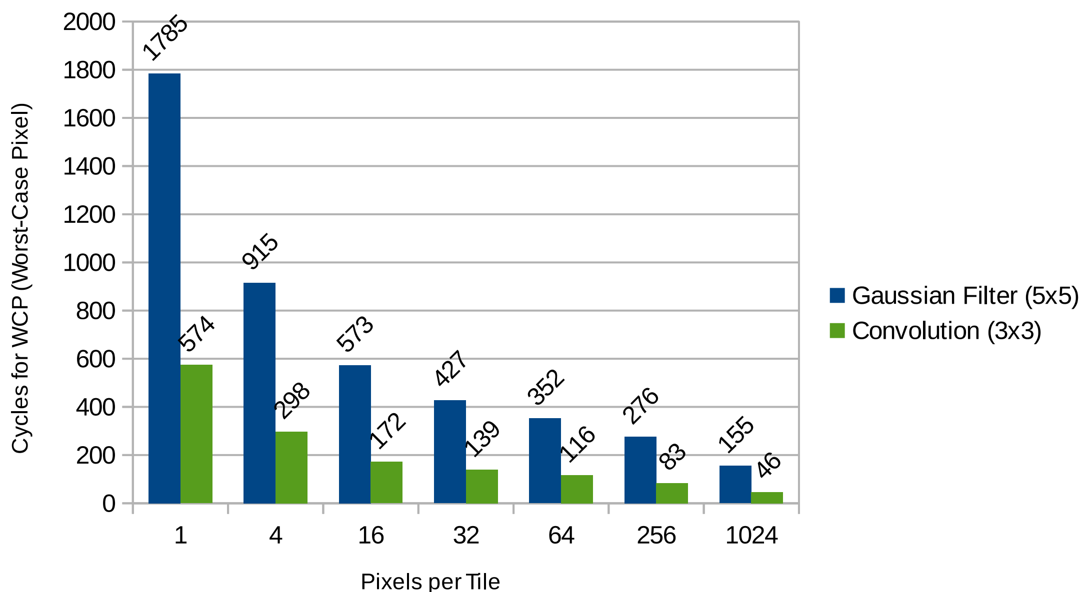


Figure 14.6: Performance of Neighborhood Operations for different number of pixels per Tile.

14.4.2 Segmentation

The processing blocks related to *Pixel algorithms* (*Edge_Strength*, *Edge_Orientation*, *Orientation_Labeling*, and *Weak_Edges_Suppression*) are all Local Operations. They do not need information from neighbour pixels (local or external). In this type of operation, there is no NoC communication: all pixels are local to the corresponding PE, and the image resolution does not affect the performance per pixel. Figure 14.7 show the performance achieved by our architecture for the Segmentation Operations of the CED application.

14.4.3 Feature Extraction

The last processing block is the *Countor_Search* operation, a particular type of Neighborhood Operation since it does not apply to all pixels. This operation uses

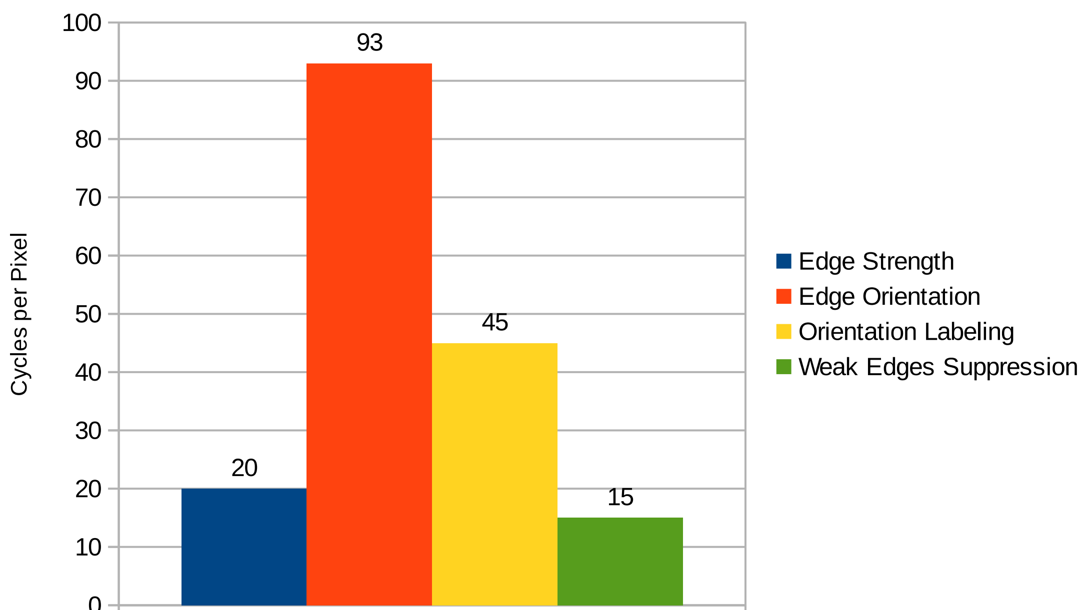


Figure 14.7: Performance of Segmentation Operations for different number of pixels per Tile.

as inputs two intermediary images: $E_lo(i,j)$ and $I_N(i,j)$. The *Countor_Search* links pixels over a certain threshold, to determine if they are linked (in the same edge) or not.

The performance of the operation depends on the image content. For this example, we used a chess board image of 64×64 pixels. Each board square is 8×8 pixels. This operation was implemented using a search window of 3×3 pixels, which give the same WCP delay as for the Convolution in Figure 14.6.

14.4.4 Complete Application

Figure 14.8 shows the performance of the complete CED application over our architecture, for different numbers of pixels per Tile. Table 14.3 shows a comparison of our architecture and state-of-art implementations from the literature. For the comparisons, we estimated our performance based on the synthesis results shown in the previous section.

Table 14.3 allows us to make a broad analysis, considering different points-of-view. Regarding the frame-rate (fps), our architecture is scalable and maintains the same frame-rate for any resolution.

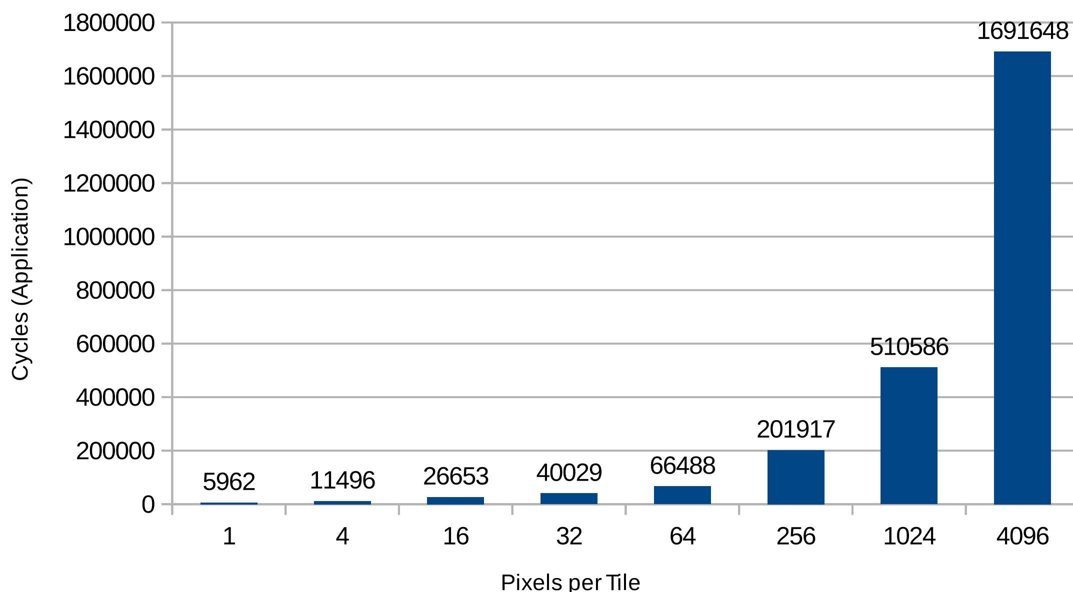


Figure 14.8: Performance of the CED application for different number of pixels per Tile.

Table 14.3: Comparison with related architectures, all implementing the CED application.

Reference	Programmable?	Performance	Platform
This work: (16 × 16) px/tile	Yes	495 fps/100MHz/any resolution	FPGA Xilinx Kintex7 fabric
[Pos+14]	No	909 fps/242MHz/512 × 512	FPGA Altera Aria V
[Pos+14]	Yes	33.3 fps/2.46GHz	CPU Intel Core 2 Duo
[Pos+14]	Yes	473 fps/1.54GHz	GPU Geforce GTX580
[Bre+11]	Yes	1.3 fps/700MHz/512 × 512	Tilera T64 VLIW-MPSoC
[Gen+10]	No	1515 fps/201MHz/512 × 512	FPGA Xilinx Spartan 6
[HY08]	No	400 fps/27MHz/360 × 280	FPGA Altera Cyclone-I

Dedicated FPGA implementations are not flexible (programmable), and application modifications can lead to an entirely different design, taking a long time to be implemented. However, the dedicated hardware can achieve high throughput in comparison with programmable implementations. Our architecture can outperform a standard CPU since it is an application-specific design.

The Tilera T64 has a mesh array of 3-issue VLIW cores on a single chip. In [Bre+11], the CED application was implemented using threads, and maybe the programming model was responsible for the poor performance achieved. Due to the exploration of multiple parallelism levels, and the design focused on the IP/CV application domain, our architecture can outperform such commercial MPSoC.

14.5 Conclusion

In this chapter, we proposed a baseline architecture for future Many-Core Vision Processors. This architecture was prototyped in an FPGA and compared to state-of-art solutions for Real-Time IP/CV. The performance of our architecture is shown to be flexible and scalable, in comparison with the state-of-art works. This implementation can serve as a basis to guide the development of more precise analysis, and more RTL implementations can help to enhance the estimation accuracy, for either Area or Power consumption.

Chapter 15

Processing Element Design

In this chapter we discuss two approaches to develop the Processing Elements for our many-core architecture [9]:

1. High-level synthesis of the *Tile-Codes*.
2. An Application-Specific Instruction set Processor (ASIP) design based on a stream buffer and a novel direct injection of data to the processor's datapath.

15.1 The High-Level Synthesis Approach

In this section, we show the development of a tool which receives as input a task-graph and some design constraints and, by interacting with the Xilinx Vivado HLS software, generates a synthesizable version of the task-graph. At the end of this section, we propose the utilisation of the *Synthesized Task-Graph* as processing element in our many-core architecture.

15.1.1 Introduction

The advances in VLSI technology during the last decades provided a high degree of miniaturisation, enabling the ubiquity of sensing/processing/actuating embedded systems. New applications appear daily, and the minimisation of the time-to-market

[9] This chapter is based on excerpts from two previously published papers: [MK16] and [Mor+16a].

is one of the primary goals of the semiconductor industry. The increasing complexity of the applications is responsible for several changes in the design process organisation and development.

The current RTL languages (e.g. VHDL and Verilog) emerged in the 1980's and have been used for ASIC (originally) and FPGA (more recently) designs, [WO00] [GR94]. Despite their success, the RTL description needs an advanced hardware knowledge and a different modelling paradigm, which is not easily learned by the designers (several of them with a software background). Also, the translation of an application to the RTL domain requires the knowledge of both application and hardware details. To reduce the design costs, High-Level Synthesis (HLS) methods have been developed for many years. HLS grew in the last years as a solution to the increasing complexity of hardware implementation of new applications.

The standard HLS design flow starts with the application's implementation in a high-level programming language (C/C++), or even using graphs, DSLs or other abstractions [Geo+13]. After the validation of the software implementation, the hardware is synthesised in several steps, in which the designer must interact with the tool, to choose among different design alternatives, until the design constraints are met, Fig.15.1. The advantage of HLS for the design process is mainly due to the abstraction level of the languages used, in contrast to RTL description languages (VHDL/Verilog) [SS15]. However, the design space is sometimes so vast, that the designer needs a good knowledge of the application, the tool, the hardware structures and the desired technology, to achieve the performance goals [Con+11].

To help the designers to achieve faster a proper hardware implementation of the desired algorithms, we propose a more straightforward method. The designer implements the algorithms as tile-codes and, after validating them (simple software validation), a Task-Graph is extracted from the original C code. Our Task-Graph Creator (TGC) tool can create different node types, covering the main structures of the language, as shown in Part III. For each node type, one or more IPs should be established and stored in an IP Database.

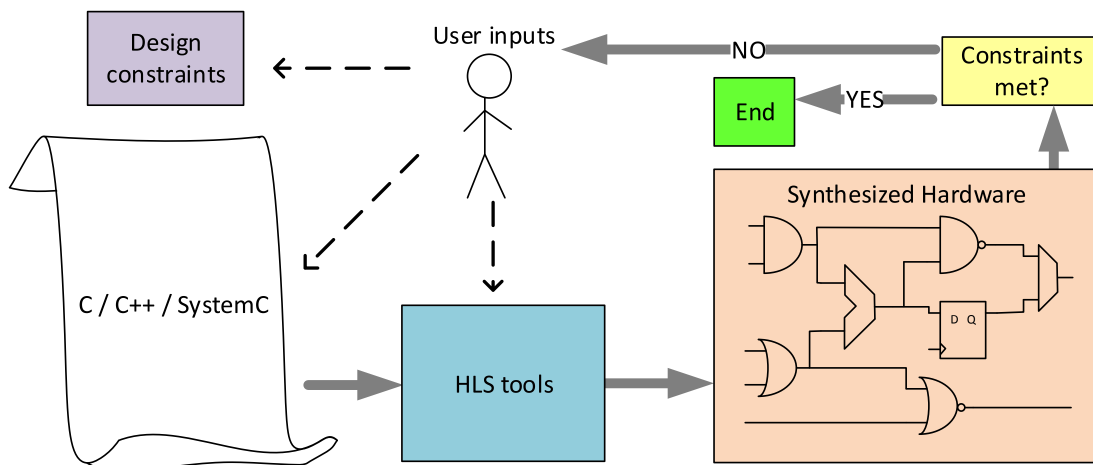


Figure 15.1: Typical High-Level Synthesis development flow.

Considering the design constraints specified (resources available, power consumption, throughput and so on), a Combinatorial Optimization Problem (COP) is solved, to identify which combination of nodes can meet the constraints. If the constraints are not fulfilled, the designer then must use any common EDA tool to identify and create new IPs.

15.1.2 System Development

As already explained, the goal of this work is not to develop a new HLS tool. Our objective is to have a fast prototyping method to help the inexperienced designer in achieving good performance results in an easier way. Figure 15.2 shows the proposed method.

The user needs to define the design constraints and the application's implementation using a subset of ANSI-C language. To handle more complex applications (composed of several functions), a Function-Graph is created by analysing the function calls in the application's source code. A Task-Graph is then created from the Application's Code, only for single isolated functions. Searching in a Database with hardware models for each block, the best combination is found using a simple optimisation algorithm.

