FPGA-based Programmable Embedded Platform for Image Processing Applications

Fahad Manzoor Siddiqui

School of Electronics, Electrical Engineering and Computer Science

Queen's University Belfast

A thesis submitted for the degree of

Doctor of Philosophy

September 11, 2018

Abstract

A vast majority of electronic systems including medical, surveillance and critical infrastructure employs image processing to provide intelligent analysis. They use onboard pre-processing to reduce data bandwidth and memory requirements before sending information to the central system. Field Programmable Gate Arrays (FPGAs) represent a strong platform as they permit reconfigurability and pipelining for streaming applications. However, rapid advances and changes in these application use cases crave adaptable hardware architectures that can process dynamic data workloads and be easily programmed to achieve efficient solutions in terms of area, time and power.

FPGA-based development needs iterative design cycles, hardware synthesis and place-and-route times which are alien to the software developers. This work proposes an FPGA-based programmable hardware acceleration approach to reduce design effort and time. This allows developers to use FPGAs to profile, optimise and quickly prototype algorithms using a more familiar software-centric, edit-compile-run design flow that enables the programming of the platform by software rather than high-level synthesis (HLS) engineering principles.

Central to the work has been the development of an optimised FPGA-based processor called Image Processing Processor (IPPro) which efficiently uses the underlying resources and presents a programmable environment to the programmer using a dataflow design principle. This gives superior performance when compared to competing alternatives. From this, a three-layered platform has been created which enables the realisation of parallel computing skeletons on FPGA which are used to efficiently express designs in high-level programming languages. From bottom-up, these layers represent programming (actor, multiple actors and parallel skeletons) and hardware (IPPro core, multicore IPPro, system infrastructure) abstraction. The platform allows acceleration of parallel and non-parallel dataflow applications.

A set of point and area image pre-processing functions are implemented on Avnet Zedboard platform which allows the evaluation of the performance. The point function achieved 2.53 times better performance than the area functions and point and area functions achieved performance improvements of 7.80 and 5.27 times over single core IPPro by exploiting data parallelism. The pipelined execution of multiple stages revealed that a dataflow graph can be decomposed into balanced actors to deliver maximum performance by hiding data transfer and processing time through exploiting task parallelism; otherwise, the maximum achievable performance is limited by the slowest actor due to the ripple effect caused by unbalanced actors. The platform delivered better performance in terms of fps/Watt/Area than Embedded Graphic Processing Unit (GPU) considering both technologies allows a software-centric design flow.

Acknowledgements

I would like to express my profound gratitude to my supervisor, Prof. Roger Woods for giving me the opportunity to undertake part-time research and in providing me continuous advice, supervision and encouragement throughout my research. I am grateful for his systematic guidance, comprehensive reviews and critical feedback to improve this thesis. In addition, I am grateful to Dr. Karen Rafferty for providing additional support, reasoning and constructive criticism for my research. I want to thank Prof. Sakir Sezer for supporting me in finishing the thesis.

I would like to thank my colleagues at Queen's University Belfast with whom I worked during my PhD including Dr. Burak Bardak and Dr. Moslem Amiri with whom I worked on the Rathlin project, for sharing their ideas and knowledge to improve my research activities. Particular thanks go to Dr. Matthew Milford, Dr. Colm Kelly, Umar Ibrahim Minhas and Tiantai Deng for sharing application use case results to optimise and improve the platform architecture.

A warm thanks to Margarita Magdenko for her continuous moral support, encouragement and affection. Above all, I would like to thank my parents who have given me the strength and wisdom to be sincere in my work, for setting high moral standards, supporting me through their hard work and their unconditional love and affection.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Image Processing has been a field of academic research over the past several decades and is extensively employed to interpret meaning from images or video. A vast majority of electronic systems from automotive industry to factory automation, medical and surveillance employs *image processing* to provide intelligent analysis of their systems and improve productivity. The processing demands of such workloads often surpass the capacity of traditional computing architectures.

Video analytics is the branch of *embedded vision* that analyses human activity and extracts information from video content that is meaningful as perceived by the human eye. It is gaining traction in a diverse set of application markets including retail, transportation, consumer, smart-cities, critical infrastructure, and enterprise, among others. These systems use smart cameras with on-board image pre-processing to process data and give a reduction in data bandwidth and memory requirements before sending it to the centralised, server-based software platforms [1], [2]. These platforms are being aided by advanced algorithms to interpret and analyse meaning of an ever-growing increase of video content.

There is a significant amount of investment in industrial and educational research, which is expected to grow in coming years considerably. The *Embedded Vision Alliance* has estimated that the revenue from analytic video hardware, software and services will increase from $858 million to nearly $3 billion by 2022, representing a compound annual growth rate (CAGR) of 19.6% [3]. This growth brings significant challenges to explore new parallel computing architectures in general and image processing architecture in particular, which are portable, efficient and easier to use for a wide range of application developers.

1.1 Research problem

The increasing demands for computation and bandwidth of existing and next-generation image processing applications pose severe challenges to both hardware and software solutions. While special purpose, hardware such as the *Graphics processing unit* (GPU) can handle the increasing computational demands of these data intensive applications, they come at the expense of higher power consumption, longer design times and significant programming effort. However, rapid advances and changes in state-of-art technology for these applications quickly make obsolete a dedicated accelerator or chip. The obsolescence is especially true in case of *Application-specific integrated circuit* (ASIC).

Field-programmable gate array (FPGA) technology has evolved significantly over the years from simple regular arrangements of *configurable logic* blocks and routing to a heterogeneous *system-on-chip* (SoC). Much of this improvement has inevitably been driven by market segments where FPGAs are particularly prevalent in signal processing due to pipelining and parallelism that they offer. While the technology gap between ASIC and FPGA is widening, most of new ASIC designs lag behind due to overall design effort, time and cost making FPGA more attractive. FPGAs are proven computing platforms that offer reconfigurability, concurrency and pipelining. GPUs seem a viable highly programmable platform but, current energy requirements and limitations of *Dennard scaling* have acted to limit clock scaling, thus limits processing capabilities [4].

Apart from FPGA being a high performance and power efficient computing technology, they have not been accepted as a mainstream computing platform. The primary inhibitor is the need to use specialist programming tools, describing algorithms in *hardware description language* (HDL) and lack of adaptability. Silicon vendors started to alleviate this issue by introducing high-level programming tools such as Xilinx's Vivado High-level Synthesis (HLS) and Intel's (Altera's) compiler for OpenCL. While the level of abstraction has been raised, a gap still exists between adaptability, performance and efficient utilisation of FPGA resources. Nevertheless, the FPGA design flow still requires design *synthesis* and *place-and-route* that can be time-consuming depending on the complexity and size of the design; this is alien to software and algorithm developers. The development of algorithms is usually an experimental process and may require many design iterations involving quick profiling, design exploration and prototyping. In such circumstances, an FPGA design flow that requires synthesis, place-and-route process is not comparable to a more familiar software-centric design flow that uses *edit-compile-run.* Therefore, an iterative development of a different application on FPGAs is a complicated and time-consuming process which inhibits widespread use of the technology.

The changing technology landscape and fast evolution of new application use-cases make it imperative that underlying hardware architecture should be *adaptable*. Such platforms are a significant part of some major research initiatives where both quick prototyping and reduced design time are of prime importance. Moreover, the computing platform should allow design exploration possibilities including decomposition and mapping to optimise applications.

1.2 Rathlin Project

Rathlin research project had undertaken to approach these research problems [5]. The scope of this project was to investigate the rapid developments in image acquisition/interpretation and intelligent algorithms. As they have not been matched by sound software engineering principles, to generate efficient solutions for time, memory and power efficient hardware.

A domain-specific image processing language *Rathlin Image Processing Language* (RIPL) for FPGAs was introduced [6]. RIPL supports algorithmic skeletons to express image processing components, which functionally inherit a dataflow *model of computation*. A RIPL description is converted into an intermediate dataflow language (CAL) which is mapped on to the FPGA as a network of stream processing units [7]. Though, one of the project objectives was to facilitate iterative development of different applications by replacing FPGA design flow to software-centric flow. Therefore, an adaptable *FPGA-based hardware acceleration platform* architecture was developed that efficiently maps and executes parallel CAL dataflow descriptions. This platform aimed to unleash the potential of state-of-art FPGAs in close synergy with a suitable software representation. Further discussion on *Rathlin* programming workflow and relevant background work will be discussed in Chapter 3.

1.3 Proposed approach

FPGA heterogeneous system-on-chip (SoC) architectures have addressed some of the hardware and software programming challenges [8], [9], [10]. However, fitting different parallel computational tasks onto the underlying FPGA hardware resources by using more processing nodes integrated into a single-chip is important. Besides, the need for architecture specific skills to port and optimise the applications to the underlying FPGA hardware resources which includes managing and exploiting parallelism and system heterogeneity, is also challenging. This problem is directly related to the optimal exploration of type and degree of parallelism among multiple processing nodes available within a heterogeneous system. Realising parallel applications on these heterogeneous platforms often involves design and development of the processing nodes or hardware accelerators. They can comprise fixed, reconfigurable or software programmable processors or combinations thereof. The adaptability of the underlying platform depends on the flexibility and programmability of its processing nodes. This adaptability can be present in the device, in the circuit, in the micro-architecture, in the system or even in the runtime software layer or among all of these.

This research work proposes an *FPGA-based programmable hardware acceleration platform*. It is a system architecture that takes advantage of heterogeneous computing. The FPGA glue logic can be used as a *programmable hardware acceleration architecture* that substitutes the traditional FPGA design flow (synthesis and place-and-route) to a software-centric *edit-compile-run* design flow [11].

FPGA-based soft-core processor architectures have been used [12], [13], [14], [15], [16] as they offer better software controlled functionalities, system flexibility/portability, and partitioning of hardware-software co-design over other approaches [17]. The *programmable hardware acceleration architecture* is a three-layer architecture as illustrated in Figure 1.1 and outlined below:

- The bottom layer is comprised of a novel FPGA-based soft-core *Image Processing Processor* **(IPPro)** architecture tailored to accelerate image pre-processing applications. It supports both shared memory and message passing data processing models. The IPPro core is an independent, self-managed, programmable hardware accelerator that handles the exchange of data among multiple producers and consumers by executing stream instructions. It is used as a basic computational unit of the proposed platform as shown in Figure 1.1.

- The middle layer is composed of multiple IPPro cores connected with an

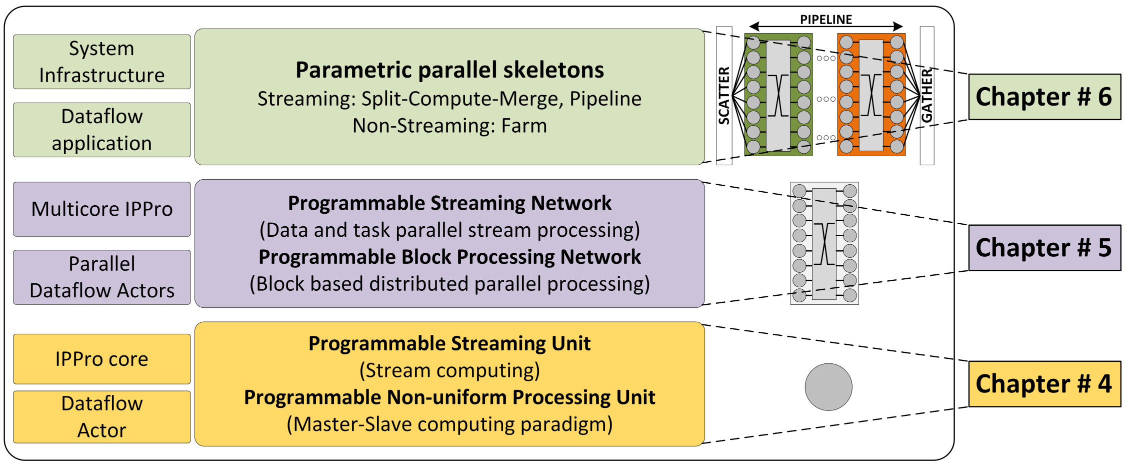

System Infrastructure
Dataflow application
Parametric parallel skeletons
Streaming: Split-Compute-Merge, Pipeline
Non-Streaming: Farm
PIPELINE
SCATTER
GATHER
Chapter # 6
Multicore IPPro
Parallel Dataflow Actors
Programmable Streaming Network
(Data and task parallel stream processing)
Programmable Block Processing Network
(Block based distributed parallel processing)
Chapter # 5
IPPro core
Dataflow Actor
Programmable Streaming Unit
(Stream computing)
Programmable Non-uniform Processing Unit
(Master-Slave computing paradigm)
Chapter # 4

Figure 1.1: Hierarchical illustration of hardware and software abstraction supported by each layer of the proposed programmable hardware acceleration architecture.

interconnect called **multicore IPPro** as shown in Figure 1.1. It extends both shared memory and stream processing and is supported by the lower layer to realise parallel computing models. The interconnect provides a deterministic, self-synchronising programmable inter-core communication mechanism to facilitate implementation of graph modelling various kinds of parallel/concurrent activities. The shared memory model offers programmable explicit synchronisation mechanism between each IPPro core and host processor to realise distributed computing and coprocessor activities.

- The top layer provides **system infrastructure** that distributes and collects data to the bottom layers. These mechanisms are necessary for efficient implementation of different parallel applications exploiting data and task parallelism as shown in Figure 1.1. Besides, it provides parametric/software configurable and dynamic data and control mechanisms to use common parallel algorithmic skeletons (split, compute and merge, farm and pipeline) and image processing operations (point and area) utilising the architectural features and processing capability provided by the bottom two layers.

The proposed approach provides a hierarchical abstraction to hardware computing resources, and the relevant communication and data access mechanisms that help to address the challenges faced by algorithm and software developers to adopt FPGAs. This approach also enables parallel exploration, profiling and implementation of different image processing algorithms to achieve the required goals.

1.4 Thesis Contributions

The following are the notable contributions presented in this thesis work:

1. Design and development of novel FPGA-based *Image Processing Processor* (IPPro) soft-core architecture tailored for acceleration of image pre-processing applications. The architecture is carefully designed to support functional computing requirements of image processing while maintaining efficient utilisation of FPGA compute and memory resources. The architecture supports both message passing and shared data models enabling stream and batch processing of uniform and non-uniform distributed data. These data processing paths provide architectural features to facilitate implementation of a *split, compute, merge, pipeline* and *farm* parallel computing skeletons. Using IPPro as a fundamental computing element makes the FPGA-based platform flexible and adaptable. It allows deployment of *edit-compile-run* flow avoiding design synthesis and place-and-route that reduces design time.

2. Design and development of IPPro-based hardware accelerator models to identify the architectural requirements of the accelerator's management and provisioning policies, and their impact on the timing results of the processor. IPPro is designed as an independent, self-managed, programmable dataflow accelerator. The program code embeds both the actor's functional description and its interaction with multiple producers and consumers. It avoids the need for external control mechanisms necessary to synchronise interaction between actors while exchanging data tokens and minimises IP-Pro core management and control overheads. It gives better controllability

on the actor's token production and consumption rate and implements different data exchange patterns (split and merge). Besides, it enables fine and coarse-grained mapping and execution of data and control flow graphs which are commonly found in image processing applications.

3. Development of a *multicore IPPro* architecture that provides flexible connectivity among multiple IPPro cores and enhances platform's programmable computing and mapping capabilities to map dataflow applications. The architecture complements the supported features of IPPro core and provides dynamic routing of dataflow streams among multiple IPPro cores. The connectivity among cores allows adaptable implementations of one-to-many, many-to-one, many-to-many producer-consumer dataflow graphs utilising the same hardware resources. These architectural features facilitate application profiling, optimisation options to the software and algorithm developer by exploiting data, task and pipeline parallelism.

4. Design and development of FPGA-based software controlled data distribution and collection architecture supporting different image resolutions. It divides an image stream into a variable number of parallel data streams that can be fed across multiple IPPro cores to realise a parallel computing paradigm. The architecture is independent, self-managed and can be integrated with both direct and buffered video processing pipelines to distribute data across multiple processing elements which are fixed in *High-level Synthesis* (HLS) system architectures. It facilitates parallel implementation of a split, compute, merge computing skeleton using multicore IPPro.

5. Design and development of an adaptable *FPGA-based hardware accelera-*

tion platform architecture. It facilitates application exploration possibilities using flexible actor-core mapping, exploiting data and task parallelism, and realisation of parallel computing skeletons on FPGA technology. The architecture provides necessary architectural functionalities to deploy edit-compile-run flow by avoiding synthesis and place-and-route times which is helpful to profile, optimise and fast prototype both parallel and non-parallel image processing algorithms on the FPGA.

1.5 Thesis Outline

The remainder of the thesis is organised as follows: Chapter 2 covers the fundamental multidisciplinary concepts of FPGA-based hardware design and implementation, and parallel computing. It includes background on parallel embedded architectures focusing on FPGA-based hardware design approaches highlighting the need of adaptable and flexible hardware architectures. This is followed by the introduction to parallel computing with a primary focus on the notion of parallelism in dataflow graphs. The literature review on different FPGA-based soft-core processor architectures using different design/programming approaches will be covered at the end of the chapter.

The presented work is a part of a larger research project called *Rathlin* and covers the underlying FPGA-based hardware architecture. Chapter 3 gives an overview of the project's scope and programming work-flow. It will help the reader to understand the bigger picture of the presented research and reasons of the adopted approach, and some of the design choices made in designing IPPro, multicore IPPro and the platform architecture.

Chapter 4 presents an FPGA-based soft-core *Image Processing Processor* (IP-Pro) architecture tailored to accelerate image pre-processing operations. The processor datapath has been developed after a detailed insight analysis of FPGA resources, processor functionality and dataflow models. It exploits FPGAs dedicated computing and memory resources to achieve the best balance between performance and area utilisation and enables software recompilation of FPGA by avoiding synthesis and place-and-route times. The processor datapath implements dedicated minimum and maximum instructions for optimised implementation of specific image pre-processing functions. A coprocessor extension is also implemented to integrate dedicated processing units and offload complex arithmetic operations transparently. At the end of the chapter, the performance and area results achieved by single-core IPPro is compared against a fixed *high-level synthesis* (HLS), FPGA-based programmable processor architecture and well-established MicroBlaze soft-core processor. The IPPro core is viable to use as a basic processing element of a *programmable hardware acceleration architecture.*

Chapter 5 presents IPPro as a *programmable dataflow accelerator* architecture that can map and execute fine and coarse-grained dataflow actor using producer-consumer computing model. These execution patterns supported by the architecture provide flexible mapping options to the user and software framework to explore and deploy different dataflow graph optimisations. It also presents a detailed analysis of management and provisioning of hardware accelerator when used in heterogeneous system architecture and their impact on the system's architectural requirements and resource utilisation.

Chapter 6 presents a heterogeneous *FPGA-based programmable hardware acceleration platform* architecture that supports a software-controlled implementation of parallel skeletons on hardware. The platform is composed of a host processor and tightly-coupled homogeneous FPGA-based programmable hardware accelerators (IPPro cores). The platform facilitates the implementation of the split, compute and merge, pipeline and farm parallel skeletons by providing software-abstraction to make it easy to use for the software developer. The platform covers three hardware and software abstraction layers as indicated in Figure 1.1. At the end of the chapter, the acceleration results of a set of image pre-processing micro-benchmarks and functions, covering data and task parallel balanced and unbalanced dataflow actors are presented. This allows the mapping flexibility and the system's adaptability to implement different applications and computing paradigms to be evaluated.

Chapter 2

Background

The changing technology landscape and fast evolution of new application use-cases raises the need for adaptable and efficient hardware architectures. These architectures shall handle the processing of dynamic data workloads and at the same time provide adaptability to implement different applications. This research problem initiated the need for look into different FPGA-based design approaches and programmable architectures. This chapter covers the multidisciplinary concepts related to FPGA-based hardware design approaches, dataflow model of computation and parallel computing and reviews their background and related work relevant to the thesis.

Section 2.1 covers the background on parallel embedded architectures focusing on FPGA-based hardware acceleration approaches and details pros and cons. of the existing approaches. It will discuss benefits of FPGA technology to realise efficient hardware acceleration technology to develop programmable/adaptable architectures. Section 2.2 encompasses the basic concepts of a dataflow model of computation and presents the notion of parallelism and dataflow transformations to achieve optimised implementations. This is followed by a discussion on parallel computing skeletons that provide high-level programming constructs suitable for software and algorithm developers in Section 2.3. Section 2.4 will review the related work on FPGA soft-core and multicore processor architectures.

2.1 Parallel embedded architectures

During the last decade, multiprocessor architectures have emerged as an important computing paradigm for parallel computing [18], [19], [20], [21]. They have driven the development of advanced parallel embedded architectures [22], [23]. The trend of integrating homogeneous and heterogeneous processing units have opened various hardware-based parallel application decomposition, mapping and design exploration possibilities [24], [25], [26], [27]. Hardware architectures composed of tens and hundreds of light-weight compute units have become a commonplace not only to optimise performance [12], [13], [14], [15], [16]. At the same time, these hardware architectures present several challenges such as architecture specific skills to port and optimise the applications to the underlying architecture which includes managing and exploiting parallelism and system heterogeneity. This section covers the background study necessary to understand these challenges and FPGA-based hardware design approaches taken by the research community.

2.1.1 FPGA multiprocessor system-on-chip

Emerging heterogenous multiprocessor system-on-chip (MPSoC) architectures such as Xilinx Zynq-7000 and Altera Arria-V SoC integrates both software programmability of a general purpose processor (ARM) with the hardware programmability of an FPGA. The integration of the hardware and software made MPSoC architectures suitable computing platform to implement mixed functionality on a single device, and to develop adaptable embedded architectures [15], [28], [29].