This combination is mounted as a hardware version of the Task-Graph, which can then be synthesised to verify if the constraints are met. If not, the user must

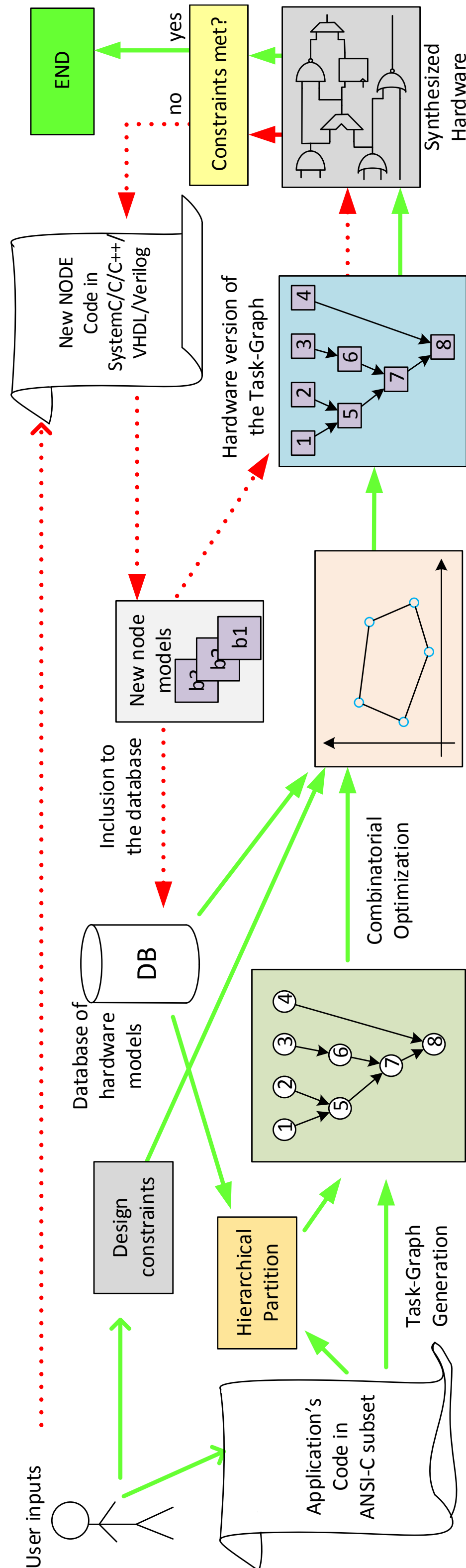


Figure 15.2: Flow of the proposed method: (a) in Green (continuous line), the straightforward flow, with few user interaction; (b) in Red (dotted line), the flow to create new Node models (same as the Common Flow from Fig.15.1).

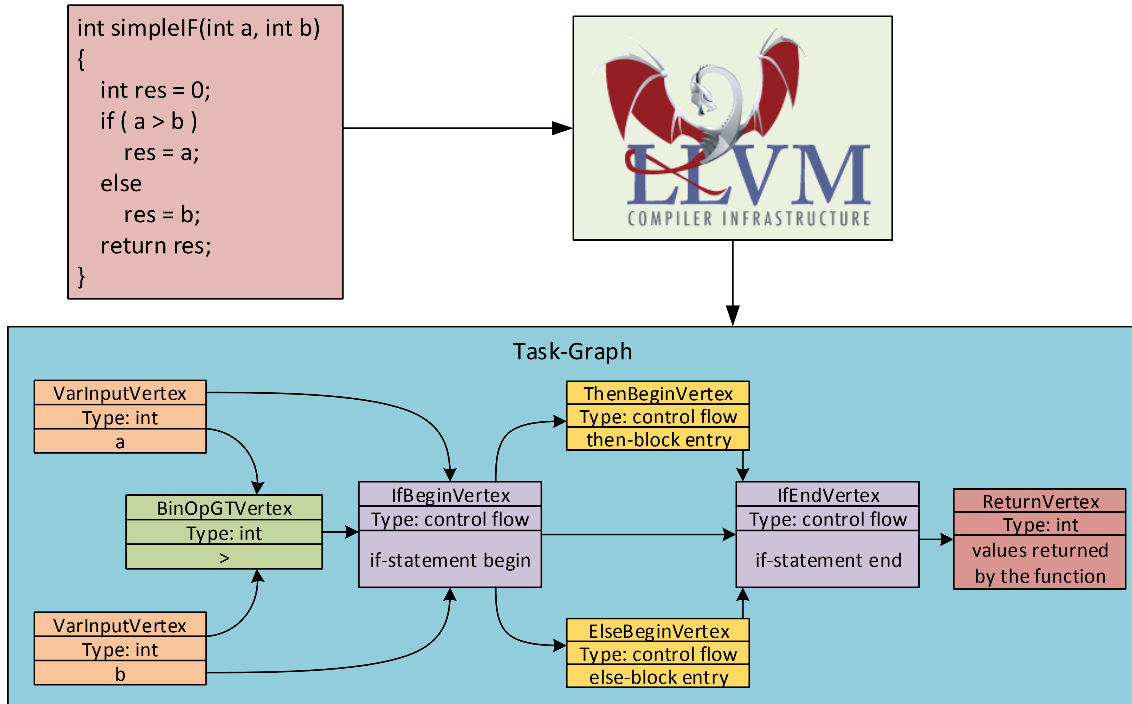


Figure 15.3: Task-Graph for a simple example.

identify the problematic(s) node(s) and create new hardware models for it (them). This model is saved in the Database for future utilisation and used in a new hardware Task-Graph. As in the common flow (Figure 15.1), the user can create the new models in any way: HLS, VHDL/Verilog and so on. The next sections explain details about each part of the system.

Task-Graph Constraints for HLS

The most important part of the flow is the creation of a Task-Graph based on the Application's Code developed by the user. The current version of the Task-Graph Creator tool supports only a subset of the ANSI-C language. However, this limitation is present also in the commercial/academic HLS tools [SS15]. The TGC tool is based on the front-end of the LLVM framework, and the application can be developed and tested as a common software on the user's workstation. The TGC uses the CLANG Code Refactoring tools, mainly the Abstract Syntax Tree (AST) Matcher [LA04b].

Input and Output Nodes: In Figure 15.3, it is possible to see nodes for inputs

(*VarInputVertex*) and for outputs (*ReturnVertex*). The output nodes are, in general, simple registers. The input nodes can have different configurations depending on the underlying technology. In Section 15.1.3, some possibilities are shown, based on Xilinx Vivado HLS tool. As mentioned before, the Task-Graphs are created for each function, and in the case of a *void* function the output node, instead of a register, is composed of a single bit. This scheme works as a flag, being activated at the function end.

Binary Operator Nodes: The *BinOpGTVertex* in Figure 15.3 represents the binary operator "Greater Then" and outputs a boolean value. The binary operators are responsible for all arithmetical and logical operations found in ANSI-C language. These operations, in hardware, can be implemented in several different ways, e.g. LUTs and DSPs in the FPGA technologies.

Branch Nodes: The example presented in Figure 15.3 contains an **IF-clause**, which is represented by three nodes:

IfBeginVertex it works as a simple multiplexer to branch the flow depending on the (single bit) input received from the previous node. The activation of an **IF-clause** is a logic value generated by a boolean expression. In the example, the boolean expression is composed by a single expression ($a > b$).

ThenBeginVertex* and *ElseBeginVertex they have the same internal structure and interfaces. The names are different only for the sake of avoiding misunderstandings during debugging and simulation. In Figure 15.3, these nodes appear compacted. In the hierarchical Task-Graph generation, these nodes are considered as *void* functions and may have internal Task-Graphs.

IfEndVertex it is just and endpoint for the **IF-clause** end.

Figure 15.4 shows a partial graph of and **IF-clause** with a more complex input expression ($(a < b)\&((c - b) > 0)$). As can be seen, the expression is first evaluated and then its output value is used as input to the *IfBeginVertex* node. This solution

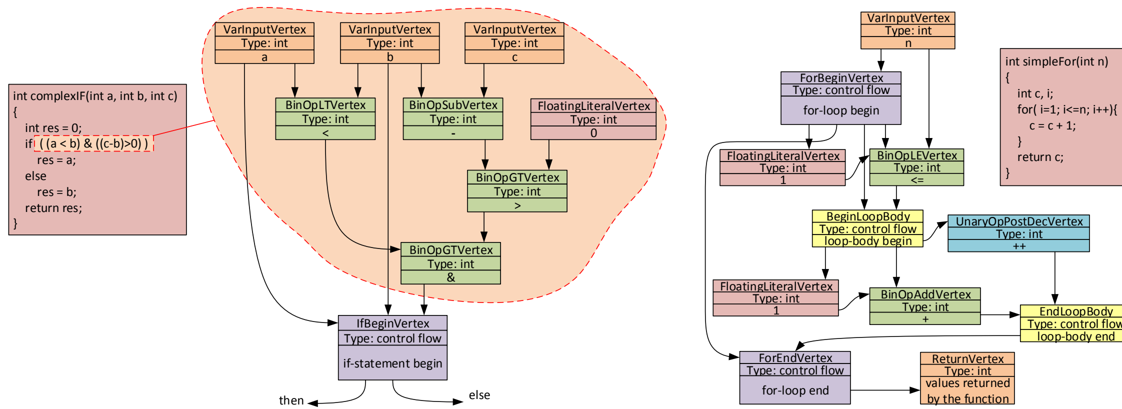


Figure 15.4: Left: Partial Task-Graph of a complex IF-clause, highlighting the expression; Right: Sample Task-Graph of a For-Loop.

provides flexibility to handle even more complex expressions.

For-Loops: A simple **For-Loop** Task-Graph can be seen in Figure 15.4. It is composed by the following nodes:

ForBeginVertex this node is the starting point to the loop and responsible to control the iterations.

BeginLoopBody this node receives and "start signal" from the *BinOpLEVertex* which handles the loop conditional. This node contains the loop body, composed by any expression or function calls.

UnaryOpPostDecVertex this node increments the loop counter, sending its value to the *EndLoopBody*, which represents the end of the loop body.

ForEndVertex this node represents the end of the loop, indicating if the execution's control goes to the next node or back to the *ForBeginVertex*.

Combinatorial Optimization

A Task-Graph contains a set of nodes and interconnections. Each node can be of different types, implementing different functions (Figure 15.4). For each node type, a set of equivalent hardware models were created (Figure 15.5). The hardware models

related to a node have the same behaviour, so they implement the same functionality. However, they were created with different optimisation options, what means that they have different characteristics of Resource Utilization, Power Consumption, Latency, Operating Cycles and Maximum Frequency.

Using the Task-Graph as input, several alternative solutions (different combinations of the hardware options) are mounted. Each of them represents a Hardware Task-Graph with the same functionality as the original Task-Graph and C-code. Given the Design Constraints, the best combination must be identified among the solutions (the Hardware Task-Graphs). It is a Combinatorial Optimization Problem, which can be solved by just computing the costs for each solution and comparing with the constraints. The best solution will be the one which best fits under the Design Constraints, Figure 15.5.

The analysis of each combination can be time-consuming, depending on the size of the Task Graph and the number of hardware options for each node. However, a pre-analysis can be performed, to eliminate Hardware Nodes options based on the constraints. In the case that no solution met the Design Constraints, then the user/designer must perform a classical timing analysis to identify bottlenecks and new optimisation possibilities. After this, new hardware models for the Task-Graph nodes can be created and inserted in the DataBase.

Hierarchical Partition

As explained in Part III, a task can have different granularities and complexities in the task-graph representation. It can be a simple arithmetical operation, as well as a complex function. Using the LLVM Front-End, all the functions used are identified. The Hardware Models Database stores not only Graphs Nodes but also Functions. This approach is used to improve code reusability in the same way as in software development and minimise the time needed to find a hardware implementation for the complete application. New functions can be created and stored in the database. However, certain aspects must be observed, as the naming rules and the behaviour

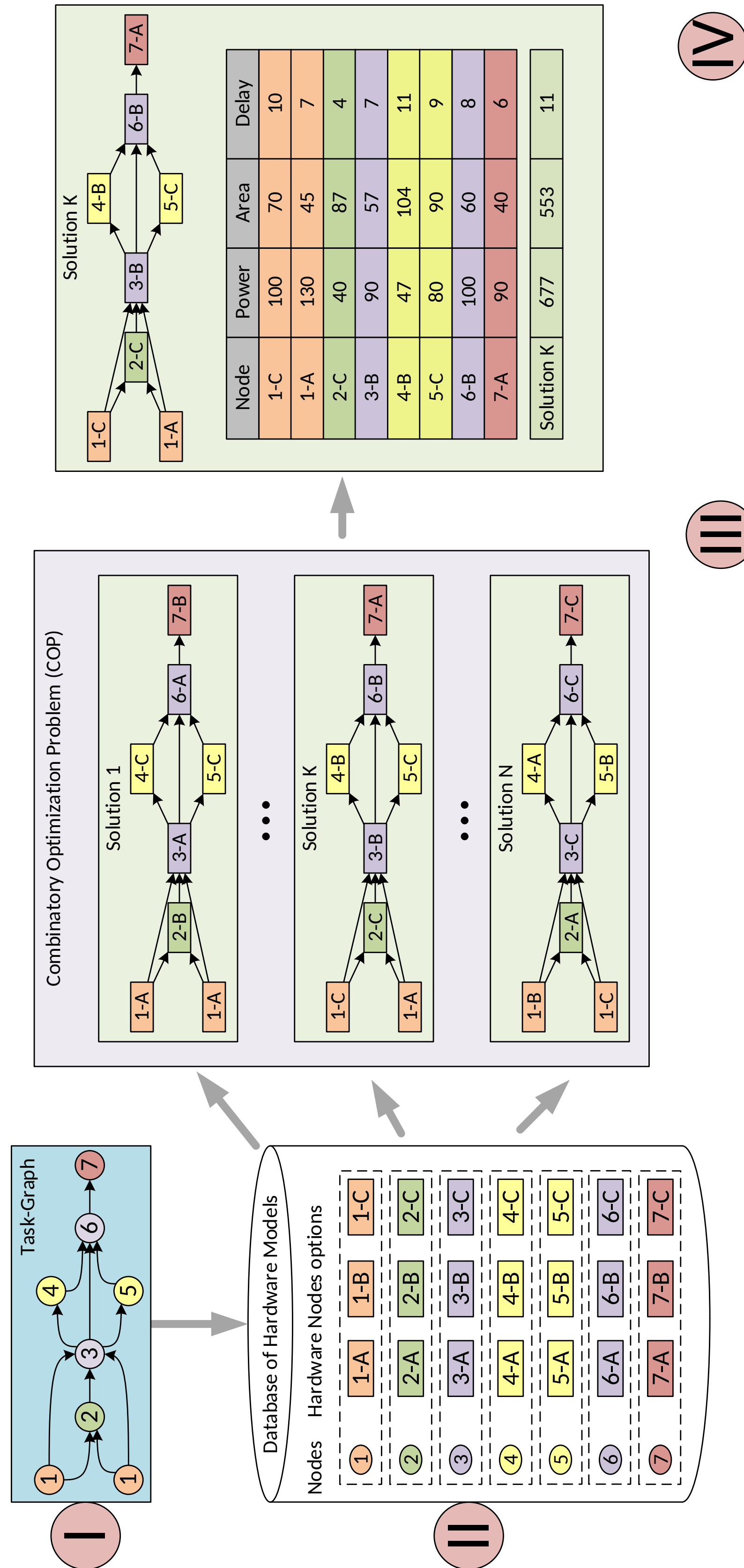


Figure 15.5: The Combinatorial Optimization Problem solved to select the best design alternative under the Design Constraints.

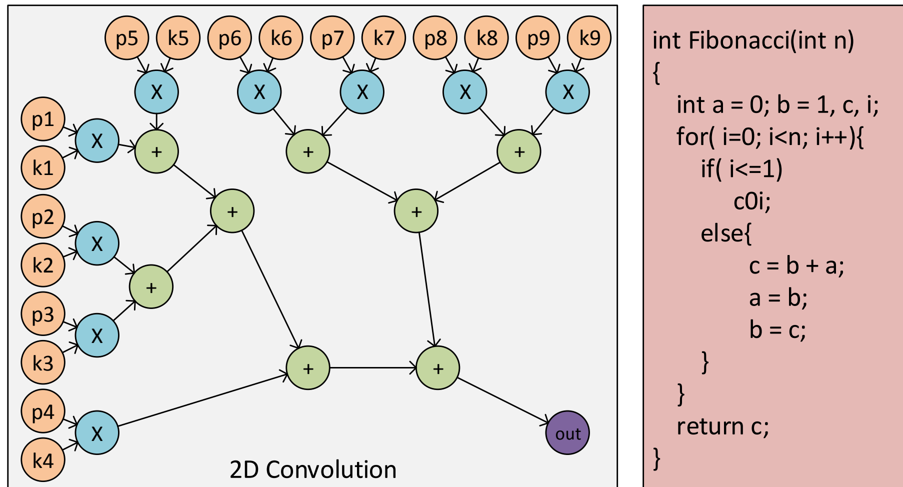


Figure 15.6: Use-cases selected: Convolution and Fibonacci.

verification, to assure the correctness of the solution. Although this step is performed before the Task-Graph Creation, we preferred to explain it here, since it is simple and straightforward.

15.1.3 Use-cases

We analyse two use-cases in this section to show the feasibility of the proposed solution. The Xilinx Vivado HLS tool was used to create Hardware Functions and Hardware Nodes for some nodes and functions. We implemented two different applications: (a) 2D Convolution for image filters and (b) a Fibonacci number generator. The two applications are simple, however, the 2D Convolution shows the exploration of data-flow applications and the Fibonacci shows a branch (**IF-clause**) inside a loop (**FOR-loop**), Figure 15.6.

Considering that branches, loops and arithmetical expressions can be used to implement a vast set of applications, there is no loss of generalisation with the use-cases selected. Table 15.1 shows some models from the database, which can be used to create the Hardware Task-Graph for the 2D Convolution and the Fibonacci generator. Alternative models can perform the same function at different costs. These models were created in Vivado HLS and differed in the optimisation and synthesis configurations used.

The Combinatorial Optimization aims to combine the different possibilities to

Table 15.1: Alternative Hardware Nodes on the Xilinx Xc7vx690tffg1761-2 FPGA device

Node name	DSP48E	FF	LUT	Freq. (MHz)	interval (cycles)	Power (mW)
add-v1	0	0	32	537	1	1
add-v2	1	0	0	443	1	1
mult-v1	3	20	18	242	6	5
mult-v2	3	19	17	242	1	6
mac-v1	3	53	49	246	4	5
mac-v2	3	148	49	243	1	7
simpleIF-v1	0	0	32	373	3	1
simpleIF-v2	0	2	16	118	2	1
simpleFOR-v1	0	0	28	332	5	1
simpleFOR-v2	0	8	10	289	2	1

find the one which best fits under the design constraints. Table 15.2 presents a comparison of the synthesis results for implementations using Vivado HLS with the ones generated using our method. The direct implementations were done by developing the application in Vivado HLS, as a common user would do. The other implementations were created as a block diagram in Vivado using the Hardware Nodes from Table 15.1. Figure 15.7 shows a comparison of the following three rows of Table 15.2:

- direct imp.1: common implementation using Vivado HLS to describe the application.
- pred-conv-add2-mult2: the result expected after the Optimisation part and before synthesis.
- conv-add2-mult2: the result of the Hardware Task-Graph after synthesised in Vivado.

We predict the solution (pred-conv-add2-mult2) by adding the resources, the interval and power characteristics of the Hardware Nodes. The frequency is estimated by the smaller value, which should be the bottleneck. There can be observed that the prediction is quite accurate with the synthesised Task-Graph. We must highlight here that the direct implementation was done without using any optimisation

options offered by Vivado HLS. This was done to show the first result an inexperienced user would have when using both tools: Vivado HLS and our method. The synthesis results (Table 15.2 and Figure 15.7) show that our method can achieve results similar to the direct implementation ones, with the advantage of having a shorter design time.

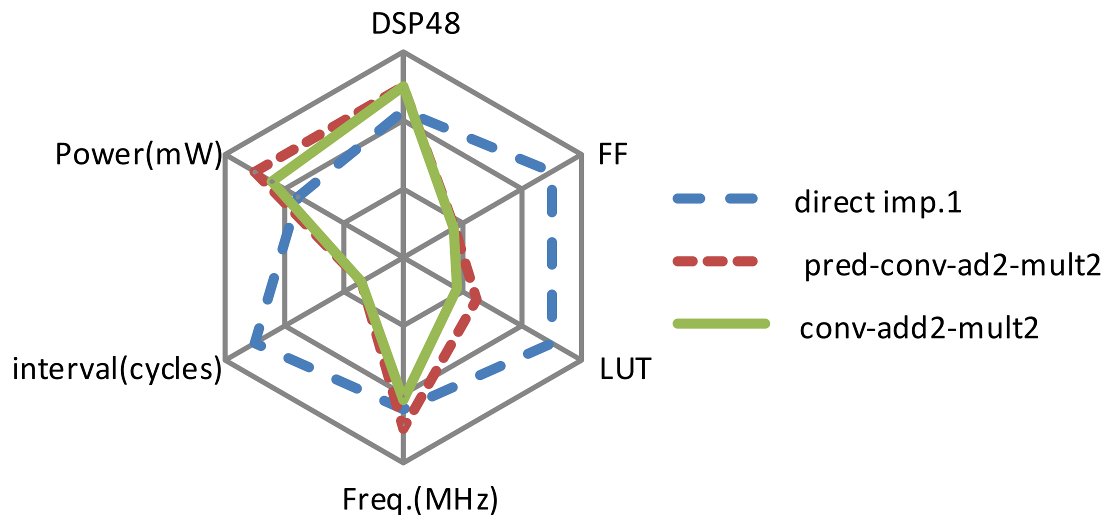


Figure 15.7: Comparison among the Design Constraints, a Direct Implementation in Vivado HLS and our method.

Table 15.2: Results of direct implementations and of our method on the Xilinx Xc7vx690tffg1761-2 FPGA device

	Version	DSP 48E	FF	LUT	Freq. (MHz)	Interval (cycles)	Power (mW)
convolution	direct imp.1	30	497	309	215	7	44
	direct imp.2	30	814	306	222	1	57
	conv-add1-mult1	27	180	418	224	7	51
	conv-add1-mult2	27	153	391	183	2	51
	conv-add2-mult1	35	153	135	214	2	54
	conv-add2-mult2	35	168	112	202	2	55
	pred-conv-add2-mult2	35	171	153	242	2	62
	conv-mac1	37	136	450	228	4	32
	conv-mac2	27	477	450	238	1	44
fibo.	direct imp.	0	129	128	156	5	4
	generated	0	152	115	167	9	6

15.1.4 HLS in the Many-Core architecture

Figure 15.8 shows the concept of using the HLS approach to generate hardware versions of the task-graphs. Each block in an Image Processing and Computer Vision (IP/CV) processing chain can be written as an independent *Tile-Code*, and will be transformed into a hardware block.

The *Task-Graph Storage* in the picture will store all the needed hardware blocks. The *Manager* will monitor the execution of these blocks, exchanging one to another. This exchange can be done by merely multiplexing or, if fast enough, dynamic partial reconfiguration, like in an FPGA.

The *Input Manager* will actuate as a data buffer, storing the pixels transferred from other tiles and, depending on control signals from the *Manager*, it will assign to the task-graph inputs the needed values.

The *Local Memory* encapsulates the Pixel Memory and a Register File used to store constants and coefficients needed in the task-graphs.

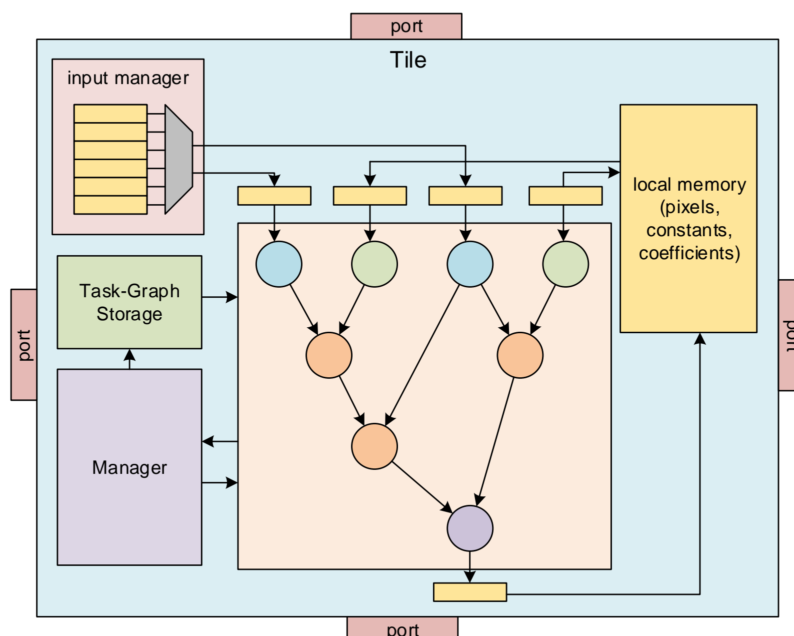


Figure 15.8: Tile architecture for the HLS approach.

15.2 The ASIP Approach

In this section, an ASIP design is provided. We start from a standard RISC processor and, with descriptions in the LISA Architecture Description Language (ADL) in the Synopsys Processor Designer tool, we develop an ASIP with a specialised input stream buffer and a novel direct injection of data into the processor's data-path.

15.2.1 Introduction

A common IP/CV application is composed of a sequence of steps, as shown in Figure 4.1. The initial steps (Acquisition, Pre-Processing, Segmentation) operate over pixel data, transforming an image into another image. The step 4 extracts information from the image and creates feature vectors, which will be interpreted by the final step (5). Some applications will need more or fewer steps.

The most common way to acquire an process and image, considering a camera as input, is: (1) the camera sends a pixel stream to the processor; (2) the processor acquires the pixels and stores them in the main memory; (3) the processor loads the pixels from the main memory to process. We repeated the steps (2) and (3) for all Intermediary Images. The Intermediary Images have the same size as the original image. Depending on the application's complexity - more complexity implies more steps in the processing chain - the memory consumption can be large enough to surpass the available memory.

To optimise the memory utilisation, we developed a memory partition model specifically for IP/CV chains. Also, an input-to-datapath scheme was used to speed up the processing by connecting pixel buffers directly to the executing stage of processor's pipeline. Also, a Peephole Optimization leads to reduce the total number of Cycles-Per-Instruction (CPI) by grouping sequences of simple instructions in new customised complex instructions.

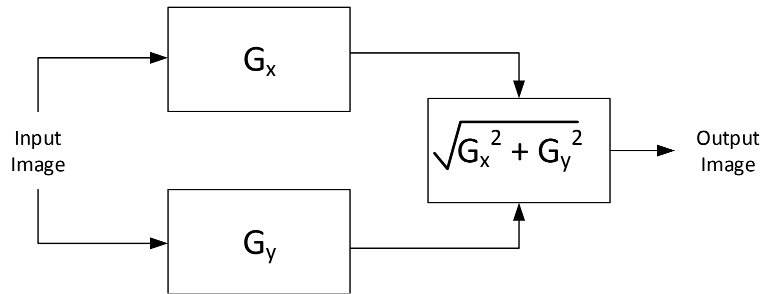


Figure 15.9: Processing chain of the Sobel edge enhancement algorithm.

15.2.2 Application's Analysis

A common IP/CV application, the Sobel algorithm, is depicted in Figure 15.9. This algorithm's execution needs to store four images. An important feature to highlight is the type of operations performed in the algorithm. In the Sobel algorithm, as well as in most of IP/CV algorithms, point and neighbourhood operations are performed. The classification is based on how much pixel locations are needed to compute the operation's output [MH14]. A point operation requires only one (x,y) location. A neighbourhood operation requires a region surrounding the (x,y) location.

15.2.3 ASIP Design

This section shows the design of the new ASIP architecture. The first topic describes the EDA tool used to develop the processor architecture. Afterwards, a description of the standard 32-bit RISC processor architecture. Then the Memory Partition is explained in details, followed by the Instruction's Creation and the Input-to-Datapath scheme.

The Synopsys Processor Designer

In this work, the design methodology offered in the Synopsys Processor Designer toolset was used (Figure 15.10). The toolset requires the complete LISA (Language for Instruction Set Architectures) description of the architecture, containing all internal structures, control signals, instruction's behaviour and so on. With the LISA description, the tool can generate both functional and cycle-accurate instruction set

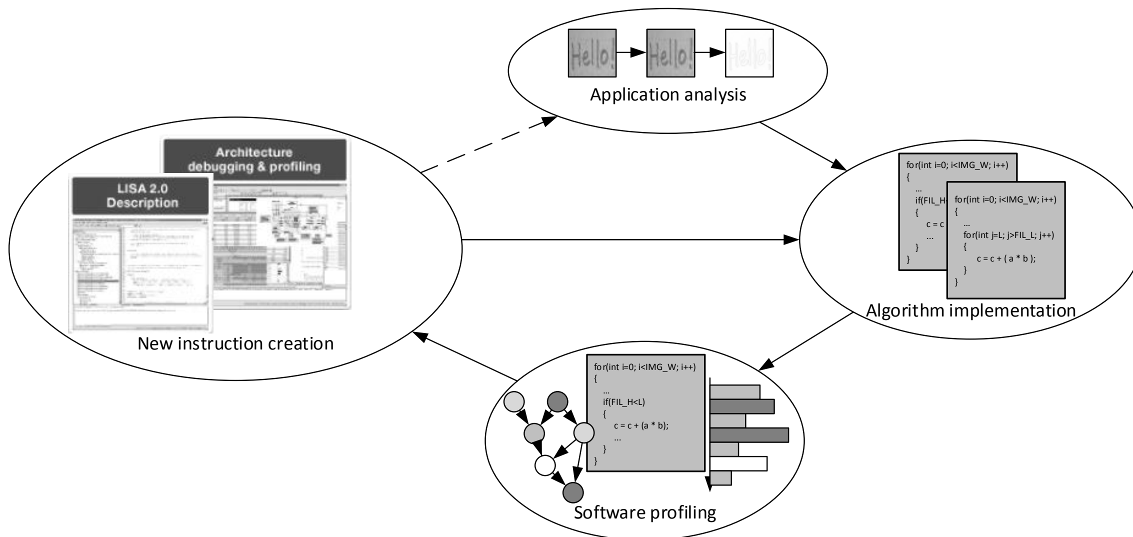


Figure 15.10: The ASIP design methodology followed in this work (adapted from [MKH15a]).

simulators (ISS). A compiler based on Cosy [htt] or LLVM [LA04a] can be generated and manually optimised by the designer. A detailed debugger tool is also available, providing data about the instruction's performance, as well as information regarding pipeline usage, register and memory allocation, hazards occurrence, cache performance and so on. It is also possible to generate a synthesizable RTL description of the processor, in both VHDL or Verilog languages [Wu13]. Figure 15.10 shows the classical ASIP design methodology: by successively loop among the stages shown in Figure 15.10, the processor architecture will be manually refined until an acceptable result is obtained. The literature presents several advanced design methodologies; however, we followed the original one because the focus of this work is not on the methodology itself but in the proposed architecture enhancements.

PD-RISC Processor Architecture

The PD-RISC processor is a standard RISC processor, shipped with the Synopsys Processor Designer tool to be a starting point for new architecture designs. It is a 32-bit load-store architecture, with a 6-stage fully-bypassed pipeline and separated ALU and Multiplication units, Figure 15.11 (for simplicity, the picture does not show the PreFetch and Fetch stages). PD-RISC is similar to some conventional

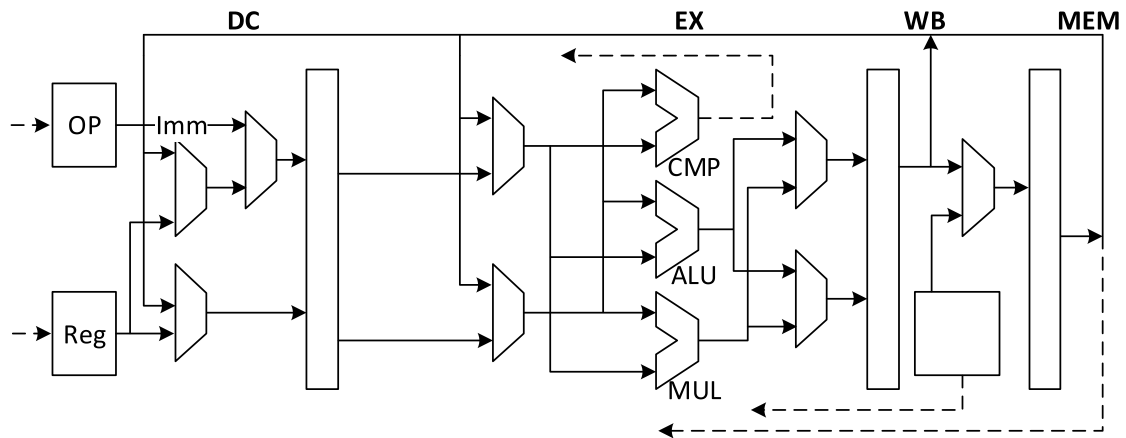


Figure 15.11: PD-RISC processor architecture (adapted from [Wu13]).

processors (such as DLX, MIPS, OR1K, RISC-V), so the analysis performed over it is general enough to be applied to other similar processor architectures. We use the original PD-RISC as a benchmark to compare the newly proposed architectures with the standard one. There is an instruction to get values from inputs in the PD-RISC. It works like a **load** instruction: it reads the value from the desired input register and writes to the Register File (RF). The I/O Controller has a single configuration parameter for the inputs: the sample period. To set a value to output, we use another instruction, similar to a **store** instruction: it reads the value from the RF and writes to the output register.