Nevertheless, these heterogeneous MPSoC platforms have addressed some of the hardware and software programming challenges. However, fitting of parallel computational tasks to the underlying hardware resources by using more processing nodes integrated into a single chip is still a challenge. This problem is directly related to the optimal exploration of type and degree of parallelism among multiple processing nodes available within the heterogeneous system [12], [13], [14], [30]. Besides, optimised realisation of parallel applications using these heterogeneous platforms, it often involves design and development of hardware accelerators to meet application requirements. The architecture of these hardware accelerators can have a dynamic range of flexibility from fixed, reconfigurable, software programmable or combination of thereof. They reside on the FPGA fabric and are usually managed by a general purpose processor such as ARM Cortex-A processors [28], [29]. There are different FPGA hardware design approaches to realise such hardware accelerators, Section 2.1.2 will discuss in further details.

2.1.2 FPGA hardware accelerator design approaches

The silicon vendors and the research community have developed and proposed different architectures, design tools and software frameworks that ease the development of hardware accelerators. The silicon vendors tools provide a cohesive

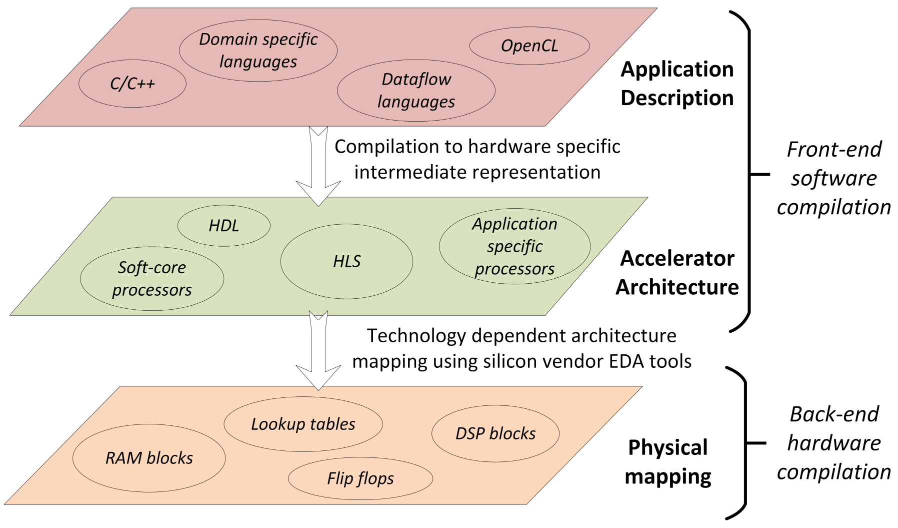

Domain specific languages
OpenCL
C/C++
Dataflow languages
Application Description
Compilation to hardware specific intermediate representation
Front-end software compilation
HDL
Application specific processors
Soft-core processors
HLS
Accelerator Architecture
Technology dependent architecture mapping using silicon vendor EDA tools
Lookup tables
DSP blocks
RAM blocks
Flip flops
Physical mapping
Back-end hardware compilation

Figure 2.1: FPGA-based hardware accelerator design compilation approaches [35].

heterogeneous hardware-software co-design solution to develop and integrate the custom FPGA-based hardware accelerators. However to realise different application use case, requires architectural changes, design synthesis and place-and-route [20], [21], [31], [32], [33], [34]. These design tools cover both hardware and software design space which can be divided into the *front-end software compilation* and *back-end hardware compilation* tasks as illustrated in Figure 2.1 [35]. The *front-end software compilation* includes *application description* and *accelerator architecture* layers. The *application description* can be a domain or target specific, while the *accelerator architecture* can encompass a wide range of hardware accelerator architectures. On the other hand, the *physical mapping* layer uses silicon vendor tools to physically map the chosen hardware accelerator architecture onto the FPGA resources to achieve *back-end hardware compilation.*

To provide programming abstraction, the application can be described in a high-level language such as C/C++, OpenCL etc. or domain-specific language

[15], [20], [36]. This application description is translated, optimised and compiled into an intermediate representation that can be mapped onto the target-hardware-accelerator. A wide range of target-hardware-accelerator approaches can be adopted ranging from a highly optimised application-specific processor, a flexible and programmable soft-core processor, an overlay architecture or a combination of thereof as illustrated in Figure 2.1. Each of these approaches has their pros and cons regarding design flexibility, area and performance [12], [20], [21], [23]. Based on the chosen target-hardware-accelerator architecture, the intermediate representation can be converted either into a set of dedicated domain-specific instructions, a program code consisting of mix of a general purpose instructions, or a *hardware description language* (HDL) or combination of thereof. The physical layer takes the HDL description of the target-hardware-accelerator design and converts into an FPGA mappable form, i.e. to the physical resources of an FPGA (flip-flops, lookup tables, dedicated DSP and memory blocks). This task requires technology dependent optimisation and routing mechanisms which are conducted by automated silicon vendor tools. The tasks involve design synthesis, place-and-route and bit-stream generation. These steps can be significantly time-consuming for iterative algorithm developement depending on the complexity and size of the hardware design [8], [9], [10], [37].

High-level synthesis (HLS)

High-level synthesis (HLS) tools take an application description, use different analysis techniques to profile and explore the design space. The majority of these tools use a dataflow model of computation, therefore Table 2.1 lists both academic and commercial HLS tools that are widely reported in the open literature [38], [39]. These tools support different high-level languages such as C, C++, OpenCL or domain specific languages to describe an application. These tools profile, explore, optimise and compile the high-level description into an intermediate representation which is translated into hardware description languages [7], [40], [41] such as VHDL, Verilog or SystemC as listed in Table 2.1.

These tools take advantage of FPGA deep pipelining to exploit parallelism and explore performance and resource optimisations by tuning the size of the *first-in-first-out* FIFOs [41], [42]. The oversized buffer uses more resources than needed, while small buffer can cause additional delays, stalls, and deadlocks during execution of the application [40], [43]. Though, all HLS tools generate a fixed hardware architecture tailored to accelerate a specific application or part of an algorithm which is not adaptable. To implement different applications, the only possibility is to rewrite and go through all front-end and back-end tasks discussed in Figure 2.1. The back-end tasks can significantly increase the design time [8], [9], [21], [23], [37] which is not appealing by software and algorithm developers due to the iterative algorithm development process which requires design exploration and fast prototyping. Section 2.1.3 discusses this problem in detail.

Table 2.1: High-level Synthesis (HLS) tools for FPGAs. [38], [39].

HLS Tool	**License**	**Input**	**Output**	**Data flow**	**Control Flow**
Catapult-C	Commercial	C/C++/SystemC	VHDL/Verilog/SystemC	✓	✓
Bluespec	Commercial	BSV	SystemVerilog	✓	✓
C-to-Silicon	Commercial	SystemC/C++	Verilog/SystemC	✓	✓
MaxCompiler	Commercial	MaxJ	RTL	✓	✗
ROCCC	Commercial	C subset	VHDL	✓	✓
GAUT	Academic	C/C++	VHDL	✓	✓
Synphony C	Commercial	BDL	VHDL/Verilog	✓	✓
LegUp	Academic	C	Verilog	✓	✓
Vivado HLS	Commercial	C/C++/SystemC	VHDL/Verilog/SystemC	✓	✓
Altera SDK	Commercial	C/OpenCL	VHDL/Verilog	✓	✓
HIPAcc	Academic	C++ Embedded DSL	C++	✓	✗
Merlin Compiler	Commercial	C/C++	C/OpenCL	✓	✓

2.1.3 Need for adaptable hardware architectures

The emerging versatile application markets raise the demand for high-performance and efficient FPGA architectures, that can handle the processing of dynamic data workloads and at the same time adaptable to accelerate different applications. One way to approach this research problem is by developing adaptable FPGA hardware architecture that enables *edit-compile-run* flow familiar to software and algorithm developers instead of hardware *synthesis and place-and-route*. It can be achieved by populating FPGA logic with a light-weight and high-performance soft-core processors used for programmable hardware acceleration. This underlying architecture will be adaptable and can be programmed using conventional software development approaches as illustrated in Figure 2.1. This approach does not require hardware design synthesis and place-and-route. Instead, it will need software re-compilation that shall generate a binary code to run on the underlying soft-core processors.

Though the HLS-based designs are use case optimised as the application is known before realising the underlying hardware. On the contrary, in processor-based approach, the underlying hardware architecture is designed, synthesised, place-and-route in advance. Therefore, the overall area is expected to be more significant and performance is supposed to be lower than HLS, which will come at the cost of adaptability and reduction in design time.

This approach provides hardware abstraction of the underlying FPGA programmable resources by allowing them to reconfigure using traditional software approaches and exposes it to the software developer. It inherits software benefits such as portability, partitioning complex hardware-software co-design, decomposition and mapping options to achieve desired area and performance goals. Besides, avoiding required iterative process of synthesis and place-and-route would reduce design time, improve productivity and allow software-controlled design exploration opportunities. Jain, Rigamonti and Liu have reported an order of magnitude improvements by compiling applications onto processor architectures over HDL and partial reconfiguration approaches [23], [44], [45]. Nevertheless one of the significant challenges is to efficiently compile, map and execute parallel applications onto the underlying programmable hardware accelerator architecture.

2.1.4 FPGA memory and computation resources

FPGA fabric provides essential digital components necessary to build any digital circuit. It has logic blocks, dedicated memory and DSP Blocks, clock management circuitry and routing resources to connect these digital components. In an FPGA, the location of these components are fixed and cannot be changed which makes it essential to consider their layout to obtain area-efficient and high-performance hardware architecture. Figure 2.2 shows the available hardware resources, their raw-computation (GMACs) and the memory resources across different families of Xilinx FPGAs. The raw-computation (GMACs) is directly proportional to the number of DSP blocks. Dinechen *et al.* show how to map different arithmetic operators on FPGA fabric utilising different approaches including LUT, DSP block etc. [46]. Similarly others presented mapping of mathematical expressions to these DSP blocks and achieved performance improvements [47], [48], [49].