Memory Partition

To optimise the memory use, we must take a look at the application's specific needs and understand in more details how the algorithms use the pixels in each step of the processing chain. Observing the processing chain in Figure 15.9, we can see the dependencies among each step and the previously needed images. In the majority of cases, the image is needed only in the subsequent step, and then can be discarded.

To improve the memory efficiency, even more, we divided the processor's memory space into two logical parts: one for pixels and the other one for any other variable in the program. Besides the logical partition, we also implemented physical partitions. In the literature, we identified an efficient architecture for image acquisition (Figure

15.12). The Neighborhood Loader (NL), after an initial filling latency, displays a new neighbourhood at each pixel_clock cycle. A new entire neighbourhood is loaded every cycle after the initial delay, and the already processed pixels are taken away, saving memory and reducing the total amount of instructions needed by the application. The size of the NL depends on the desired region. In this work, we considered only 3x3 neighbourhoods. We can insert an NL before each Neighborhood Operation. If the step is a Point operation, then a simple buffer is used [MLB12].

With this approach, we have the processing chain synchronised by the pixel_clock from the camera. We described the NL architecture in LISA language and included it in the processor's architecture. It is a long shift register that works as a peripheral, receiving and storing the pixels automatically from the input pins, without using the processor's datapath. It has a controller synchronised with the camera's pixel_clock. To have access to the available neighborhood from the program, we created a new instruction (*getnl*, Figure 15.13). This instruction was then added to the PD-RISC's ISA, generating a new processor hence called NL-RISC.

Then we included this instruction in the compiler library as Compiler Known Function (CKF), mapped directly to the new instruction. For each pixel needed from the neighbourhood, the instruction must be called. The *getnl* instruction maps a value from a specified NL location to a program variable. The implemented NL can be used receiving values from an input pin, as well as from the Register File. Another instruction (*setnl*) takes a value from the Register File and writes to the NL input.

Instruction's Creation

In this topic, we show a simple procedure to create new instructions. The method shown in [FH14] was used to identify sequences of instructions to be substituted by a single complex instruction. It is similar to the well-known Peephole Optimization found in compiler's structures. The steps followed were (Figure 15.13):

1. Profile the application, to find the most repeated sequences of instructions.

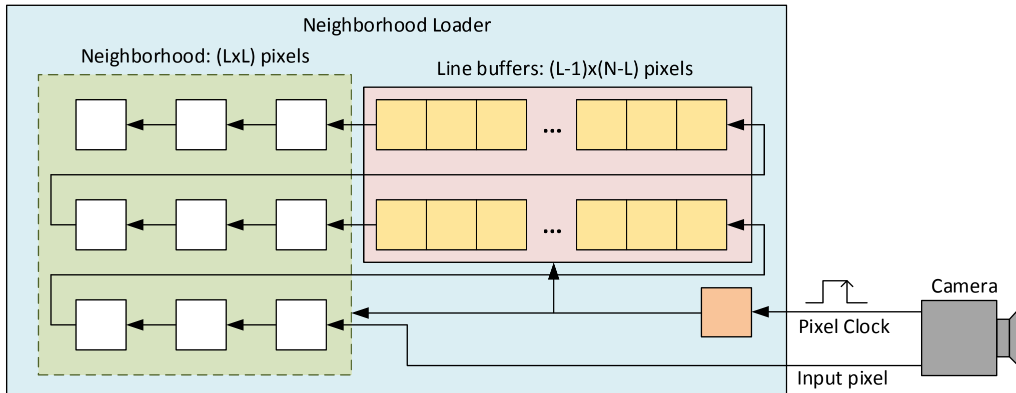


Figure 15.12: Neighborhood Loader architecture.

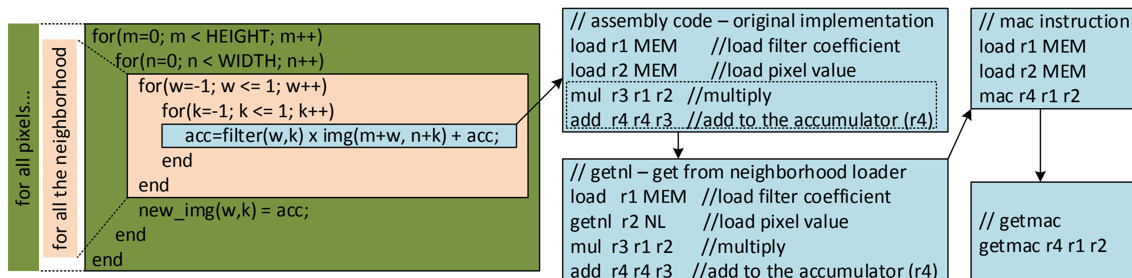


Figure 15.13: Assembly code reduction achieved by creating new instructions.

2. Determine the C code equivalent to the identified sequence of instructions.

3. Create a new instruction to give the same result as the original sequence.

4. Modify the compiler to add a CKF related to the new instruction created.

5. Substitute, in the C source code, the corresponding lines by the new CKF.

The most repeated sequence of instructions (Figure 15.13) identified in the analyzed application is equivalent to a *Multiply -And -Accumulate* (MAC) operation, well known in DSP domain. The new instruction created, *MAC* can operate a single clock cycle, faster than the PD-RISC processor. This instruction was added to the NL-RISC processor, generating a new architecture hence called NL-MAC.

Input-to-Datapath

We developed the NL-RISC and NL-MAC architectures by adding new features to the primary processor. In this section, it is proposed a fusion of both ideas, to reduce the control overhead due to separate hardware blocks. A single new

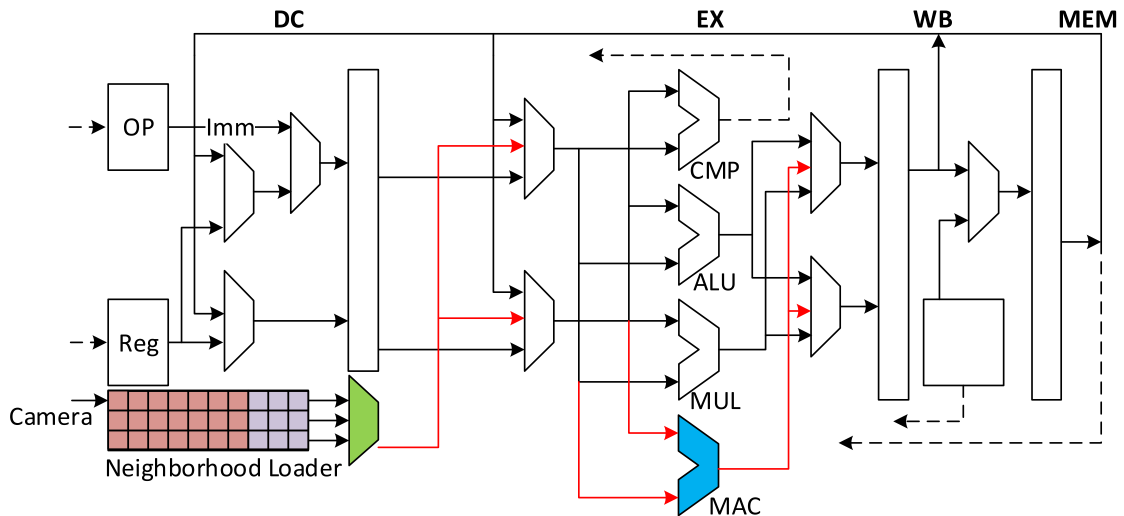


Figure 15.14: PWA architecture: with the Neighborhood Loader and input-MAC unit.

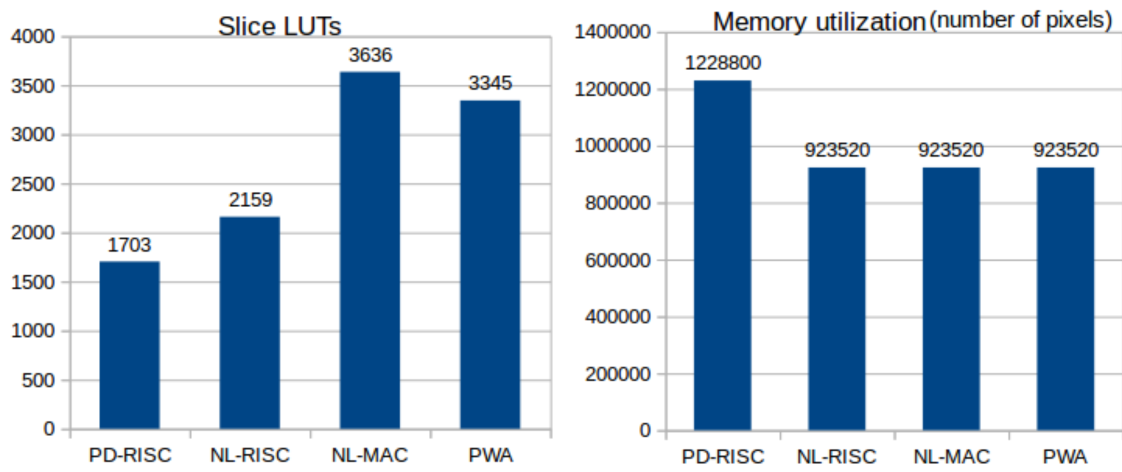


Figure 15.15: Slice LUTs used by each architecture.

instruction, *GETMAC* was designed to, in a single cycle, get the pixel value from the NL and compute a *MAC* operation (Figure 15.13). Its CKF is called with three input parameters: an accumulator; filter coefficient; position in the NL. This architecture is hence called PWA (Process-While-Acquiring), Figure 15.14. We can see a direct connection from the NL to the MAC block in the picture.

15.2.4 Results

The area is an essential aspect of an integrated circuit since it relates directly to the chip's fabrication cost. In our analysis, Look-Up Tables (LUTs) were considered for FPGA synthesis. In the chart of Figure 15.15, it can be seen that the difference in the area between the PD-RISC and the NL-RISC models is 27%, due to control overhead. The NL itself was implemented using BRAMs, instead of LUTs. The addition of the MAC instruction to the processor resulted in an area increase of 68%. The complexity of the new instruction is responsible for this overhead.

The PWA architecture - by tightly integrating the Neighborhood Loader and the MAC instruction - could reduce the control overhead in 8% over the NL-MAC architecture. Memory utilisation is another crucial aspect of the design of real-time IP/CV systems. The chart in Figure 15.15 presents the total amount of memory used by each architecture. It is important to highlight that we did not measure the memory utilisation in Bytes, but in the number of pixels since the pixel depth can vary. The values were determined for an image with 640×480 pixels, considering the application in Figure 15.9.

In that application, four images are used/generated. The inclusion of the Neighborhood Loader was responsible for reducing the total amount of pixels stored in ca. 25%. Another aspect is that this metrics is not taking into account the extra memory needed for the program (variables, temporary allocation, and so on). We can reduce the main memory's size since we are not storing the pixels there anymore. Figure 15.16 shows the maximum operating frequency achieved by each architecture in the technology selected for synthesis (Xilinx xc7k325t FPGA). We can see a variation of less than 10%. Considering the small variation among the architectures, we searched for other speed metrics.

The number of cycles per pixel is shown in Figure 15.16. We can see a considerable reduction (ca. 43%) from the PD-RISC to the NL-RISC, indicating the impact of reducing the memory access (in this case, due to the Neighborhood Loader). The new MAC instruction contributed to a reduction of ca. 29%, showing that the

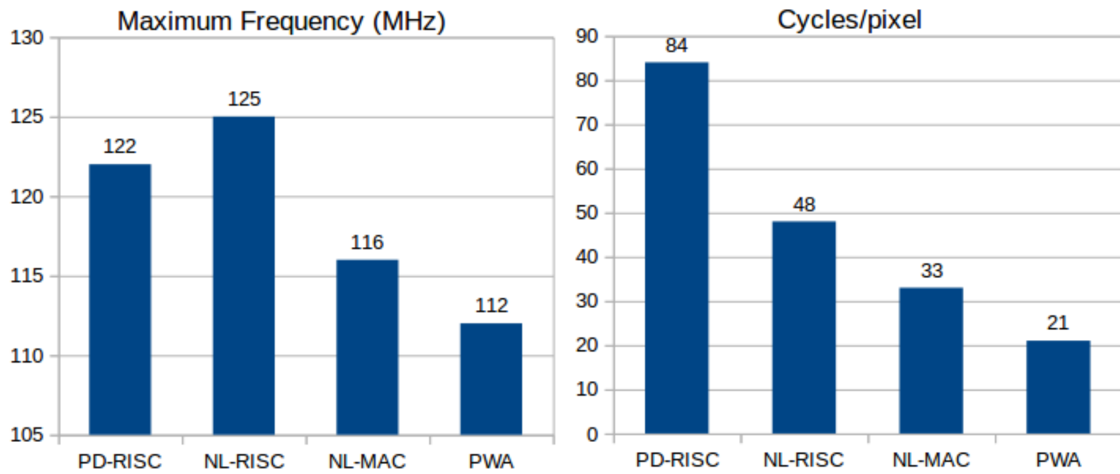


Figure 15.16: Maximum frequency achieved by each architecture.

methodology used to identify the sequences of instructions succeed. In the PWA architecture, the combination of acquisition and processing could enhance the NL-MAC architecture in ca. 36%.

The chart in Figure 15.17 shows how many frames per second can be processed in each architecture, running at the same frequency (100Mhz) and running at the maximum frequency (Figure 15.16). We can observe that the reduction in the number of cycles per pixel (Figure 15.16) is responsible for an enormous increase in the frame rate achieved by each architecture.

The power consumption is one of the most critical issues in the IP/CV domain. Figure 15.18 shows the power estimation after synthesis, for each architecture. We can see that the new architectures have a mean power consumption up to 60% higher. However, this comparison can be made if the applications are running for the same amount of time. Considering the specific application we are analysing, we must use a different metric to compare the architectures.

The chart in Figure 15.18 shows the energy consumption for each processed pixel. This metric is justified because after the processor finishes the processing of an image, it can stay idle until the next image. This metric also includes the time of processing, showing the power efficiency of each architecture. Even with an increase in the area, the addition of a Neighborhood Loader increased the efficiency in 29%

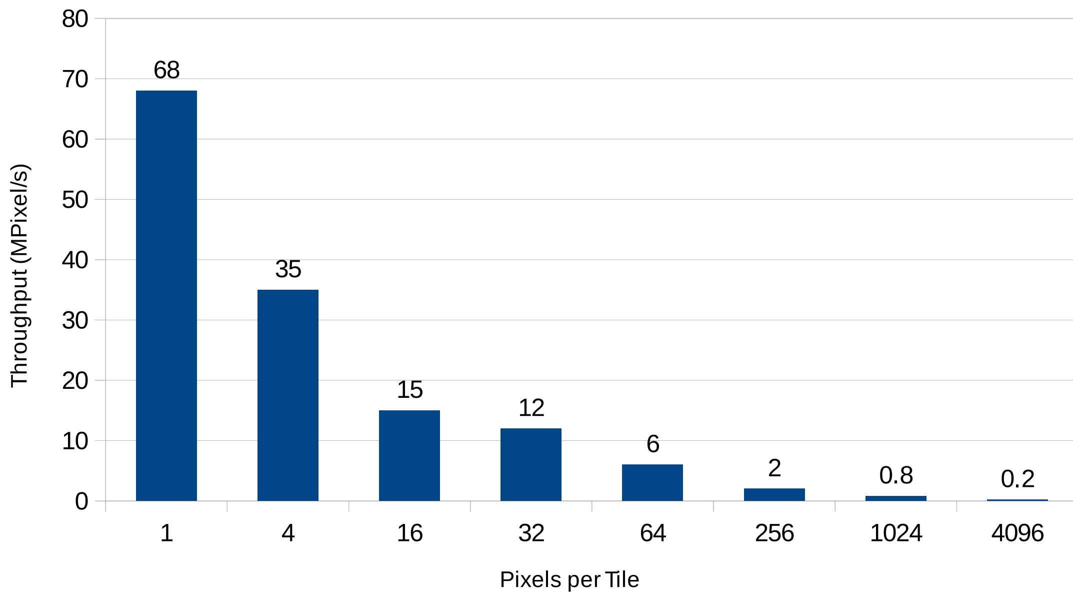


Figure 15.17: Frame rate achieved by each architecture.

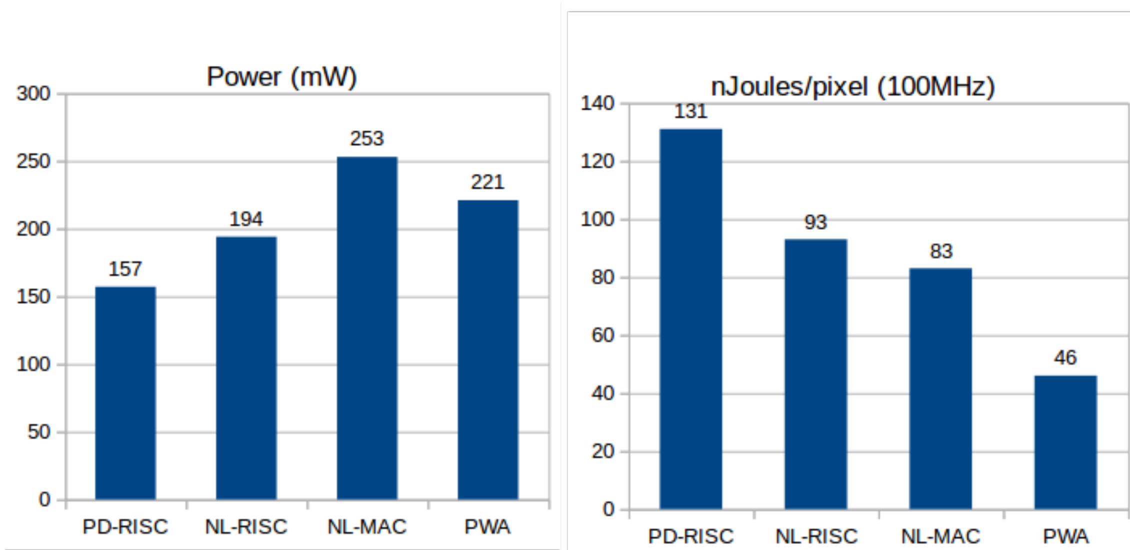


Figure 15.18: Power consumption estimated for each architecture.

(from PD-RISC to NL-RISC).

The new MAC instruction reduced the energy use in 10%, and finally, the PWA architecture offered a reduction of 44% in the energy needed per pixel, over the NL-MAC architecture. It is important to highlight that this energy efficiency could be achieved mainly due to the reduction of the number of clock cycles per pixel, even with an increase in area and because the application has a high switching frequency, due to the massive amount of data.

Comparison with other works

Table 15.3 shows a comparison of the architectures presented in the current work and similar ones found in recent literature. The custom architecture from [Bro+11] presents the highest throughput. However, it is not programmable. The area occupied by our final processor (PWA) is half of the area of the Microblaze, the standard Xilinx softprocessor. Regarding the frequency, the PWA processor achieved a frequency higher than the $\rho - VEX$ processor. However, the $\rho - VEX$ is a VLIW, which offers a more significant Cycles/Instruction factor.

The throughput obtained by the PWA processor (Cycles/pixel) is higher than the $\rho - VEX$ and the Microblaze processors. However, in [HWA15b], can be seen that for higher resolution images, the VLIW processor surpasses the Microblaze processor by a factor of more than 3×. Considering that the PWA processor is a RISC architecture - like the Microblaze - we can expect that for bigger images, the $\rho - VEX$ processor would have a better performance. Another important aspect of this comparison is that [HWA15b] does not have an image acquisition stage. The authors there consider the image pre-loaded in the processor's memory. The results we present for our processors include the acquisition part from a camera.

Table 15.3: System's comparison

Work	Architecture	Device	Area	Frequency	Throughput	Power (mW)
current	PD-RISC	Xilinx xc7k325t FPGA	1703 Slices	122 (MHz)	84 Cycles/pixel	157
current	NL-RISC	Xilinx xc7k325t FPGA	2159 Slices	125 (MHz)	48 Cycles/pixel	194
current	NL-MAC	Xilinx xc7k325t FPGA	3636 Slices	116 (MHz)	33 Cycles/pixel	253
current	PWA	Xilinx xc7k325t FPGA	3345 Slices	112 (MHz)	21 Cycles/pixel	221
[Bro+11]	custom	Xilinx Virtex-6 xc6vlx75T	409 Slices	420 (MHz)	10 Cycles/pixel	n/a
[HWA15b]	VLIW $\rho - VEX$	Xilinx Virtex-6 xc6vlx240T	7506 Slices	75 (MHz)	183 Cycles/pixel	n/a
[HWA15b]	Microblaze	Xilinx Virtex-6 xc6vlx240T	7376 Slices	150 (MHz)	488 Cycles/pixel	n/a

This section shows the development of power efficient ASIP for Image Processing and Computer Vision Applications. Starting from a standard RISC processor, new

features were included in the micro-architecture, providing advantages and disadvantages. Each design step generated a new architecture, incrementally leading to a more efficient architecture.

The PWA architecture provides an enhanced throughput (4x cycles/pixel), higher frame rate (ca. 4x speed-up), less memory utilisation (25% reduction) and less energy consumption per pixel (2.8x reduction) than the original PD-RISC. However, it has a drawback: the overhead in the area (ca. 25%). A complete study must be performed with more complex applications and higher resolution images, to show how the proposed architecture would behave with, for example, longer image processing chains and/or more NLs.

15.2.5 The ASIP in the Many-Core Architecture

As we have discussed in this section, the development of specialised processors is somehow essential to handle the real-time constraints in IP/CV applications. Considering the architectures proposed here, all of them present interesting characteristics that could be integrated into our many-core architecture. We discuss each one on the following topics:

- **NL-RISC**: The Neighborhood Loader architecture is useful for pixel stream processing. However, considering the IP/CV processing chain, it can be efficiently used only for the first processing block. In this case, its area overhead makes this solution not suitable for our problem.

- **NL-MAC**: The addition of custom instructions to the PD-RISC processor has shown its advantages and a suitable cost. A more extensive study is needed to identify which instructions should be added to the processor's architecture to benefit a large number of processing blocks.

- **PWA**: The approach of inserting input data directly to the processor's datapath offers a huge reduction in the cycle count per output pixel, which can benefit a lot the throughput of our architecture. The main drawback is that

with this approach we would need extra hardware to handle synchronisation signals among the tiles.

15.3 Remarks

In this chapter, we proposed two different approaches to develop the Processing Elements for our Many-Core architecture. Several other techniques could be explored, like the utilisation of Very-Large Instruction Word (VLIW) processors (as simulated in the Chapter 13). Another possibility would be to explore other architectures. Embedded GPUs could be an interesting solution. However, the area overhead might be prohibitive. Transport-Triggered Architecture (TTA) architectures are also good candidates since there are reports in the literature about a more efficient area utilisation in comparison to VLIW processors. Automatic generation of processor models can also be achieved by some techniques presented in the literature. Our intention here was not to perform an exhaustive exploration, but to provide some insights in this direction.

Chapter 16

Conclusion and Future Work

Current trends in embedded and cyber-physical systems in industry and academy show that novel hardware architectures are required to provide the requested computational performance for modern applications, e.g. in the domain of image and signal processing.

It is a fact that Moore's law will end, caused by physical limits related mainly to the Dennard Scaling problem. Moreover, therefore, novel technologies and hardware system architectures, need to be investigated and provided to support the demand from industrial applications.

This thesis followed precisely this motivation by investigating an entirely new concept of a highly flexible, scalable and maximal parallel hardware architecture for that type of applications. The approach is different to traditional multi/many-core architectures, as well a to GPU-like hardware structures, since it includes a new concept for the communication infrastructure and therefore the data transfer and distribution, the processing elements and the programming model.

In this work, we developed contributions to all these three areas (communication infrastructure, processing element design and programming model) by developing the approach in three levels.

In the first level, we developed a method and tools to explore the programmability of the system, using the *Tile-Code* concept, and to analyse the application using *Task-Graphs.* This approach also leads to the development of a many-core simulator

able to perform fine-grained scheduling of tasks. The promising results of this part have shown that the distributed programming model based on *Tile-Codes* and the *IP/CV* processing chain are valid and convincing. However, they also indicate the need for another level of abstraction, to enable more detailed modelling and evaluation of the hardware structure.

An evolution of the first simulator was developed in the second level, as well as a more specific definition of the design space to be explored. The development of a SystemC-based model of the hardware architecture was investigated resulting that a novel communication over shared register banks was integrated. Also, a tool for estimation of power consumption, area and timing were integrated into the simulator, enabling the possibility for early estimation of hardware performance. The more inside view of the behaviour of the architecture led to the final abstraction level.

In the third level, we selected one of the hardware realisations from the second level to extract results of the FPGA hardware utilisation and provided a first view of the excellent performance and behaviour of the system. This realisation is only one demonstrator, for a feasibility study, which shows that the approach can be realised with the most modern FPGA architectures. However, the approach is not related only to FPGA. It is envisioned that in extensions of the work an ASIC synthesis with estimations of power and performance will be developed. Even a realisation on a System-on-Chip can be envisioned and should be considered.

The approach opens varieties of such a highly parallel architecture that almost endless numbers of instantiations for several algorithms can be developed. It would be of great interest to use the developed hardware structure for machine learning algorithms, especially for convolutional neural networks. Discussions with specialists from this domain brought out promising details since the hardware structure supports the structure of the machine learning algorithm.

Further experiments would be of interest in the domain of signal processing with, e.g. an array of radar antennas. Here, the massive parallelism and synchronous

data processing are of interest for specific MIMO radar algorithms. Finally, using FPGA as target platform would allow to scale and adapt the hardware during runtime by dynamic and partial reconfiguration. This feature would allow reacting to requirements of the algorithms and the environment of the system during operation.

After all, we have as the main result of this thesis the basis for several further research projects, in varied directions, such as applications of multi-dimensional signal processing; research in task-mapping and scheduling techniques; exascale computing; high-performance computing, and so on.

Part VI

Backmatter

References

[Abb99] Boyd A. Fowler Abbas El Gamal David X. D. Yang. "Pixel-level processing: why, what, and how?" In: *Proc. SPIE 3650, Sensors, Cameras, and Applications for Digital Photography.* 1999.

[AFI93] A Astrom, R Forchheimer, and P Ingelhag. "An integrated sensor/processor architecture based on near-sensor image processing". In: *1993 Computer Architectures for Machine Perception.* IEEE Comput. Soc. Press, 1993, pp. 147–154. ISBN: 0-8186-5420-1. DOI: 10.1109/CAMP.1993.622468. URL: http://ieeexplore.ieee.org/document/622468/.

[BD05] Luca Benini and Giovanni De Micheli. "Networks on Chips: A New Paradigm for Component-Based MPSoC Design". In: *Multiprocessor Systems-on-Chips* (2005), pp. 49–80. ISSN: 18759661. DOI: 10.1016/B978-012385251-9/50017-7.

[Bla+10] David C Black et al. *SystemC: From the Ground Up.* 2nd ed. Springer US, 2010, p. 279. DOI: 10.1007/978-0-387-69958-5.

[Bor07] Shekhar Borkar. "Thousand core chips - A technology perspective". In: *Proceedings - Design Automation Conference* (2007), pp. 746–749. ISSN: 0738100X. DOI: 10.1109/DAC.2007.375263.

[Bra+02] Ulrik Brandes et al. "GraphML Progress Report Structural Layer Proposal". In: *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers.* Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2002, pp. 501–512. ISBN: 978-3-540-45848-7. DOI: 10.1007/3-540-45848-4_59. URL: https://doi.org/10.1007/3-540-45848-4_59.

[Brä+13] Thomas Bräunl et al. *Parallel image processing*. Springer Science & Business Media, 2013.

[Bre+11] Andrew Z Brethorst et al. "Performance evaluation of Canny edge detection on a tiled multicore architecture". In: *Proceedings of SPIE - The International Society for Optical Engineering* 7872 (2011), The Society for Imaging Science and Technology (IS. ISSN: 0277786X. DOI: 10.1117/12.873004. URL: http://dx.doi.org/10.1117/12.873004.

[Bro+11] V. Brost et al. "Flexible VLIW processor based on FPGA for real-time image processing". In: *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*. Nov. 2011, pp. 1–8. DOI: 10.1109/DASIP.2011.6136855.

[Can+13] Andrew Canis et al. "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.2 (2013), p. 24.

[Can86] John Canny. "A Computational Approach to Edge Detection". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* PAMI-8.6 (Nov. 1986), pp. 679–698. ISSN: 0162-8828. DOI: 10.1109/TPAMI.1986.4767851.

[C+08] J. Ceng, J. Castrillon, et al. "MAPS: An Integrated Framework for MPSoC Application Parallelization". In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: ACM, 2008, pp. 754–759. ISBN: 978-1-60558-115-6. DOI: 10.1145/1391469.1391663. URL: http://doi.acm.org/10.1145/1391469.1391663.

[Che+11] Qian Chen et al. "Source code profiling for ASIP design: Strategy and implementation". In: *Electronics, Communications and Control*

(ICECC), 2011 International Conference on. Sept. 2011, pp. 1032–1035. DOI: 10.1109/ICECC.2011.6066550.

[Con+11] Jason Cong et al. "High-level synthesis for FPGAs: From prototyping to deployment". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 30.4 (2011), pp. 473–491.

[Cza+12] Tomasz S Czajkowski et al. "From OpenCL to high-performance hardware on FPGAs". In: *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE. 2012, pp. 531–534.

[DT03] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 0122007514.

[Dia+12] D. Diamantopoulos et al. "SPARTAN project: On profiling computer vision algorithms for rover navigation". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*. June 2012, pp. 174–181. DOI: 10.1109/AHS.2012.6268647.

[Edi17] Stephan Ramm (Editor). *The OpenVX Specification*. The Khronos Group Inc., 2017.

[ESÅ96] Jan Erik Eklund, Christer Svensson, and Anders Åström. "VLSI implementation of a focal plane image processor - A realization of the near-sensor image processing concept". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.3 (1996), pp. 322–335. ISSN: 10638210. DOI: 10.1109/92.532033.

[EYF99] Abbas El Gamal, David X D Yang, and Boyd A Fowler. "Pixel Level Processing — Why, What, and How?" In: *Electronic Imaging'99* (1999), pp. 2–13. ISSN: 0277786X. DOI: 10.1117/12.342849. URL: http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=979328.

[Elo+04a] A. Elouardi et al. "On chip vision system architecture using a CMOS retina". In: *Intelligent Vehicles Symposium, 2004 IEEE.* June 2004, pp. 206–211. DOI: 10.1109/IVS.2004.1336382.

[Elo+04b] a. Elouardi et al. "On chip vision system architecture using a CMOS retina". In: *IEEE Intelligent Vehicles Symposium, 2004* (2004). DOI: 10.1109/IVS.2004.1336382.

[EKC01a] R. Etienne-Cummings, Z.K. Kalayjian, and Donghui Cai. "A programmable focal-plane MIMD image processor chip". In: *Solid-State Circuits, IEEE Journal of* 36.1 (Jan. 2001), pp. 64–73. ISSN: 0018-9200. DOI: 10.1109/4.896230.