While the computing resources and bandwidth are high, the memory in FPGA is limited compared to other computing technologies. Figure 2.3 shows the distri-

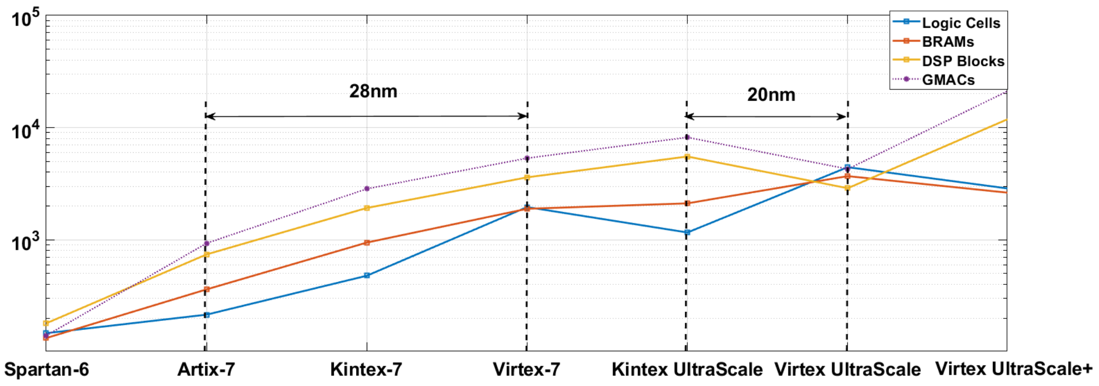


Figure 2.2: Trend of hardware resources, their raw-computation (GMACs) and memory across different families of Xilinx FPGAs [50], [51].

bution of on-chip memory and bandwidth on the Virtex-7 FPGA. Moving away from the datapath the memory size increases while the bandwidth get limited. On-chip memory consists of LUT-based Distributed RAM that are small and close to the datapath which can provide faster access to data at higher bandwidth. On the other hand, Block RAM is comparatively larger but limited in bandwidth. It shows that there is a trade-off between the memory-size and bandwidth.

Focusing on the FPGA technology, the 7 series Xilinx FPGAs comes in three different families. The 7 series combine the power reducing process, design tech-

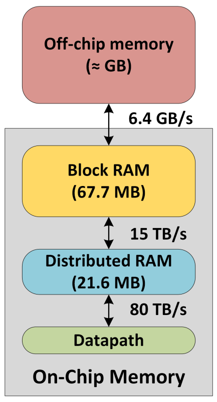


Figure 2.3: FPGA memory and bandwidth hierarchy of Xilinx Virtex-7 FPGA.

niques, and architectural enhancements to deliver the lowest-in-class power consumption, compared to the previous generation of Xilinx FPGAs. It covers a low-cost Artix-7 family, a midrange Kintex-7 family, and a high-end Virtex-7 family of FPGAs. All three families uses the same 28nm silicon process technology and have the basic FPGA building blocks of logic cells, DSP blocks, BlockRAM making it simpler to migrate designs across FPGA family. The Kintex-7 device family features a perfect balance of FPGA fabric clock rate performance versus power consumption, high-speed I/O, capacity and reliability. Artix-7 uses the same FPGA resources as Kintex-7, but optimized for even lower power consumption and smaller size packages, delivering similar advantages, at the cost of lower chip price and performance.

2.1.5 DSP block

Most of the digital signal processing applications extensively use multiply and accumulate operations that can be performed efficiently using these DSP blocks. These blocks are uniformly-distributed inside the FPGA fabric. They are capable of performing basic arithmetic and logic operations on data that is suitable to design efficient, high-performance *arithmetic logic unit* (ALU) of a processor. Xilinx DSP Block (DSP48E1 and DSP48E2) supports these operations and can be dynamically configurable in contrast to Altera. Figure 2.4 shows the simplified functional block diagram of the DSP48E1. It has four main arithmetic blocks:

1. 25-bits Pre-Adder
2. 25x18 bits Multiplier
3. 48-bits Adder, Subtractor, Logical

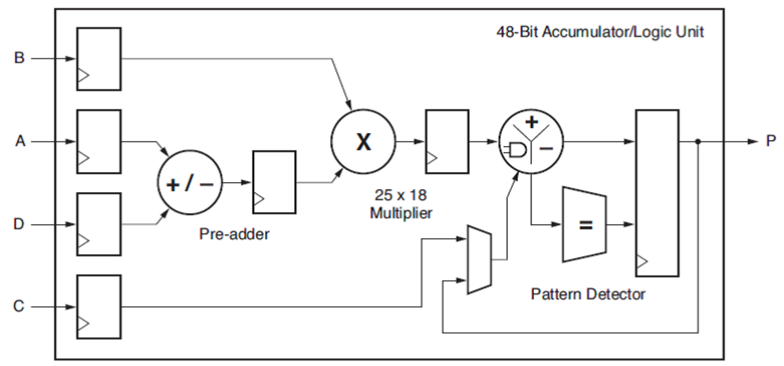


Figure 2.4: Block diagram of Xilinx dedicated DSP block (DSP48E1) [52].

4. Comparator and pattern detector

The DSP48E1 is capable of multiply, multiply- accumulate, add, subtract and other operations. Besides, a set of control registers that allow controlling the internal datapath on a cycle-to-cycle basis (for details see Appendix B Table B.1). There are pipeline registers that enable/disable the internal pipelining of the DSP48E1 block and improve the timing of the block by reducing the critical path [52]. Three internal multiplexers allow mapping of input and output operands to multiplier and adder/subtracter.

2.2 Dataflow model of computation

In early 1970s, various classes of model of a computation (MoC) had been introduced that models the architecture independent functional requirements through semantics, interfaces and provides synergy between processing units [53], choosing a suitable MoC is one of the key hardware design decision. The dataflow MoC could possibly be expressive programming and efficient execution model. It has the property to express applications as network processes which offer parallelism scalability, modularity, portability and adaptivity. These characteristics are vital to unify the system level design of heterogeneous platforms. Moreover, it follows the principle of stream processing [22] that are suitable for FPGA-based hardware architectures [54], [55], [56], [57].

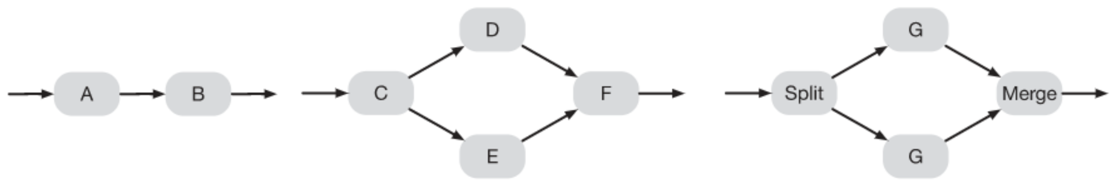
2.2.1 Notion of parallelism in dataflow graphs

Stream and dataflow driven programming models allows efficient implementation of different types of parallelism [30], [58], [59]:

Pipeline parallelism A pipeline is a chain of actors $a_1, \ldots, a_n$ that are directly connected in the stream graph. Each pair $(a_i, a_{i+1}), i \in \{1, \ldots, n-1\}$ has a producer/consumer relationship, that is, a_i consumes items produced by a_{i-1} and produces items that serve as input for a_{i+1}. Figure 2.5 shows a pipelined execution of function A and B. It is important to note that the throughput shall only be as fast as the slowest group of actors in the pipeline [60].

Task parallelism Two actors a_1, a_2 are task parallel if they are on different branches of the stream graph. In contrast to pipelines, there are no input/output dependencies between a_1 and a_2. Figure 2.5 shows task parallel actor D and E.

Data parallelism is the property of an actor to have no dependencies between



(a) Pipeline-parallel A || B. (b) Task-parallel D || E. (c) Data-parallel G || G.

Figure 2.5: Illustration of pipeline, task and data parallelism in dataflow graphs [58].

one execution and the next. The actor can be replicated by using multiple instances of an actor such as G is replaced twice as shown in Figure 2.5.

2.2.2 Dataflow transformation

Dataflow transformations are frequently used to enhance system performance, by improving the performance of slower dataflow nodes or part of the graph [24] [26]. These transformations maintain the functionality of original dataflow graph, but increase the throughput or decrease the latency [26], [30]. Dataflow graphs are amenable to coarse-grained transformation to exploit data, task and pipeline parallelism that can be efficiently implemented using FPGA [24]. *Single instruction multiple data* (SIMD) based hardware architectures had been used to accelerate applications including image pre-processing due to massive pixel processing [61], [62], [63]. The dataflow specific optimisations (decomposition, mapping, and scheduling) and transformations (fission, fusion, etc.) can be exploited to improve performance [24], [30], [64]. These transformations allow decomposition and design space exploration possibilities to achieve desired application goals. The application can map on a multicore architecture, which will enable exploiting data and task parallelism by supporting edit-compile-run design flow.

2.3 Parallel computing skeletons

Parallel computing skeletons capture common parallel-programming paradigms and abstract to the programmer as high-level programming constructs equipped with well-defined functional semantics [12], [65], [66], [67], [68]. They model a precise parallel pattern to exploit parallelism and hides pattern implementation details from the programmer to exploit parallelism as shown in Figure 2.6. These patterns are *parametric* and can be *re-used* in different applications. This approach is adopted by several parallel programming frameworks [31], [69], [70].

2.3.1 Pipeline

The basic idea of the *pipeline* skeleton is to split processing into a series of sequential steps, with storage at the end of each step as shown in Figure 2.6. It is possible by distributing a sequential application into multiple independent but sequential tasks, where preceding task feeds data to the following task. It enables concurrency where that tasks can execute in parallel as soon as the data is available at the processing node. The computational load of tasks may vary and is not known before run-time unless static model of computation such as static dataflow is used to define the parallel application. Though, the maximum achievable processing rate depends on the processing rate of the slowest task, which is faster than the time needed to perform all the steps at once. However, by static profiling of the application in hand, it is possible to find an efficient decomposition that could lead to balanced tasks with bounded memory requirements.

2.3.2 Split, compute and merge

This skeleton is used to process regularly distributed data-based on static decomposition. The data is divided into a number of equal sized blocks (row-based, column-based or block-based) where the number of parallel data blocks defines the level of exploitable data parallelism. In architectural terms, it is know as *scatter-gather* or *split-compute and merge* parallel programming model as shown

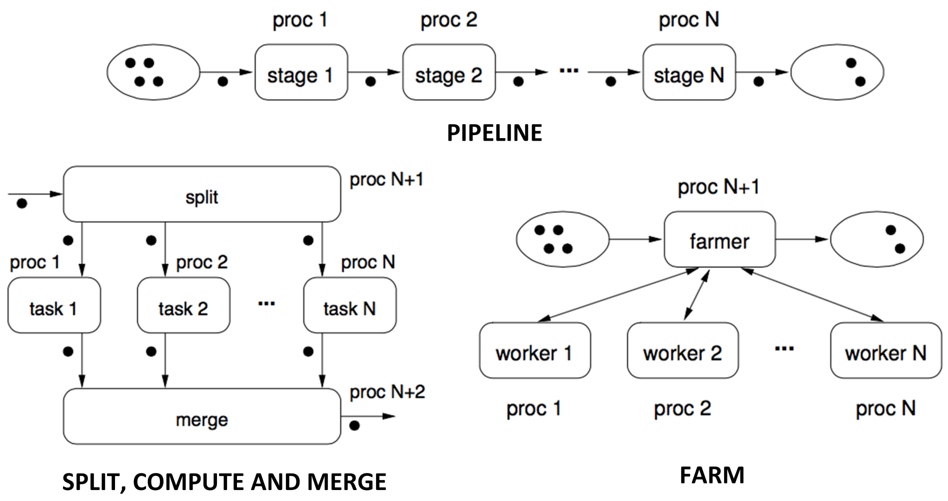


Figure 2.6: Illustration of parallel computing skeletons using dataflow actors [71].

in Figure 2.6. Moreover, it can also be extended to implement different derived multi-stage pipelined skeletons to exploit both data and task parallelism using the *pipeline*, *split, compute, communication, compute, merge* or a combination of thereof, to achieve better performance. The benefit of pipelining multiple stages is that it reduces data transfer overhead, improves data bandwidth, avoids memory bottlenecks in contrast to shared memory model-based acceleration approach where the bandwidth and cache coherency significantly degrade the performance.