[EKC01b] Ralph Etienne-Cummings, Zaven Kevork Kalayjian, and Donghui Cai. "Programmable focal-plane MIMD image processor chip". In: *IEEE Journal of Solid-State Circuits* 36.1 (2001), pp. 64–73. ISSN: 00189200. DOI: 10.1109/4.896230.

[EWL13] J.F. Eusse, C. Williams, and R. Leupers. "CoEx: A novel profiling-based algorithm/architecture co-exploration for ASIP design". In: *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on.* July 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581520.

[EWL15] Juan Fernando Eusse, Christopher Williams, and Rainer Leupers. "CoEx: A Novel Profiling-Based Algorithm/Architecture Co-Exploration for ASIP Design". In: *ACM Trans. Reconfigurable Technol. Syst.* 8.3 (May 2015), 17:1–17:16. ISSN: 1936-7406. DOI: 10.1145/2629563. URL: http://doi.acm.org/10.1145/2629563.

[Eus+14] Juan Fernando Eusse et al. "A flexible ASIP architecture for connected components labeling in embedded vision applications". In: *Design, Automation {&} Test in Europe Conference {&} Exhibition (DATE), 2014* (2014), pp. 1–6. ISSN: 15301591. DOI: 10.7873/DATE.2014.367.

URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6800568.

[FH14] Max Ferger and Michael Hübner. "Instruction Set Optimization for Application Specific Processors". English. In: *Reconfigurable Computing: Architectures, Tools, and Applications*. Ed. by Diana Goehringer et al. Vol. 8405. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 268–274. ISBN: 978-3-319-05959-4. DOI: 10.1007/978-3-319-05960-0_28. URL: http://dx.doi.org/10.1007/978-3-319-05960-0_28.

[FZR08] Peter Foldesy, Akos Zarandy, and Csaba Rekeczky. "Configurable 3D-integrated focal-plane cellular sensor–processor array architecture". In: *International Journal of Circuit Theory and Applications* 36.5-6 (2008), pp. 573–588. ISSN: 1097-007X. DOI: 10.1002/cta.509. URL: http://dx.doi.org/10.1002/cta.509.

[FA94a] Robert Forchheimer and A Astrom. "Near-sensor image processing: a new paradigm". In: *Image Processing, IEEE Transactions on* 3.6 (1994), pp. 736–746.

[FA94b] Robert Forchheimer and A Astrom. "Near-sensor image processing: a new paradigm". In: *Image Processing, IEEE Transactions on* 3.6 (1994), pp. 736–746.

[For+14] Blair Fort et al. "Automating the Design of Processor/Accelerator Embedded Systems with LegUp High-Level Synthesis". In: *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*. IEEE. 2014, pp. 120–129.

[Fos89] Eric R Fossum. "Architectures For Focal Plane Image Processing". In: *Optical Engineering* 28.8 (Aug. 1989). ISSN: 0091-3286. DOI: 10.1117/12.7977048. URL: http://opticalengineering.spiedigitallibrary.org/article.aspx?doi=10.1117/12.7977048.

[Fos97] Eric R Fossum. “CMOS image sensors: electronic camera-on-a-chip”. In: *IEEE Transactions on Electron Devices* 44.10 (1997), pp. 1689–1698. ISSN: 00189383. DOI: 10.1109/16.628824.

[Fos98] Eric R Fossum. “Digital camera system on a chip”. In: *IEEE Micro* 18.3 (1998), pp. 8–15. ISSN: 02721732. DOI: 10.1109/40.683047.

[Fri15] Florian Fricke. “Exploring task-graph clustering for multi-core systems design”. Master in Electrical Engineering and Information Technology. MA thesis. Rurh-University Bochum, 2015.

[GR94] Daniel D Gajski and Loganath Ramachandran. “Introduction to high-level synthesis”. In: *Design & Test of Computers, IEEE* 11.4 (1994), pp. 44–54.

[Gaj+09] Daniel D. Gajski et al. *Embedded System Design : Modeling, Synthesis and Verification.* ISBN: 978-1-4419-0503-1. Springer Science + Business Media, 2009.

[GN00] Emden R. Gansner and Stephen C. North. “An Open Graph Visualization System and Its Applications to Software Engineering”. In: *Softw. Pract. Exper.* 30.11 (Sept. 2000), pp. 1203–1233. ISSN: 0038-0644. DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.3.CO;2-E. URL: http://dx.doi.org/10.1002/1097-024X(200009)30:11%3C1203::AID-SPE338%3E3.3.CO;2-E.

[Gen+10] Christos Gentsos et al. “Real-time canny edge detection parallel implementation for FPGAs”. In: *2010 IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2010 - Proceedings* (2010), pp. 499–502. DOI: 10.1109/ICECS.2010.5724558.

[Geo+13] Nivia George et al. “Making domain-specific hardware synthesis tools cost-efficient”. In: *Field-Programmable Technology (FPT), 2013 International Conference on.* IEEE. 2013, pp. 120–127.

[Geo+14] Nivia George et al. "Hardware system synthesis from domain-specific languages". In: *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE. 2014, pp. 1–8.

[Ger+09] Andreas Gerstlauer et al. "Electronic system-level synthesis methodologies". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.10 (2009), pp. 1517–1530. ISSN: 02780070. DOI: 10.1109/TCAD.2009.2026356.

[Goh+08] D. Gohringer et al. "Runtime adaptive multi-processor system-on-chip: RAMPSoC". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. Apr. 2008, pp. 1–7. DOI: 10.1109/IPDPS.2008.4536503.

[Gub+13] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. ISSN: 0167739X. DOI: 10.1016/j.future.2013.01.010. arXiv: 1207.0203. URL: http://dx.doi.org/10.1016/j.future.2013.01.010.

[GBC09] Radha Guha, Nader Bagherzadeh, and Pai Chou. "Resource management and task partitioning and scheduling on a run-time reconfigurable embedded system". In: *Computers & Electrical Engineering* 35.2 (2009), pp. 258–285.

[Gup+04] Sumit Gupta et al. "Using global code motions to improve the quality of results for high-level synthesis". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 23.2 (2004), pp. 302–312.

[Han+05] Frank Hannig et al. "Co-Design of Massively Parallel Embedded Processor Architectures". In: *Memory* (2005).

[H+14] Frank Hannig, Vahid Lari, et al. "Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Ap-

proach". In: *ACM Trans. Embed. Comput. Syst.* 13.4s (Apr. 2014), 133:1–133:29. ISSN: 1539-9087. DOI: 10.1145/2584660. URL: http://doi.acm.org/10.1145/2584660.

[HY08] Wenhao He and Kui Yuan. "An improved canny edge detector and its realization on FPGA". In: *Proceedings of the World Congress on Intelligent Control and Automation (WCICA)* (2008), pp. 6561–6564. DOI: 10.1109/WCICA.2008.4594570.

[Hes+16] Salma Hesham et al. "Survey on Real-Time Networks-on-Chip". In: *IEEE Transactions on Parallel and Distributed Systems* (2016), pp. 1–1. ISSN: 1045-9219. DOI: 10.1109/TPDS.2016.2623619. URL: http://ieeexplore.ieee.org/document/7728147/.

[HWA15a] J Hoozemans, S Wong, and Z Al-Ars. "Using VLIW softcore processors for image processing applications". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. 2015, pp. 315–318. DOI: 10.1109/SAMOS.2015.7363691.

[HWA15b] Joost Hoozemans, Stephan Wong, and Zaid Al-Ars. "Using VLIW Softcore Processors for Image Processing Applications". In: *Proceedings of the 15th International Conference on Systems, Architectures, Modeling and Simulation (SAMOS)*. 2015.

[htt] http://www.ace.nl/compiler/cosy.html. *Cosy compiler development system.*

[Hua+15] Qijing Huang et al. "The Effect of Compiler Optimizations on High-Level Synthesis-Generated Hardware". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 8.3 (2015), p. 14.

[Iss+12] Tsuyoshi Isshiki et al. "Application-specific Instruction-Set Processor design methodology for wireless image transmission systems". In: *SoC*

Design Conference (ISOCC), 2012 International. IEEE. 2012, pp. 293–296.

[Jan+14] B. Janßen et al. "Future Trends on Adaptive Processing Systems". In: *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*. Aug. 2014, pp. 166–173. DOI: 10.1109/ISPA.2014.30.

[Jer] Andrew Lumsdaine Jeremy Siek Lie-Quan Lee. *The Boost Graph Library - User Guide and Reference Manual*. ISBN 0-201-72914-8.

[Jor+17] Roel Jordans et al. "Automatic instruction-set architecture synthesis for VLIW processor cores in the ASAM project". In: *Microprocessors and Microsystems* 51 (2017), pp. 114–133. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2017.04.011. URL: http://www.sciencedirect.com/science/article/pii/S0141933116304318.

[JL11] Lech Jozwiak and Menno Lindwer. "Issues and challenges in development of massively-parallel heterogeneous MPSoCs based on adaptable ASIPs". In: *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*. IEEE. 2011, pp. 483–487.

[Kar+13] Madhushika ME Karunarathna et al. "Algorithm clustering for multi-algorithm processor design". In: *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE. 2013, pp. 451–454.

[KG06] Nasser Kehtarnavaz and Mark Gamadia. "Real-time image and video processing: from research to reality". In: *Synthesis Lectures on Image, Video & Multimedia Processing* 2.1 (2006), pp. 1–108.

[KT11] Joachim Keinert and Jürgen Teich. *Design of Image Processing Embedded Systems Using Multidimensional Data Flow*. New York, NY: Springer New York, 2011. ISBN: 978-1-4419-7181-4. DOI: 10.1007/978-

1-4419-7182-1. URL: http://link.springer.com/10.1007/978-1-4419-7182-1.

[KKK13] Yongmin Kim, Myeongsu Kang, and Jong Myon Kim. "Exploration of optimal many-core models for efficient image segmentation." In: *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 22.5 (2013), pp. 1767–1777. ISSN: 19410042. DOI: 10.1109/TIP.2012.2235851.

[K+10] Ralf Koenig, Lars Bauer, et al. "KAHRISMA: A Novel Hypermorphic Reconfigurable-instruction-set Multi-grained-array Architecture". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '10. Dresden, Germany: European Design and Automation Association, 2010, pp. 819–824. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1871127.

[KII97a] Takashi Komuro, I Ishii, and Masatoshi Ishikawa. "Vision chip architecture using general-purpose processing elements for 1 ms vision system". In: *Proceedings Fourth IEEE International Workshop on Computer Architecture for Machine Perception. CAMP'97*. IEEE Comput. Soc, 1997, pp. 276–279. ISBN: 0-8186-7987-5. DOI: 10.1109/CAMP.1997.632052. URL: http://ieeexplore.ieee.org/document/632052/.

[KII97b] Takashi Komuro, Idaku Ishii, and Masatoshi Ishikawa. "Vision chip architecture using general-purpose processing elements for 1 ms vision system". In: *Computer Architecture for Machine Perception, 1997. CAMP 97. Proceedings. 1997 Fourth IEEE International Workshop on.* IEEE. 1997, pp. 276–279.

[KOA05] S. Kyo, S. Okazaki, and T. Arai. "An integrated memory array processor architecture for embedded image recognition systems". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. June 2005, pp. 134–145. DOI: 10.1109/ISCA.2005.11.

[LA04a] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[LA04b] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE. 2004, pp. 75–86.

[Li+13a] Sheng Li et al. "The McPAT Framework for Multicore and Manycore Architectures: Simultaneously Modeling Power, Area, and Timing". In: *ACM Transactions on Architecture and Code Optimization* 10.1 (2013), pp. 1–29. ISSN: 15443566. DOI: 10.1145/2445572.2445577.

[Li+13b] Shuo Li et al. "System level synthesis of hardware for DSP applications using pre-characterized function implementations". In: *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE. 2013, pp. 1–10.

[LZL12] Xiaoyang Li, Wenbiao Zhou, and Dake Liu. "Application Source Codes Profiling for ASIP Memory Subsystem Design". In: *Procedia Engineering* 29 (2012), pp. 3160–3164.

[Lia+12] Hsuanchun Liao et al. "A Reconfigurable High Performance ASIP Engine for Image Signal Processing". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE. 2012, pp. 368–375.

[Lin+08] Zhiqiang Lin et al. "A CMOS image sensor for multi-level focal plane image decomposition". In: *Circuits and Systems I: Regular Papers, IEEE Transactions on* 55.9 (2008), pp. 2561–2572.

[LD06] a. Lopich and P. Dudek. "Architecture of a VLSI cellular processor array for synchronous/asynchronous image processing". In: *2006 IEEE*

International Symposium on Circuits and Systems (2006), pp. 3618–3621. ISSN: 02714310. DOI: 10.1109/ISCAS.2006.1693410.

[LD08] Alexey Lopich and Piotr Dudek. "ASPA: Focal plane digital processor array with asynchronous processing capabilities". In: *Proceedings - IEEE International Symposium on Circuits and Systems* (2008), pp. 1592–1595. ISSN: 02714310. DOI: 10.1109/ISCAS.2008.4541737.

[LD11] Alexey Lopich and Piotr Dudek. "ASPA: Asynchronous-Synchronous Focal-Plane Sensor-Processor Chip". English. In: *Focal-Plane Sensor-Processor Chips*. Ed. by Akos Zarandy. Springer New York, 2011, pp. 73–104. ISBN: 978-1-4419-6474-8. DOI: 10.1007/978-1-4419-6475-5_4. URL: http://dx.doi.org/10.1007/978-1-4419-6475-5_4.

[MHS10] Omer Malik, Ahmed Hemani, and Muhammad Ali Shami. "High level synthesis framework for a coarse grain reconfigurable architecture". In: *NORCHIP, 2010*. IEEE. 2010, pp. 1–6.

[Mia+08a] Wei Miao et al. "A programmable SIMD vision chip for real-time vision applications". In: *Solid-State Circuits, IEEE Journal of* 43.6 (2008), pp. 1470–1479.

[Mia+08b] Wei Miao et al. "A programmable SIMD vision chip for real-time vision applications". In: *IEEE Journal of Solid-State Circuits* 43.6 (2008), pp. 1470–1479. ISSN: 00189200. DOI: 10.1109/JSSC.2008.923621.

[Moi97] A Moini. "Vision chips or seeing silicon, 1997". In: *URL http://www.eleceng. adelaide. edu. au/Groups/GAAS/Bugeye/visionchips* 240 (1997).

[Mor+16a] J. Y. Mori et al. "A Rapid Prototyping Method to Reduce the Design Time in Commercial High-Level Synthesis Tools". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 253–258. DOI: 10.1109/IPDPSW.2016.56.

[MH16] Jones Y Mori and Michael Hubner. "Multi-level parallelism analysis and system-level simulation for many-core Vision processor design". In: *2016 5th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, June 2016, pp. 90–95. ISBN: 978-1-5090-2222-9. DOI: 10.1109/MECO.2016.7525710. URL: http://ieeexplore.ieee.org/document/7525710/.

[MK16] Jones Y Mori and Frederik Kautz. "Efficient camera input system and memory partition for a vision soft-processor". In: *Applied Reconfigurable Computing*. Ed. by Vanderlei Bonato, Christos Bouganis, and Marek Borgon. 9625th ed. Springer International Publishing, 2016, pp. 328–333. ISBN: 978-3-319-30481-6. DOI: 10.1007/978-3-319-30481-6_27. URL: http://link.springer.com/chapter/10.1007/978-3-319-30481-6%7B%5C%%7D7B%7B%5C%%7D25%7B%5C%%7D7D7B%7B%5C%%7D7B%7B%5C_%7D%7B%5C%%7D7D%7B%5C%%7D7B%7B%5C%%7D25%7B%5C%%7D7D7D27.

[Mor+16b] Jones Yudi Mori et al. "A Design Methodology for the Next Generation Real-Time Vision Processors". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9625. 2016, pp. 14–25. ISBN: 9783319304809. DOI: 10.1007/978-3-319-30481-6_2. URL: http://link.springer.com/10.1007/978-3-319-30481-6%7B%5C_%7D2.