2.3.3 Farm

This skeleton is used to process irregular data. The farmer (host/master processor) allocates the tasks to the workers until none are left as shown in Figure 2.6. Then, the farmer waits for a result from a worker and immediately sends another work item to it. Each worker receives a work packet, process it, and returns the result to the farmer until it gets a stop condition from the farmer. The advantage of this approach is that the farmer knows which workers have yielded the results of their tasks and are hence idle. Thus, the farmer can forward incoming tasks to the idle workers.

However, this approach has its disadvantages. It causes substantial overhead due to the exchange of messages between the farmer and the workers [72]. Moreover, the farmer might become a bottleneck, if the number of workers is large. In this case, the farmer will not be able to keep all workers busy, leading to wasted workers. The number of workers which the farmer can keep occupied depends on the sizes of the tasks and the sizes of the messages the farmer has to propagate. However, since the process of assigning task is cyclic which could lead to deadlocks in case of data dependent tasks where the computation of certain workers might depend on the results of the others leading to deadlocks. On the other hand, one has to make sure that the farmer reacts as quickly as possible to newly arriving tasks and workers delivering their results.

This section has discussed the concept of parallel computing skeletons which will be used to approach the issue of lack of hardware abstraction in FPGA-based architectures. Since application designers face difficulty, utilising the available resources efficiently without hardware knowledge. It involves handling of the low-level core, inter-core and system communication and system interfaces etc. Therefore, Chapter 6 will present a detailed multicore and system level architecture that shall support these parallel computing skeletons.

2.4 Related work on FPGA soft processors

This section investigates different FPGA-based soft processor architectures. Emphasis will be placed on the various word sizes, maximum clock frequencies, and resource usage to evaluate these processors. *Word size* is essential parameter because at least 16-bit words are required to accurately represent the pixel data in different colour spaces with some redundancy. *Clock frequency* directly influences the maximum throughput of the design, which in turn affects the observed speed-up. Processors with less resource usage allow more logic to be used for multicore architectures, which can achieve superior performance.

2.4.1 Scalar Processors

The commercial off-shelf offering from leading FPGA vendors Xilinx and Altera are the *MicroBlaze* and the *Nios II* processors respectively. Both are 32-bit soft processors-based on RISC architecture accompanied by their respective software development tool-chains. The performance optimised MicroBlaze is capable of delivering up to 262 DMIPs having a 5-stage pipeline, while Altera's capable of delivering 30 DMIPS. Both the Nios II and MicroBlaze are highly configurable with options including a floating point unit, memory management, and interfacing to custom hardware accelerators. These commercial soft-core processors have been investigated and modified in several papers [73], [74]. But, managed to achieve the maximum operating frequency ranging from 77 - 112 MHz.

Other processors are made available under open source licenses such as the OpenRISC. It is an open source RISC-based processor with 32 and 64-bit modes and optional vector support [75]. The LEON3 is a 32-bit SPARC V8 compliant processor described in VHDL which is available under the GNU GPL [76]. It uses 7-stage pipeline, incorporates a floating point unit, supports symmetric multiprocessing and operates at up to 125 MHz.

2.4.2 Multicore Processors

In open literature, many FPGA multicore processor architectures have been presented to accelerate different applications. Silicon Hive [77] accelerator architecture replaced ASIC accelerators with the reconfigurable cores, making accelerators fully programmable after fabrication and flexible to maintain throughout the product life-cycle. The basic component of Silicon Hive architecture is the *Processing and Storage Element* (PSE) consists of multiple functional units (FU) connected via interconnect network (IN) as shown in Figure 2.7. It has one or more operation-issue slots (IS) associated with the FUs, distributed register files (RF) and an optional local memory storage (MEM). The PSE was designed in such as way that it ensures easy and clean datapaths for a compiler to handle, and guaranteeing high-level of programmability. A matrix of one or more PSEs, together with a controller (CTRL) and configuration memory (CONFIG. MEM), makes up a *cell*. The PSEs within a cell can communicate with each other via

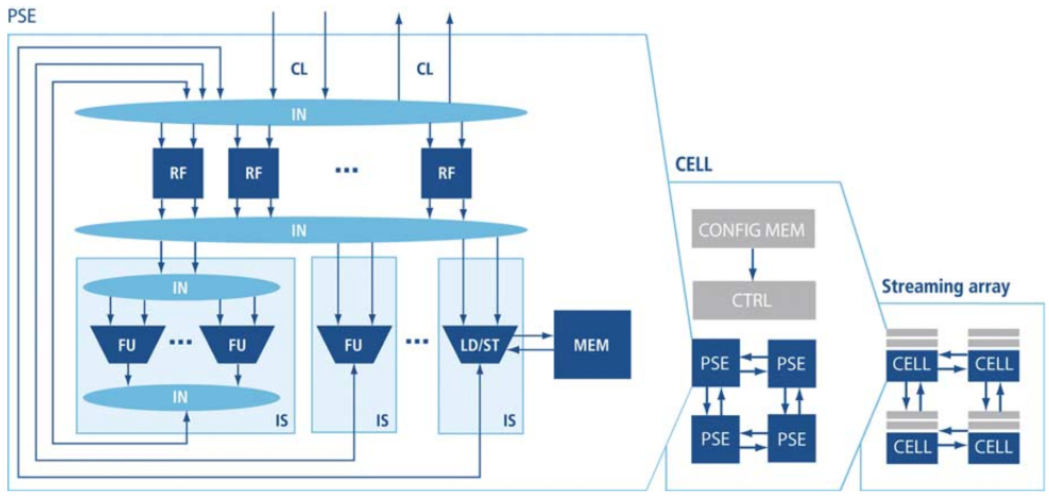


Figure 2.7: The layered block diagram of Silicon Hive architecture illustrating *Processing Storage Element* (PSE), *cell* and *streaming array* of cores [77].

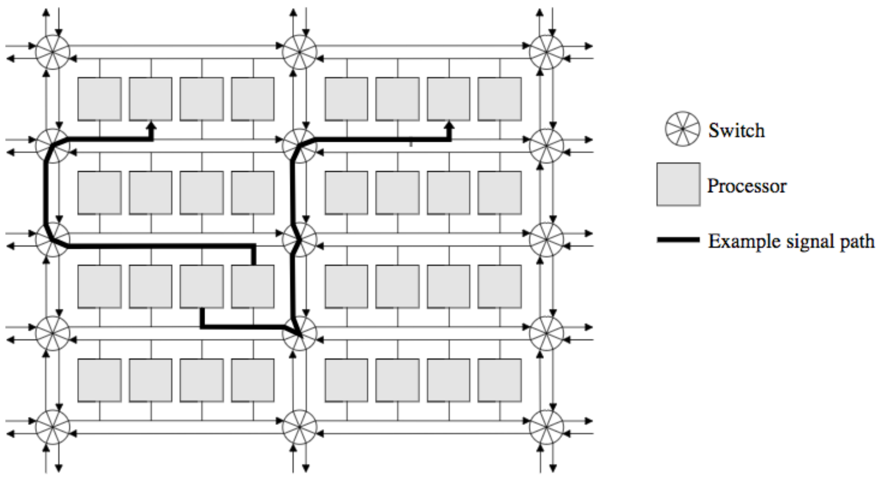


Figure 2.8: The block diagram of PicoArray processors organised in a two dimensional grid connected together using a deterministic picoBus interconnect [79].

data communication lines (CL). An array of one or more *cells* connected via a data-driven communication mechanism forms a *streaming array* as shown in Figure 2.7. The communication across cells takes place through blocking FIFOs accessed from load/store (LD/ST) units within the cells, allowing multiple functions to be concurrently mapped onto the *streaming array*. Dan *et al.* extended the Silicon Hive approach and proposed HiveFlex Moustique-IC2 processor [78] as a synthesisable soft-RTL core with an I/O subsystem specifically designed for image processing applications. The Moustique-IC2 was a Single-Instruction-Multiple-Data (SIMD) machine, which means that the same program simultaneously operates on all pixels. By increasing the SIMD factor, the same program can be used to process more pixels at once, thereby increasing the throughput. The 24-way SIMD processor achieved the operating frequency of 200 MHz on a 90 nm technology.

Duller *et al.* have proposed PicoArray [79] which is a massively parallel architecture designed as an alternative for creating ASIC designs which are complex to design, expensive in cost and requires larger design time and effort. The PicoArray is a tiled processor architecture, composed of a large number of heterogeneous processing cores. The architecture had been primarily designed for wireless infrastructure applications. The processors are organised in a two-dimensional grid and connected together using a deterministic picoBus interconnect as shown in Figure 2.8. The inter-processor communication protocol was based on a time division multiplexing (TDM) scheme, where data transfers between processor ports occur during automatically scheduled time slots by the tool and controlled by the bus switches. The communication between the processors is fixed at the compile time and cannot be changed dynamically. The PicoArray is designed as a 16-bit, 3-way VLIW RISC processor with Harvard memory architecture. It supports four different variants of processors (standard, multiply-accumulate, memory and control). Each variant was designed for a mixture of DSP, stream and block-based processing and therefore, had different internal memory distribution. All four variants use the same RISC instruction set, except the MAC instruction which can only be executed on standard processor. With the exception of loads and branches, all instructions execute in a single cycle. Each processor can only access its own internal memory (between 1KB and 32KB) and communicates with other processors using input/output data ports. Each processor was initialised using a special configuration bus and programmed using assembly language. The PicoChip PC102 runs at 160 MHz on Xilinx Virtex-4 FPGA [80].

Classical vector processing involves sending a stream of values into pipelined functional units [81]. Later, there are several architectures have been proposed including [82] and [83]. Some optimisations have been done to speed up the performance and to reduce the execution time by incorporating vector chaining, control flow execution and banked register file *etc.* The processor runs at a maximum operating frequency of 200 MHz [83]. Others proposed multiprocessor architectures to exploit the hidden parallelism in some parts of streaming applications for efficient implementation such as VENICE and VectorBlox MXP are designed to exploit data level parallelism (DLP) by processing vectors [83], [84]. Nachiket *et al.* has proposed a GraphSoC custom soft processor [16] for accelerating graph algorithms using Xilinx Zynq SoC. It is 3-stage pipelined processor that supports graph semantics (node, edge operations). A single FPGA can fit multiple instances of these processors interconnected using *network-on-chip* (NoC). The graphs functional description is stored in the on-chip BRAM for fast local access. Larger graphs can be partitioned into sub-graphs and loaded one-by-one or split across multiple processors. The execute stage of the processor is customisable and supports four graph specific instructions, i.e. (send, receive, accumulate and update) which are implemented as micro-coded datapath shown in the Figure 2.9. The processor datapath has no register file instead, it has special purpose registers to hold edge and node information. The reported timing results shows

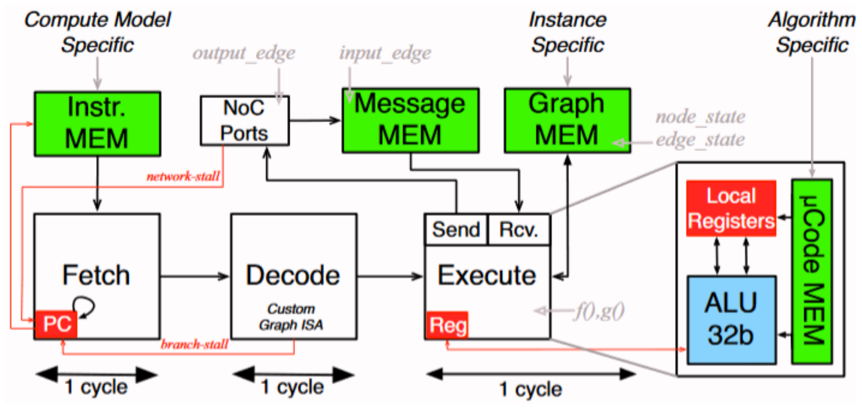


Figure 2.9: Datapath of a basic pipelined processing node used in GraphSoC [16].

that a fully pipelined design can run at 200 MHz.

Andryc *et al.* have proposed a FlexGrip [36] a customizable softcore architecture that allows the execution of *general-purpose processing units* (GPGPU) code on an FPGA without the need of design synthesis and place-and-route. FlexGrip is a 32-bit multicore scalable, configurable processor architecture based on a single instruction, multiple-thread (SIMT) model in which an instruction is fetched and mapped simultaneously on multiple scalar processors (SPs) as shown in Figure 2.10. A streaming multiprocessor (SM) is composed of multiple SP that enable multi-threaded execution. The number of threads are equivalent to the number of scalar processors inside a streaming multiprocessor (SM). SM is a five stage pipelined architecture consists of fetch, decode, read, execute and write stages as shown in Figure 2.10. The execute stage consists of multiple scalar processors and a single control flow unit. This unit operates on control flow instructions such as branch and synchronization instructions. Each thread is mapped to one scalar processor, enabling parallel execution of threads. The Write stage stores intermediate data in the vector register file, memory addresses

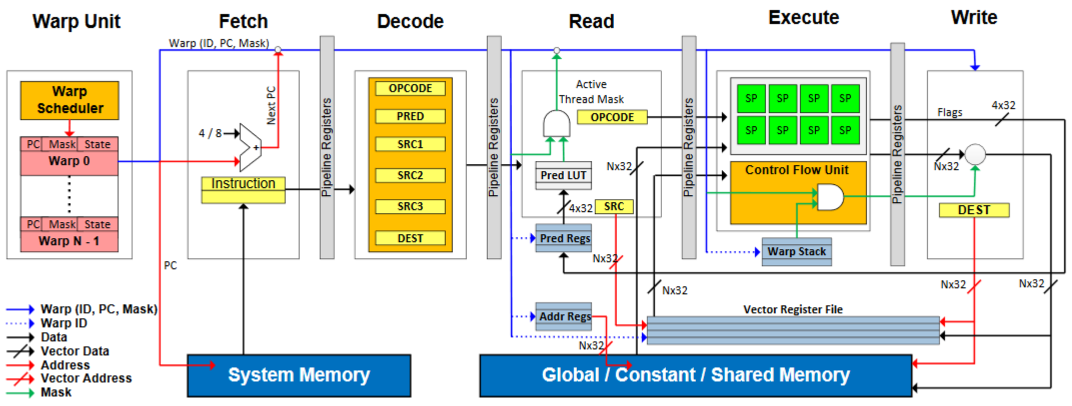


Figure 2.10: Datapath of FlexGrip *Streaming Multiprocessor* (SM) [36].

in the address register file, and predicate flags in the predicate register file. Final results are stored in the global memory. A design with single SM and 8 SP implemented on Xilinx Virtex-7 device achieved maximum operating frequency of 100MHz.

2.4.3 DSP Slice Processors

In 2009 the concept of using the DSP slice on Xilinx FPGAs as the basis for a soft-core processor was presented by Milford and McAllister [85]. In this paper, the authors design the FPGA *Streaming Element* (fSE) which is 8-stage pipelined and uses device primitives to maximise the efficiency of the processor. The instruction width is 22-bit where, two bits for the opcode, 32-bit data word and 16-bit for real and imaginary components. They implemented a 16-point FFT and compared it against the Xilinx dedicated IP core implementation. The processor not only runs at a faster- operating clock speed (430 MHz) but also uses fewer LUTs (145) and requires fewer cycles to complete. The same authors adopt this fSE processor as the basis for a 16-way SIMD processor architecture [47]. They also include custom units for minimum and switch operations to decrease the instruction count for their chosen application of a sphere decoder. They have achieved real-time performance for the 802.11n standard with a clock speed of 265 MHz on a Xilinx Virtex-5 FPGA.

Cheah *et al.* have proposed iDEA processor-based on the DSP48E1 primitive blocks [86]. It is based on a RISC load/store architecture and executes 32-bit instruction words on 32-bit data. They investigated a range of pipelining configurations and achieved a maximum of operating frequency of 407 MHz with a

9-stage pipeline on a Virtex-6 FPGA. Their design uses two 36 Kb BRAM primitives, 1 DSP slice, 404 Slice Registers, and 335 Slice LUTs which compares to the smallest Microblaze. While providing a lower instruction count than the Microblaze for three test applications (Fibonacci, FIR, Median), cycle count performance has impacted by lack of data forwarding requiring leading use of NOPs.

After reviewing different soft-core processor architectures, the major architectural challenge to design a light-weight soft-core processor architecture is to find the best trade-off between *functionality* (ability to execute different kernels) and *performance* (ability to meet performance requirements). Because the design choices made at the soft-core level dictates the functionality and performance of the multicore and hardware acceleration platform.

2.5 Summary

In this chapter, we have discussed the significance of heterogeneous MPSoC platforms in Section 2.1 that provides hardware acceleration opportunities by providing FPGA programmable logic, which can be used to accelerate computation intensive portion of an application. However, a major inhibitor to use this technology to realise adaptable solutions is the lack of hardware abstraction and complexity of FPGA design flow, especially for software and algorithm developers. Both commercial and academic research community have developed *high-level synthesis* (HLS) tools that allow programming FPGA in high-level programming languages which are familiar to software and algorithm developers, *i.e.* (C, C++, SystemC, OpenCL etc.). But, these tools generate the application description that requires synthesis, place-and-route which can be significantly time-consuming for iterative application development process due to lack of adaptability.

To approach this problem, we propose a multicore processor approach that shall replace the traditional hardware synthesis, place-and-route to *edit-compile-run* design flow. This approach will allow hardware abstraction to the underlying FPGA resources and provide adaptability by programming the underlying architecture using conventional software development approaches. Section 2.4 reviewed range of soft-core processor architectures and shows that they are either not area-efficient or does not deliver high raw-computation evaluated in terms of their maximum operating frequency (f_{Max}) essential for hardware acceleration. It is vital that the soft-core processor shall be light-weight, high-performance and efficiently utilises the FPGA resources. Therefore, Chapter 4 will present the novel *Image Processing Processor* (IPPro) architecture that will be used as a basic computational unit to realise the flexible and adaptable multicore architecture.

Section 2.2 has briefly covered the concepts and related-work necessary for a novel FPGA-based soft-core *Image Processing Processor* (IPPro) architecture presented in Chapter 5 to map and execute dataflow actor. Besides, the notion of parallelism and transformations is covered to set the background for multicore IPPro architecture presented in Chapter 6.

Section 2.3 has discussed the concept to model parallel patterns to exploit parallelism. They hide the pattern implementation details and underlying hardware peculiarities from the programmer and provides clean and give a clean abstraction to the programmer to exploit parallelism. These patterns are portable, reusable and shall be supported in the underlying hardware architecture. Implementation of these parallel patterns is central to realise our proposed hardware acceleration approach. Chapter 6 will present a detailed multicore IPPro and system architecture that support the adaptable hardware implementation of discussed parallel skeletons.

Chapter 3

Rathlin Project

To approach the outlined research problems, a collaborative research project called *Rathlin* had undertaken between *Queen's University Belfast* and *Heriot-Watt University* which was funded by *Engineering and Physical Sciences Research Council* (EPSRC) [5]. The scope of the project is to investigate the rapid developments in image acquisition/interpretation and intelligent algorithms. As FPGA-based hardware architecture development have not been matched by sound software engineering principles, to generate efficient solutions for time, memory and power efficient hardware.

One of the primary objectives of *Rathlin* project was to design and develop an *FPGA-based hardware acceleration platform* architecture for image processing applications which was my contribution to the project. The aim was to unleash the potential of state-of-art FPGAs in close synergy with a suitable software representation. This representation allows application and a programming environment to facilitate exploration, profiling and optimisation and parallel implementation of image processing applications using conventional programming approaches. Therefore, some design decisions and choices in the presented work have been driven by the scope of *Rathlin* to complement its aim and objectives.

This chapter aims to present the key objectives of *Rathlin*, its programming workflow and the model of computation. As it gives a bigger picture of the performed research and understand some of the design decisions to derive the *Image Processing Processor* (IPPro), *multicore* IPPro and the *FPGA-based hardware acceleration platform*.

3.1 Rathlin Objectives

The primary project objectives are:

- Creation of a dataflow model of computation representation that allows the processing and data organisation needs of image processing algorithms.
- Design and development of an adaptable *FPGA-based hardware acceleration platform* using the IPPro soft-core processor and focusing on the efficient utilisation of FPGA resources while matching the computational and memory requirements of the algorithms/applications.
- Development of a programming environment for a Domain-Specific Language (DSL), optimally compiled to the platform using dataflow techniques and integrated with a standard *Application Programming Interface* (API) to execute on the underlying hardware platform.
- An adaptable realisation of a set of image processing algorithms to evaluate the performance and adaptability of the platform to accelerate different image processing applications.