[MKH15a] Jones Mori, Frederik Kautz, and Michael Huebner. "Design of an ISA-Extension for an Image Processing ASIP". In: *Proceedings of the German SNUG 2015*. 2015.

[MH14] J.Y. Mori and M. Huebner. "A high-level analysis of a multi-core vision processor using SystemC and TLM2.0". In: *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. Dec. 2014, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032491.

[MKH15b] J.Y. Mori, F. Kautz, and M. Huebner. “Design of and ISA-Extension for an Image Processing ASIP”. In: *Proceedings of the Germany SNUG 2015*. Ed. by Synopsys Users Group. 2015.

[MLB12] J.Y. Mori, C.H. Llanos, and P.A. Berger. “Kernel analysis for architecture design trade off in convolution-based image filtering”. In: *Integrated Circuits and Systems Design (SBCCI), 2012 25th Symposium on*. Aug. 2012, pp. 1–6. DOI: 10.1109/SBCCI.2012.6344453.

[MSL12] J.Y. Mori, C. Sanchez-Ferreira, and C.H. Llanos. “Real-Time Image Processing based on Neighborhood Operations using FPGA”. In: *Proceedings of the XVIII International IBERCHIP Workshop*. Feb. 2012. URL: http://www-elec.inaoep.mx/~IWS2012/ProceedingsIBERCHIP2012.pdf.

[Mus+10] Z. Musoromy et al. “Comparison of real-time DSP-based edge detection techniques for license plate detection”. In: *Information Assurance and Security (IAS), 2010 Sixth International Conference on*. Aug. 2010, pp. 323–328. DOI: 10.1109/ISIAS.2010.5604056.

[Na00] Yoshihiro Nakabo and et al. “1 ms column parallel vision system and its application of high speed target tracking”. In: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*. Vol. 1. IEEE. 2000, pp. 650–655.

[PS08] Alistair Palmer and Oliver Sinnen. “Scheduling Algorithm Based on Force Directed Clustering”. In: *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies* (2008), pp. 311–318. DOI: 10.1109/PDCAT.2008.89. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4710996.

[PDu11] P.Dudek. "Focal-Plane Sensor-Processor Chips". In: ed. by A. Zarandy. Springer, 2011. Chap. SCAMP-3: A Vision Chip with SIMD Current-mode Analogue Processor Array, pp. 17–43.

[Per09] Steve Perry. "Model based design needs high level synthesis: a collection of high level synthesis techniques to improve productivity and quality of results for model based electronic design". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association. 2009, pp. 1202–1207.

[Pha+13] Phi Hung Pham et al. "An on-chip network fabric supporting coarse-grained processor array". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.1 (2013), pp. 178–182. ISSN: 10638210. DOI: 10.1109/TVLSI.2011.2181546.

[PF13] Christian Pilato and Fabrizio Ferrandi. "Bambu: A modular framework for the high level synthesis of memory-intensive applications". In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE. 2013, pp. 1–4.

[Pos+14] Paulo Ricardo Possa et al. "A multi-resolution FPGA-based architecture for real-time edge and corner detection". In: *IEEE Transactions on Computers* 63.10 (2014), pp. 2376–2388. ISSN: 00189340. DOI: 10.1109/TC.2013.130.

[PMR14] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. "Fast and standalone design space exploration for high-level synthesis under resource constraints". In: *Journal of Systems Architecture* 60.1 (2014), pp. 79–93.

[Raj+10] Ragunathan Rajkumar et al. "Cyber-Physical Systems: The Next Computing Revolution Ragunathan". In: *Proceedings of the 47th Design Automation Conference on - DAC '10* (2010), p. 731. ISSN: 0738-100X.

DOI: 10.1145/1837274.1837461. URL: http://portal.acm.org/citation.cfm?doid=1837274.1837461.

[RG14] J. Rettkowski and D. Gohringer. "RAR-NoC: A reconfigurable and adaptive routable Network-on-Chip for FPGA-based multiprocessor systems". In: *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. Dec. 2014, pp. 1–6. DOI: 10.1109/ReConFig.2014.7032552.

[RAS11] Sandro Rigo, Rodolfo Azevedo, and Luiz Santos. *Electronic System Level Design: An Open-Source Approach*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 1402099398, 9781402099397.

[SFP13] Sergio Saponara, Luca Fanucci, and Esa Petri. "A multi-processor NoC-based architecture for real-time image/video enhancement". In: *Journal of Real-Time Image Processing* 8.1 (2013), pp. 111–125. ISSN: 18618200. DOI: 10.1007/s11554-011-0215-8.

[Sch07] Satu Elisa Schaeffer. "Graph clustering". In: *Computer Science Review* 1.1 (2007), pp. 27–64. ISSN: 1574-0137. DOI: https://doi.org/10.1016/j.cosrev.2007.05.001. URL: http://www.sciencedirect.com/science/article/pii/S1574013707000020.

[SML07] O. Schliebusch, H. Meyr, and R. Leupers. *Optimized ASIP Synthesis from Architecture Description Language Models*. Springer, 2007.

[Sch+16] Joseph A Schmitz et al. "A 1000 frames/s Vision Chip Using Scalable Pixel-Neighborhood-Level Parallel Processing". In: *IEEE Journal of Solid-State Circuits* (2016), pp. 1–13. ISSN: 00189200. DOI: 10.1109/JSSC.2016.2613094.

[SBH14] Muhammad Shafique, Lars Bauer, and Jörg Henkel. "Adaptive energy management for dynamically reconfigurable processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and*

Systems 33.1 (2014), pp. 50–63. ISSN: 02780070. DOI: 10.1109/TCAD.2013.2282265.

[SS15] Sharad Sinha and Thambipillai Srikanthan. “High-Level Synthesis: Boosting Designer Productivity and Reducing Time to Market”. In: *Potentials, IEEE* 34.4 (2015), pp. 31–35.

[Tay12] Michael B Taylor. “Is dark silicon useful?” In: *Proceedings of the 49th Annual Design Automation Conference on - DAC '12* (2012), p. 1131. ISSN: 0738-100X. DOI: 10.1145/2228360.2228567.

[tea17] OpenCV team. *OpenCV specification and API.* 2017. URL: https://opencv.org/.

[TTH13] J. Teich, A. Tanase, and F. Hannig. “Symbolic parallelization of loop programs for massively parallel processor arrays”. In: *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on.* June 2013, pp. 1–9. DOI: 10.1109/ASAP.2013.6567543.

[TG08a] J. Trajkovic and D.D. Gajski. “Custom Processor Core Construction from C Code”. In: *Application Specific Processors, 2008. SASP 2008. Symposium on.* June 2008, pp. 1–6. DOI: 10.1109/SASP.2008.4570778.

[TG08b] Jelena Trajkovic and DD Gajski. “Generation of custom co-processor structure from C-code”. In: *Center for Embedded Computer Systems* (2008).

[T+06] J. Trajkovic, M. Reshadi, et al. “A Graph Based Algorithm for Data Path Optimization in Custom Processors”. In: *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on.* 2006, pp. 496–503. DOI: 10.1109/DSD.2006.7.

[TV98] Emanuele Trucco and Alessandro Verri. *Introductory techniques for 3-D computer vision.* ISBN:0-13-261108-2. Prentice-Hall, Inc., 1998.

[Ung58] S.H. Unger. "A Computer Oriented toward Spatial Problems". In: *Proceedings of the IRE* 46.10 (Oct. 1958), pp. 1744–1750. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1958.286755.

[Vii+14] Timo Viitanen et al. "Heuristics for Greedy Transport Triggered Architecture Interconnect Exploration". In: *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (2014), 2:1–2:7. DOI: 10.1145/2656106.2656123. URL: http://doi.acm.org/10.1145/2656106.2656123.

[Vil+10] Jason Villarreal et al. "Designing modular hardware accelerators in C with ROCCC 2.0". In: *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on.* IEEE. 2010, pp. 127–134.

[WO00] Kazutoshi Wakabayashi and Takumi Okamoto. "C-based SoC design flow and EDA tools: An ASIC and system vendor perspective". In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 19.12 (2000), pp. 1507–1522.

[WD12] Bin Wang and Piotr Dudek. "Coarse grain mapping method for image processing on fine grain cellular processor arrays". In: *International Workshop on Cellular Nanoscale Networks and their Applications* 1 (2012). ISSN: 21650160. DOI: 10.1109/CNNA.2012.6331421.

[Wer15] Andre Werner. "Parameterizable SystemC/TLM2.0 simulator for multicore systems". Master in Electrical Engineering and Information Technology. MA thesis. Ruhr-University Bochum, 2015.

[Won+11] Stephan Wong et al. "Early results from ERA - Embedded Reconfigurable Architectures". In: *2011 9th IEEE International Conference on Industrial Informatics.* IEEE, July 2011, pp. 816–822. ISBN: 978-1-4577-0435-2. DOI: 10.1109/INDIN.2011.6034998. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C%%7D7B%7B%5C_%7D%

7B%5C%%7D7Dall.jsp?arnumber=6034998%20http://ieeexplore.ieee.org/document/6034998/.

[Wu13] Bo Wu. *Design of Embedded Vision Processors*. Tech. rep. Synopsys White Paper, 2013.

[YLH17] Jones Yudi, Carlos Humberto Llanos, and Michael Huebner. "System-level design space identification for Many-Core Vision Processors". In: *Microprocessors and Microsystems* 52.Supplement C (2017), pp. 2–22. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2017.05.013. URL: http://www.sciencedirect.com/science/article/pii/S0141933117300571.

[Zar+11] Akos Zarandy et al. "VISCUBE: A Multi-Layer Vision Chip". English. In: *Focal-Plane Sensor-Processor Chips*. Springer New York, 2011, pp. 181–208. ISBN: 978-1-4419-6474-8. DOI: 10.1007/978-1-4419-6475-5_8. URL: http://dx.doi.org/10.1007/978-1-4419-6475-5_8.

[Zar11] Ákos Zarándy. *Focal-plane sensor-processor chips*. Springer Science & Business Media, 2011.

[ZFW11] Wancheng Zhang, Qiuyu Fu, and Nan-Jian Wu. "A Programmable Vision Chip Based on Multiple Levels of Parallel Processors". In: *Solid-State Circuits, IEEE Journal of* 46.9 (Sept. 2011), pp. 2132–2147. ISSN: 0018-9200. DOI: 10.1109/JSSC.2011.2158024.

[Zuo+13] Wei Zuo et al. "Improving high level synthesis optimization opportunity through polyhedral transformations". In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2013, pp. 9–18.

[Zuo+14] Wei Zuo et al. "New solutions for system-level and high-level synthesis". In: *Integrated Circuits (ISIC), 2014 14th International Symposium on*. IEEE. 2014, pp. 71–74.

List of Figures

1.1 Top: Common sensor system with remote processing; Bottom: Smart Sensor system with local processing. 4

1.2 Conceptual model of a Multiple-Input/Multiple-Output (MIMO) heterogeneous processing architecture [Jan+14]. 5

1.3 A Smart Camera with a complete multiple face recognition embedded application. 7

1.4 Complementary Metal-Oxide-Semiconductor (CMOS) pixel array acquisition types: single and multiple pixel streams [Mor+16b], and the design trade-off related. 9

1.5 Fill-factor reduction when adding more functionalities to the image sensor [Mor+16b]. 11

1.6 Technological trends in the design of Image Processing and Computer Vision (IP/CV) systems: Parallelisation and Integration. 12

1.7 *Focal-Plane/Near-Sensor* Image Processing concept [Fos98] [MH14] [MKH15a]. 13

2.1 A Y-chart showing the levels of an electronics design, with the parts covered in this work highlighted [Gaj+09]. 16

3.1 Architecture of the SRVC (SIMD Real-time Vision Chip) [Mia+08a]. 26

3.2 The SIMD multilayer chip from [FZR08]. 27

3.3 Type of operations [ZFW11]. 28

3.4 Schematic of the [ZFW11] chip architecture: image sensor array, A/D converters, PE array, RP array, and MPU. 29
3.5 The ASPA chip architecture [LD11]. 30
3.6 The SCAMP3 chip architecture [PDu11]. 30
3.7 The Neighborhood Processor Array [Sch+16]. 31
3.8 The WiMedia MAC MpSoC architecture [Iss+12]. 38
3.9 Block diagram of the IMAP-CE MPSoC and PE [KOA05]. 39
3.10 An example of a Weakly-Programmable Processor Array (WPPA) with parameterizable Processing Elements (PEs) [Han+05]. 40
3.11 On the left, the Tightly-Coupled Processor Array (TCPA) architecture; on the right, an heterogeneous Multi-Processor System-on-Chip (MPSoC) example [H+14]. 41
3.12 On the left, the RAR-NoC MPSoC; on the right, the configurable router architecture [RG14]. 42
3.13 On the left, the ME tile diagram; on the right, the NoC-based MPSoC architecture [SFP13]. 43
3.14 The NoC-based MPSoC proposed by [Pha+13]. 44

4.1 A general IP/CV processing chain [KG06; MH14; MK16; YLH17]. . . 48
4.2 Basic pixel-level IP/CV operations [KG06]. 50
4.3 OpenVX graph of a simple application [Edi17],[Mor+16b] 51

5.1 Throughput considering: $M = N = 1024$, $W = Z = \{variable\}$. . . . 55
5.2 Initial Delay considering: $M = N = 1024$, $L = K = \{3, 7, 15\}$, $W = Z = \{variable\}$. 56
5.3 Total Delay considering: $M = N = 1024$, $L = K = \{3, 7, 15\}$, $W = Z = \{variable\}$. 57
5.4 *Focal-Plane/Near-Sensor* Image Processing concept [Fos98] [MH14] [MKH15a]. 58
5.5 Simple Edge Detection application. 59

5.6 Macro-Pipeline: exploring the parallelism in image sequences [MH16]. 60
5.7 Operation level Parallelism: 3×3 Convolution example [MH16]. . . . 61
5.8 Loop-Tiling example for a generic IP/CV block, as defined in [MKH15a]. 61
5.9 Exploring the Spatial Parallelism using a *Region-based* configuration: each PE is responsible for a region of the image [MH16]. 62
5.10 General Tiled Architecture 63

6.1 An example of task-graph mapping and scheduling. 68
6.2 The High-Level methodology [MKH15a]. 70

7.1 Processor absolute numbering 76
7.2 Processor relative numbering 77
7.3 BGL Konzepte 78
7.4 AST Literale 79
7.5 Task-Graph Literale 79
7.6 Abstract Syntax Tree (AST) for function parameters access [Fri15]. . 80
7.7 AST for variable access [Fri15]. 80
7.8 Task-Graph (TG) for function parameters access [Fri15]. 81
7.9 TG for variable access [Fri15]. 81
7.10 AST Unary operators 82
7.11 Task-Graph Unary Operators 83
7.12 AST binary operators 84
7.13 TG binary operators 85
7.14 AST Conditional expression 86
7.15 Task-Graph Conditional expression 87
7.16 AST function call 88
7.17 Task graph function call 88
7.18 AST if branching 91
7.19 Task-Graph if branching 92

8.1 TG example 96

8.2 Springs 97
8.3 ClusterPE 99
8.4 3×3 convolution example 99
8.5 Basic pixel assignment for the sample architecture. 100
8.6 communication structure convolution manual 101
8.7 Communication structure convolution automatic 103
8.8 Manual Clustering [Fri15]. 105
8.9 Automatic Clustering [Fri15]. 105
8.10 Clustering Rank-Order-Filter 106

9.1 From the clustered task-graph to a many-core model 110
9.2 Example for a *task graph* with clustering [Wer15]. 111
9.3 From the Task-Graph to a many-core model [Wer15]. 113
9.6 The tiles and the communication pattern after clustering [Wer15]. . . 115
9.4 *Task Graph* for the example [Wer15].. 117
9.5 *Task Graph* after the clustering 118