3.2 Programming workflow

One of the project objectives is the adaptable realisation of image processing algorithms on a FPGA-based hardware platform. Such a realisation consists of various stages as illustrated in Figure 3.1. From top to bottom it involves algorithm development in *Rathlin Image Processing Language* (RIPL) [6] that was being developed by Heriot-Watt University, a dataflow language *Cal Actor Language* (CAL) [87] and IPPro-based hardware platform.

The programming workflow consists of RIPL DSL, an intermediate representation and a compiler framework to profile and optimise the IPPro code generation

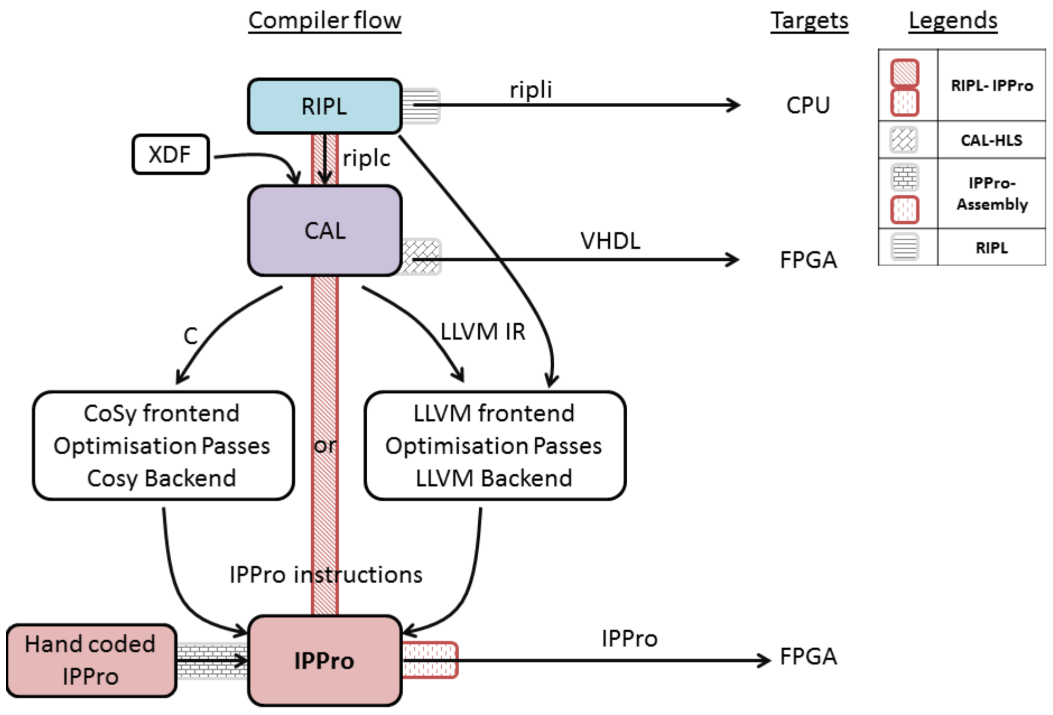


Figure 3.1: Rathlin workflow of RIPL to IPPro-based platform with alternative compilation paths [88].

that can execute on the platform. The CAL language has been chosen as the intermediate dataflow language between the RIPL and a compiler framework that generates the IPPro code. Because, the CAL compiler (Orcc [89]) allows to generate application specific implementations for different target platforms (C/JAVA for the CPU, VHDL for the FPGA) using the available runtime libraries. This flexibility has enabled alternative design routes to the project team members to carry on their research activities by implementing, verifying and benchmarking different applications without dependent on the IPPro design route. The interaction and connectivity details between dataflow actors is described as *XML DataFlow* (XDF). From the user perspective, the compiled IPPro code can run on the FPGA-based hardware platform as executable binary code which avoids the need of synthesis, place-and-route and bit-stream generation.

3.3 Cal Actor Language (CAL)

CAL is a programming language-based on the dataflow MoC where the actor executes a sequence of discrete computational steps known as *actor firing*. In each step, an actor may (a) consume a finite number of input tokens, (b) produce a finite number of output tokens, and (c) modify it's internal state if an actor has any. In CAL, it has specified as one or more actions. Each action describes the conditions under which it may be fired. It includes the availability and the values of input tokens, the actor's state and what happens when the action is triggered, i.e. how many tokens are consumed and produced at each port, the values of the output tokens, and how the actor state is modified. The execution of such an actor consists of two alternating phases: the determination of an actor

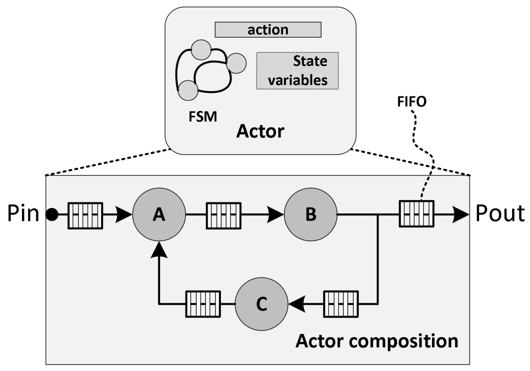


Figure 3.2: Block diagram of a CAL dataflow actor and its components [7], [54].

firing conditions are fulfilled, and the execution of that actor itself. In this work, a single action per dataflow actor has been considered scoping the research work to static dataflow than covering dynamic dataflow graphs.

3.3.1 Semantics and execution model

A CAL actor is defined by a set of input ports (P_{in}), output ports (P_{out}), actions, internal variables and a dataflow network. The dataflow network is composed of a set of dataflow actors A, B and C, and set of FIFOs depicted in Figure 3.2. An action is activated according to its input patterns known as *actor firing rule*. The patterns are determined by the amount of data required for the input sequences that need to be satisfied for enabling the execution of an action.

The CAL execution model is the execution of four stages. The execution starts by checking the *actor firing rule* which defines the number of expected input tokens from each port and output tokens produced by an actor. Once the *actor firing rule* is satisfied, the CAL actor execution starts sequentially by reading the input tokens followed by execution of the actor and storing the produced output tokens into the output FIFO queues. The following are the key advantages of

Table 3.1: Dataflow semantics and their functional requirements to implement on a hardware architecture [7], [54].

CAL semantics		Functional req.	Description
Components	Entry, Exit, Port, Bus, InBuf, OutBuf, dependency, control port and buses	Input/output FIFOs, control instructions	It is a node in a data flow graph that describes a module and the firing rules
Operations	And, or, not, mux, reg, ValueOP, UnaryOP, BinaryOP	Arithmetic and logical instructions	An operation carry out arithmetic, boolean operations and generates a single value.
Memory elements	Allocation, memory access, absolute memory read/write	Read/write access to local memory	It is a symbolic representation of a memory space in a design. The two primary attributes are the allocation and access of memory elements.

adopting a dataflow model:

- Intuitive and easily understood by programmers especially in DSP.
- Ability to express concurrency without complex synchronisation.
- Explicitly exposes the natural parallelism (pipeline, task, data).
- Modular programming allows reusability, reconfigurability and hierarchical composition of processing blocks.

The CAL semantic is list down in Table 3.1, categorised into *components*, *operations* and *memory elements* along with the identified functional requirements to map each on the FPGA-based hardware architecture. It can observe that the underlying hardware architecture shall have an input/output FIFOs, ALU and memory to support execution of CAL programs. These functional require-

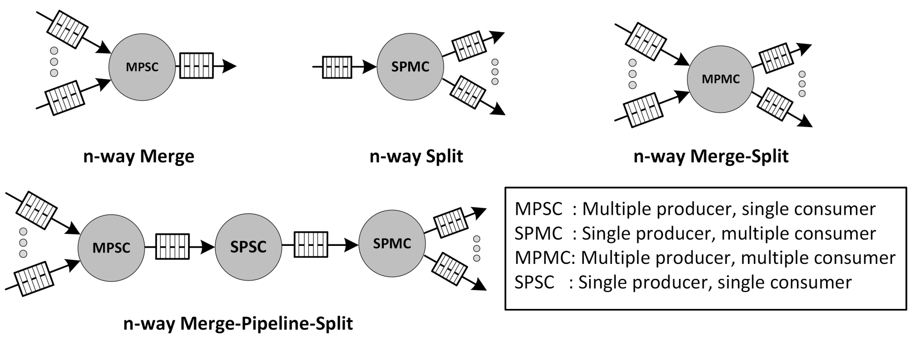

MPSC
n-way Merge
SPMC
n-way Split
MPMC
n-way Merge-Split
MPSC
SPSC
SPMC
n-way Merge-Pipeline-Split
MPSC : Multiple producer, single consumer
SPMC : Single producer, multiple consumer
MPMC: Multiple producer, multiple consumer
SPSC : Single producer, single consumer

Figure 3.3: Producer-consumer driven data exchange patterns [91].

ments lay down the architecture requirements of the IPPro soft-core processor as a dataflow accelerator which is discussed in Chapter 5.

3.4 Producer-consumer computing

The producer-consumer model is a data-driven data exchange mechanism which is widely used by dataflow-based hardware architectures to pipeline multiple actors and computing stages [55], [56], [57]. Generally, a FIFO data structure is used to implement and ensure deterministic and deadlock-free access to data tokens [90]. The FIFO holds data tokens in the order they have received them and provides access to data tokens using a first-in-first-out access policy. It also isolates the execution boundaries which enables concurrent execution of producer and consumer actors.

There are different possible data passing patterns among dataflow nodes depending on the number of producer and consumer nodes directly connected. Figure 3.3 illustrates multiple actor (many-to-one, one-to-many and many-to-many) data passing patterns [91], [92], [93]. They can further drive other patterns such as (merge-pipeline-split) to implement tree reduction and expansion dataflow graphs. A *split* and *merge* can express by *Single-Producer-Multiple-Consumers* (SPMC) and *Multiple-Producers-Single-Consumer* (MPSC) in a producer-consumer model, or used to implement *data parallel* computation. Similarly, a *feed-forward* can represent by *single-producer-single-consumer* (SPSC) in producer-consumer model, or used to achieve *pipelining* or *task parallel* computation. Since these patterns are reusable, different nested data passing patterns can be derived such as *merge-pipeline-split* or *split-pipeline-merge* as shown in Figure 3.3.

To support fine and coarse-grained mapping and execution of dataflow applications onto the proposed *FPGA-based hardware acceleration platform* requires these data exchange patterns between dataflow actors to be supported by the multicore IPPro. It would enable application exploration, profiling and optimisation opportunities which are discussed in Chapter 5 and 6.

3.5 Summary

This chapter outlined the scope and programming workflow of *Rathlin* project to differentiate and characterise the novelty of the presented research work beyond *FPGA-based hardware acceleration platform* architecture itself. Since the platform architecture is one of the parts of the project, some of the architectural choices have been driven by the programming workflow. It shows the need of:

- Hardware and software abstraction to design and develop an adaptable FPGA-based hardware architecture to efficiently utilise them using sound software engineering principles. This will be discussed in Chapter 4 by developing FPGA-based soft-core processor architecture.