12.1 The simulation tool's organization. 126
12.2 Example of how the simulator implements different PE types. 127
12.3 Example of pixel exchange needs among different image regions [MH14].128
12.4 Private vs. Shared Pixel Memory for external accesses. 129
12.5 4-Connected and 8-Connected topologies simulated. 131
12.6 Programming Model overview. 132
12.7 Integration of our simulator with the McPAT Framework (adapted from [Li+13a]). 134

13.1 Estimated area for all configurations, considering $W = [4, 8, 16, 32, 64]$ pixels. 139
13.2 Estimated power for all configurations, considering $W = [4, 8, 16, 32, 64]$ pixels. 140

13.3 Estimated cycles for all configurations, considering $W = [4, 8, 16, 32, 64]$ pixels. 141

14.1 Many-Core Vision Processor and its Tile structure. 145
14.2 *Pixel_Memory* synthesis for different amounts of pixels per Tile (8 bits per pixel). 146
14.3 *Processing_Element* internal organization. 147
14.4 *Router*'s internal structure. 148
14.5 The Canny Edge Detector method: building blocks and operation's types. 150
14.6 Performance of Neighborhood Operations for different number of pixels per Tile. 151
14.7 Performance of Segmentation Operations for different number of pixels per Tile. 152
14.8 Performance of the CED application for different number of pixels per Tile. 153

15.1 Typical High-Level Synthesis development flow. 157
15.2 Flow of the proposed method: (a) in Green (continuous line), the straightforward flow, with few user interaction; (b) in Red (dotted line), the flow to create new Node models (same as the Common Flow from Fig.15.1). 158
15.3 Task-Graph for a simple example. 159
15.4 Left: Partial Task-Graph of a complex IF-clause, highlighting the expression; Right: Sample Task-Graph of a For-Loop. 161
15.5 The Combinatorial Optimization Problem solved to select the best design alternative under the Design Constraints. 163
15.6 Use-cases selected: Convolution and Fibonacci. 164
15.7 Comparison among the Design Constraints, a Direct Implementation in Vivado HLS and our method. 166

15.8 Tile architecture for the HLS approach. 167
15.9 Processing chain of the Sobel edge enhancement algorithm. 169
15.10The ASIP design methodology followed in this work (adapted from [MKH15a]). 170
15.11PD-RISC processor architecture (adapted from [Wu13]). 171
15.12Neighborhood Loader architecture. 173
15.13Assembly code reduction achieved by creating new instructions. . . . 173
15.14PWA architecture: with the Neighborhood Loader and input-MAC unit. 174
15.15Slice LUTs used by each architecture. 174
15.16Maximum frequency achieved by each architecture. 176
15.17Frame rate achieved by each architecture. 177
15.18Power consumption estimated for each architecture. 177

List of Tables

1.1 Summary of the main Design Space Exploration (DSE) parameters explored in this work. 14

3.1 Summary of most relevant related Vision Processors. 31

3.2 ASIP design methodologies/tools from the literature review. 33

3.3 Comparison of methods and tools for High-Level Synthesis (HLS) . . 37

3.4 My caption 45

5.1 Throughput for each acquisition scheme. 54

5.2 Initial Delay for each acquisition scheme. 55

7.1 Control flow constructions. 89

9.1 Clustering parameters used in the example. 115

9.2 Simulationszeit für die Berechnung der Faltung 116

13.1 Overview of configurations in [Sch+16] and in this work. 137

13.2 Simulated architectural configurations. 138

14.1 Instruction Set used in the PEs (Reduced Instruction Set Computer (RISC) processors). 147

14.2 Router synthesis results: 256 × 256 resolution, 5 steps, 1 frame, 12 messages. 149

14.3 Comparison with related architectures, all implementing the CED application. 153

15.1 Alternative Hardware Nodes on the Xilinx Xc7vx690tffg1761-2 FPGA device 165
15.2 Results of direct implementations and of our method on the Xilinx Xc7vx690tffg1761-2 FPGA device 166
15.3 System's comparison 178

Acronyms

ADC Analog-to-Digital Converter

ADL Architecture Description Language

AI Artificial Intelligence

APS Active Pixel Sensor

ASIP Application-Specific Instruction set Processor

AST Abstract Syntax Tree

AT Approximately-Timed

BGL Boost Graph Library

CCD Charge-Coupled Devices

CMOS Complementary Metal-Oxide-Semiconductor

CMOS-APS CMOS and Active Pixel Sensor (APS)

CMP Chip Multi-Processor

CPS Cyber-Physical System

DSE Design Space Exploration

DSL Domain-Specific Language

FDC Force-Directed Clustering

FIFO First-In First-Out

FIMP Function Implementation

FPGA Field-Programmable Gate Array

FPIP Focal-Plane Image Processing

FSM Finite State Machine

GPP General-Purpose Processor

HLS High-Level Synthesis

ILP Instruction-Level Parallelism

IoT Internet of Things

IP/CV Image Processing and Computer Vision

IR Intermediate Representation

KPN Kahn Process Network

LT Loosely-Timed

MIMO Multiple-Input/Multiple-Output

MPSoC Multi-Processor System-on-Chip

NoC Network-on-Chip

NSIP Near-Sensor Image Processing

P2P Point-to-Point

PE Processing Element

PU Processing Unit

PvC Pervasive Computing

QoS Quality of Service

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SER Specify, Explore-and-Refine

SIMD Single-Instruction Multiple-Data

SoC System-on-Chip

TCPA Tightly-Coupled Processor Array

TG Task-Graph

TSV Through-Silicon Vias

TTA Transport-Triggered Architecture

UbiC Ubiquitous Computing

VHDL Very-High Scale of Integration Chip Hardware Description Language

VLIW Very-Large Instruction Word

VLSI Very-Large Scale of Integration

WPPA Weakly-Programmable Processor Array

WPPE Weakly-Programmable Processing Element

About the Author

Working experience

- **2014-2018** Research Assistant, Chair of Embedded Systems for Information Technology, Ruhr-University Bochum (www.esit.rub.de). PhD scholarship granted by the CAPES Foundation, Brazilian Ministry of Education, under the Science without Borders Program (www.cienciasemfronteiras.gov.br).
- **2010-current** Auxiliar Professor, Department of Mechanical Engineering, University of Brasilia (www.unb.br / www.enm.unb.br). On a working leave from 2014-2018 to pursue the PhD at the Ruhr-University Bochum.
- **2010-2010** Digital IC Design Internship, Information Technology Research Centre (www.cti.gov.br), Brazilian Ministry of Science and Technology. Technology Development sponsorship granted by the IC Brazil Program (www.ci-brasil.gov.br).
- **2009-2009** Science and Technology Analyst, CAPES Foundation, Brazilian Ministry of Education (www.capes.gov.br).
- **2008-2009** Lecturer, Higher Education Institute of Brasilia (www.iesb.br).
- **2008-2011** Product Development Engineer, SEIT (Solutions for Engineering and Technological Innovation).

Academic Formation

- **2007-2010** Master Student, Mechatronics System Graduate Program, University of Brasilia (www.ppmec.unb.br).

Master Thesis: *Implementação de técnicas de processamento de imagens no domínio espacial em sistemas reconfiguráveis / Implementation of spatial-domain image processing techniques in reconfigurable systems.* - available only in Portuguese (http://repositorio.unb.br/handle/10482/7237).

- **2001-2007** Bachelor Student, Mechatronics Engineering, University of Brasilia (www.mecatronica.unb.br).

  Bachelor Thesis: *Desenvolvimento de um algoritmo baseado em processamento de imagens para medição de rugosidade / Design of an image processing algorithm for surface finishing measurement* .

Publications

16.1 Publications within the Ruhr-University Bochum

Journal Papers

- **2017 Jones Yudi**, Carlos Humberto Llanos, Michael Huebner, System-level design space identification for Many-Core Vision Processors, In Microprocessors and Microsystems, Volume 52, 2017, Pages 2-22, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2017.05.013.
- **Accepted on December 2017** Muhammed Al Kadi, Benedikt Janssen, **Jones Yudi**, Michael Huebner, General Purpose Computing with Soft GPUs on FPGAs, In ACM Transactions on Reconfigurable Technology and Systems (TRETS), ISSN 1936-7406.

Book Chapters

- **2015** Schwiegelshohn, F.; Wehner, P.; **Mori, J.Y.**; Janssen, B.l; Navarro, O.; Rettkowski, J.; Al Kadi, M.; Goehringer, D.; Huebner, M. - Reconfigurable Processors and Multicore Architectures. In: Pierre-Emmanuel Gaillardon. (Org.). Reconfigurable Logic: Architecture, Tools, and Applications. 1ed.: CRC Press, 2015, v. , p. 259-292.

Conference Papers

- **2017** Navarro O., **Mori J.**, Hoffmann J., Stuckmann F., Hübner M. (2017) A Machine Learning Methodology for Cache Recommendation. In: Wong

S., Beck A., Bertels K., Carro L. (eds) Applied Reconfigurable Computing. ARC 2017. Lecture Notes in Computer Science, vol 10216. Springer, Cham, https://doi.org/10.1007/978-3-319-56258-2_27.

- **2016 J. Y. Mori**, A. Werner, F. Fricke and M. Hüebner, "A Rapid Prototyping Method to Reduce the Design Time in Commercial High-Level Synthesis Tools," 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, 2016, pp. 253-258. doi: 10.1109/IPDPSW.2016.56

- **2016 Mori J.Y.**, Kautz F., Hübner M. (2016) Efficient Camera Input System and Memory Partition for a Vision Soft-Processor. In: Bonato V., Bouganis C., Gorgon M. (eds) Applied Reconfigurable Computing. ARC 2016. Lecture Notes in Computer Science, vol 9625. Springer, Cham. https://doi.org/10.1007/978-3-319-30481-6_27.

- **2016 Mori J.Y.**, Werner A., Shallufa A., Fricke F., Hübner M. (2016) A Design Methodology for the Next Generation Real-Time Vision Processors. In: Bonato V., Bouganis C., Gorgon M. (eds) Applied Reconfigurable Computing. ARC 2016. Lecture Notes in Computer Science, vol 9625. Springer, Cham. https://doi.org/10.1007/978-3-319-30481-6_2

- **2016 J. Y. Mori** and M. Hübner, "Multi-level parallelism analysis and system-level simulation for many-core Vision processor design," 2016 5th Mediterranean Conference on Embedded Computing (MECO), Bar, 2016, pp. 90-95. doi: 10.1109/MECO.2016.7525710

- **2015 J. Y. Mori**, C. H. Llanos and M. Hüebner, "A Framework to the Design and Programming of Many-Core Focal-Plane Vision Processors," 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing, Porto, 2015, pp. 193-198. doi: 10.1109/EUC.2015.24

- **2014 J. Y. Mori** and M. Huebner, "A high-level analysis of a multi-core

vision processor using SystemC and TLM2.0," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Cancun, 2014, pp. 1-6. doi: 10.1109/ReConFig.2014.7032491

- **2014** B. Janßen, **J. Y. Mori**, O. Navarro, D. Göhringer and M. Hübner, "Future Trends on Adaptive Processing Systems," 2014 IEEE International Symposium on Parallel and Distributed Processing with Applications, Milan, 2014, pp. 166-173. doi: 10.1109/ISPA.2014.30

Other Publications

- **2015 Mori J.Y.**, Kautz F., Huebener M.; Design of an ISA-Extension for an Image Processing ASIP. In: Synopsys User Group Conference, Munich, Proceedings of the SNUG Germany 2015. https://www.synopsys.com/community/snug/snug-germany/location-proceedings-2015.html

16.2 Publications within the University of Brasilia

Journal Papers

- **2016** Sánchez-Ferreira, C.; **Mori, J.Y.**; Farias, M.C.Q.; Llanos, C.H. - A real-time stereo vision system for distance measurement and underwater image restoration - Journal of the Brazilian Society of Mechanical Sciences and Engineering (2016) 38: 2039. https://doi.org/10.1007/s40430-016-0596-5.

- **2012 Jones Y. Mori**, Janier Arias-Garcia, Camilo Sánchez-Ferreira, Daniel M. Muñoz, Carlos H. Llanos, and J. M. S. T. Motta, "An FPGA-Based Omnidirectional Vision Sensor for Motion Detection on Mobile Robots," International Journal of Reconfigurable Computing, vol. 2012, Article ID 148190, 16 pages, 2012. doi:10.1155/2012/148190

Book Chapters

- **2012 Mori, J.Y.**; Llanos, Carlos H. . Accelerating stereo vision processing using reconfigurable hardware on embedded systems. In: Sadek C. A. Alfaro; Jose Mauricio S. T. Motta; Victor J. De Negri. (Org.). Symposium Series in Mechatronics. 1ed.Rio de Janeiro: ABCM - Brazilian Society of Mechanical Sciences and Engineering, 2012, v. 5, p. 1-8. abcm.org.br/symposium-series/SSM_Vol5/Section_I_Computer_Vision/06251.pdf

Conference Papers

- **2013** C. Sánchez-Ferreira, **J. Y. Mori**, C. H. Llanos and E. Fortaleza, "Development of a stereo vision measurement architecture for an underwater robot," 2013 IEEE 4th Latin American Symposium on Circuits and Systems (LASCAS), Cusco, 2013, pp. 1-4. doi: 10.1109/LASCAS.2013.6519001

- **2012 J. Y. Mori**, C. H. Llanos and P. A. Berger, "Kernel analysis for architecture design trade off in convolution-based image filtering," 2012 25th Symposium on Integrated Circuits and Systems Design (SBCCI), Brasilia, 2012, pp. 1-6. doi: 10.1109/SBCCI.2012.6344453

- **2012** C. Sánchez-Ferreira, **J. Y. Mori** and C. H. Llanos, "Background subtraction algorithm for moving object detection in FPGA," 2012 VIII Southern Conference on Programmable Logic, Bento Goncalves, 2012, pp. 1-6. doi: 10.1109/SPL.2012.6211792

- **2011 Jones Yudi Mori**, Daniel Muñoz Arboleda, Janier Arias Garcia, Carlos Llanos Quintero, and José Motta. 2011. FPGA-based image processing for omnidirectional vision on mobile robots. In Proceedings of the 24th symposium on Integrated circuits and systems design (SBCCI '11). ACM, New York, NY, USA, 113-118. DOI=http://dx.doi.org/10.1145/2020876.2020904

- **2011** J. Y. Mori, C. Sánchez-Ferreira, D. M. Muñoz, C. H. Llanos and P. Berger, "An unified approach for convolution-based image filtering on recon-

figurable systems," 2011 VII Southern Conference on Programmable Logic (SPL), Cordoba, 2011, pp. 63-68. doi: 10.1109/SPL.2011.5782626

Theses supervised

Master Theses

1. **Carsten Grautstück**. *High-Level Synthesis of ArchC processor models*. Electrical Engineering and Information Technology. Ruhr-University Bochum. 2016.

2. **Andre Werner**. *Parameterizable SystemC/TLM2.0 simulator for multi-core systems*. Electrical Engineering and Information Technology. Ruhr-University Bochum. 2015.

3. **Florian Fricke**. *Exploring task-graph clustering for multi-core systems design*. Electrical Engineering and Information Technology. Ruhr-University Bochum. 2015.

Bachelor Theses

1. **Fabian Stuckmann**. *Hardware acceleration of convolutional neural networks training*. Electrical Engineering and Information Technology. Ruhr-University Bochum. 2017.

2. **Frederik Kautz**. *Design of a Processor Architecture for Computer Vision Tasks*. Electrical Engineering and Information Technology. Ruhr-University Bochum. 2015.

3. **Arij Shallufa**. *Design and simulation of parallel image processing systems in a virtual prototyping tool*. Applied Informatics. Ruhr-University Bochum. 2015.

4. **Yannick Bihege**. *Analysis, debugging and profiling of a RISC processor for computer vision algorithms.* Applied Informatics. Ruhr-Universiyt Bochum. 2015.

Undergraduate Projects

1. **Kevin Winge**. *Free-View Point generator using OpenGL.* Electrical Engineering and Information Technology. Ruhr-University Bochum. 2016.

2. **Henry Bathke**. *3D Face Tracking mechanisms for Free-View Point visualization.* Electrical Engineering and Information Technology. Ruhr-University Bochum. 2016.