- Supporting of familiar software driven *edit-compile-run* flow would facilitate profiling, optimisation and fast prototyping of different image processing applications by avoiding synthesis, place-and-route times.

- Supporting dataflow model of computation as one of the data processing models of IPPro and multicore IPPro architecture. Both architectures shall allow flexible mapping and granular execution of CAL algorithmic description. Architectural details of these dataflow semantics and data passing patterns will be discussed in Chapter 4 and 5.

- Supporting of parallel computing skeletons (split-compute-merge, farm, pipeline) to efficiently map high-level domain specific parallel descriptions on the platform. Architectural details will be discussed in Chapter 6.

Chapter 4

Image Processing Processor (IPPro)

4.1 Introduction

The integration of the hardware and software has made system-on-chip (SoC) architectures such as Xilinx Zynq-7000 SoC, suitable as a computing platform to meet the computing demands of wide range of applications. In these architectures, the FPGA programmable logic is tightly-coupled with a general-purpose processor which can be efficiently used by off-loading compute intensive tasks. Nevertheless, the changing technology landscape and fast evolution of new application use cases make it imperative that the underlying hardware architecture shall be adaptable. Silicon vendors have alleviated this issue by introducing high-level programming tools such as Xilinx Vivado high-level synthesis (HLS). While this raises the level of abstraction, a part of the FPGA design tool-flow still requires lengthy FPGA synthesis and place-and-route times [10], [37], [44], [45] which are alien to software and algorithm developers.

This research aims to propose a programmable approach that replaces the specialised HDL-based hardware accelerator design, to software like recompilation of FPGA resources. It can achieve by populating the underlying FPGA architecture with multiple light-weight, high-performance soft-core processors. The user applications are compiled and mapped onto the soft-core processors as a binary code rather than a FPGA bit-stream. It avoids synthesis and place-and-route and provides software developers with the familiar *edit-compile-run* flow which reduces design time and effort. Compared to the HDL approach, it will be straight-forward, easy to debug/profile and enable better application optimisation possibilities. The first step to realise this approach is to develop an efficient, light-weight, programmable processing node in the form of soft-core processor tailored to accelerate image pre-processing functions. Following are the novelties and contributions of this chapter:

- Exploration of dataflow and FPGA-based soft-core datapath models to identify the best balance among dataflow graph mapping possibilities, processor datapath functionalities and performance. The outcome laid down the architectural design choices for a high-performance and area efficient FPGA-based soft-core processor architecture.

- A novel FPGA-based soft-core Image Processing Processor (IPPro) architecture tailored to accelerate image pre-processing applications. The architecture provides a balance between efficient utilisation of FPGA resources and performance while enabling deployment of *edit-compile-run* design flow. These features make IPPro suitable to be used as a basic computational unit of many and multicore hardware acceleration architecture.

- Optimisation of the IPPro datapath to support additional instructions. It include coprocessor extension and dedicated minimum/maximum instructions to improve hardware acceleration results. The optimised datapath supports parallel execution of variable latency custom coprocessors.
- Acceleration of chosen point and area-based pre-processing image processing functions on an Avnet Zedboard using single core IPPro. The results of the proposed adaptable hardware acceleration approach are compared against two programmable approaches including well-established soft-core processor and a fixed high-level synthesis approach.

In this chapter, Section 4.2 presents the most suitable class of image pre-processing algorithms considering minimal data-dependency and efficient utilisation of FPGA dedicated hardware resources. Section 4.3 presents a detailed evaluation of different dataflow and soft-core processor models to find the best balance between dataflow mapping possibilities and achievable performance. Section 4.4 introduces a novel FPGA-based soft-core *Image Processing Processor* (IPPro) architecture tailored to accelerate point and area image processing operations. Section 4.5 covers the IPPro datapath optimisations supporting dedicated instructions and coprocessor extension to off-load instructions that are computationally expensive to implement using native IPPro instruction set. At the end of this chapter, acceleration of chosen point and area image processing functions are accelerated using proposed approach and compared against hand-coded HLS implementation. In addition, two comparison against programmable FPGA-based approaches including a well-established MicroBlaze soft-core processor.

4.2 Algorithmic characteristics of image processing algorithms

In an image processing pipeline, each stage depending on intended use may have predominant tasks and corresponding pre-processing operations [94], [95], [96]. They operate at the beginning of a processing pipeline and therefore, computationally data intensive due to heavy pixel processing which makes them a suitable candidate for hardware acceleration. Table 4.1 shows the categorisation of these pre-processing image operations, where each class has distinctive data dependency, memory access and execution pattern. These algorithmic characteristics provide the functional hardware requirements that shall be supported by the *Image Processing Processor* (IPPro) architecture to accelerate these class of image pre-processing applications.

To achieve improved acceleration and efficient utilisation of FPGAs compute and memory resources, it is crucial to select the most suitable class of image processing operations. FPGA delivers the best performance for streaming applications due to spatial locality and minimal data dependency which are common in *point* and *area* image processing operations. They require basic arithmetic, logic

Table 4.1: Categorisation of image processing operations based on their memory and execution patterns [65].

Operation type	**Output depends on**	**Memory Pattern**	**Execution Pattern**	**Examples**
Point	Single input pixel	Pipelined	One-one	Intensity change by factor, Negative image inversion
Area/Local	Neighbouring pixels	Coalesced	Tree	Convolution functions: Sobel, Sharpen, Emboss, Morphology
Geometric	Whole frame	Recursive non-coalesced	Large reduction tree	Rotate, Scale, Translate, Reflect, Perspective and Affine

and condition operations which can be efficiently implemented using FPGA logic which makes *point* and *area* image pre-processing operations suitable for hardware acceleration. Also, the development of high performance and area-efficient soft-core processor requires analysis of functional configurations of the dedicated DSP and memory blocks as they impact the maximum achievable operating frequency f_{Max} and balance between FPGA compute and memory resources. Moreover, identification of soft-core processor design choices needs detailed datapath analysis. Because optimising the design for one design goals very often reduces the possibility of achieving some of the others.

4.3 Exploration of efficient FPGA soft-core processor

In FPGA fabric, dedicated DSP and memory blocks are hardware optimised computation and memory blocks. Image processing applications extensively use multiply and accumulate operation for image segmentation and filtering tasks which can efficiently map to the DSP block. The dedicated memory blocks are placed next to the DSP blocks to minimise timing penalty. Despite the fact that FPGA has these optimised hardware blocks, the maximum operating frequency (f_{Max}) of a design depends on the length of the critical path. In case of soft-core processors, f_{Max} represents the raw-computation rate of the processor. It is one of the reasons that current many and multicore architectures use simple, light-weight processing datapaths over complex and large out-of-order processors. However, to maintain balance among soft processor functionality, scalability, performance and efficient utilisation of FPGA resources remains an open challenge.

4.3.1 Balance between compute and memory resources

The goal is to build a soft-core processor that implements arithmetic and logic functions by exploiting DSP and memory blocks where computing is defined as the raw performance of a soft-core processor is expressed by f_{Max} . Therefore, this section evaluates different configurations of Xilinx DSP block (DSP48E1) and Block RAM (BRAM), and their impact on f_{Max} using different FPGAs. It has six configurations that offer different functionalities (multiplier, accumulator, pre-adder and pattern detector) based on different internal pipeline configurations of DSP48E1 [52]. Therefore each configuration directly impacts the f_{Max} of DSP48E1 (central to realise high-performance processor architecture). The Xilinx Vivado Design Suite v2015.2 is used for each DSP48E1 configuration and obtained f_{Max} trend is reported in Figure 4.1.

It can be observed that a drastic variation of $\approx$ 15 - 52% has recorded for same speed-grade and reduction of $\approx$ 12 - 20% when the same design has ported from -3 to -1 speed grade. This analysis shows that the configuration of DSP48E1 block significantly impacts the f_{Max} and identifying the optimum configuration is essential. Therefore, a fully pipelined DSP48E1 block with a pattern detector **PATDET** configuration is selected as it gives fully pipelined multiply, accumulate, add, subtract and pattern detector functionality with minimal f_{Max} penalty of $\approx$ 12% compared to fully pipelined DSP48E1 block without a pattern detector **NOPAT**. The built-in pattern detector allows implementation of condition statement and execution of data dependent instructions which are commonly found in image processing functions. The presented results in Figure 4.1 shows that soft-core processor could run at 627, 549, 463 MHz for speed grade -3, -2, -1 respectively, if DSP48E1 is used as an ALU.

To analyse the impact of dedicated memory resources on f_{Max} , BRAM is configured as single and true-dual port RAM [97]. Figure 4.2 shows the f_{Max} trend across Artix-7, Kintex-7 and Virtex-7 to analyse the impact across different FPGA fabrics. The true-dual port RAM configuration result f_{Max} reduction of $\approx$ 25%. On the other hand, improvement of $\approx$ 16% is possible by migrating the design from Artix-7 to Kintex-7 FPGA technology. FPGAs are limited in memory and for efficient design, it is vital to find the balance between memory and performance. Table 4.2 shows the distribution of compute (DSP48E1)

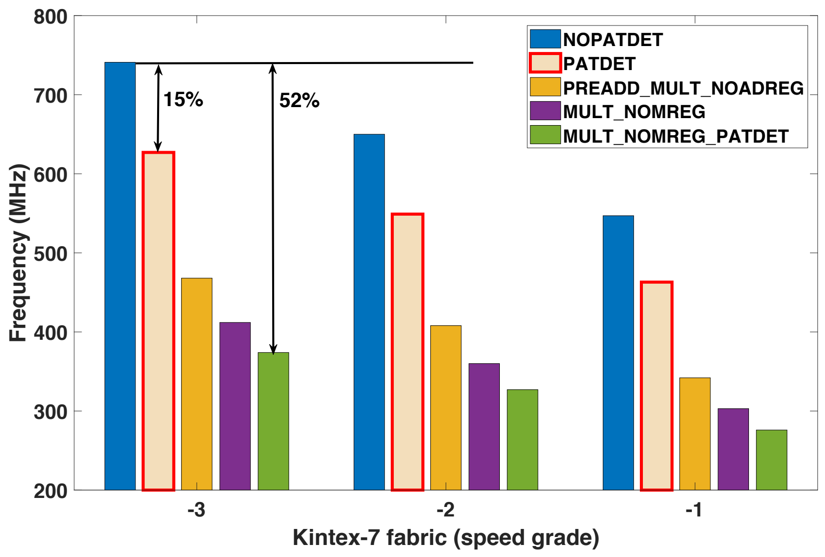


Figure 4.1: Impact of DSP48E1 configurations on maximum achievable clock frequency (f_{Max}) using different speed grades of Kintex-7 FPGAs. The DSP48E1 configuration used are: fully pipelined datapath with no pattern detector (NOPAT), with pattern detector (PATDET), multiply with no output register MREG (MULT_NOMREG) and pattern detector (MULT_NOMREG_PATDET) and a Multiply, pre-adder, no ADREG (PREADD_MULT_NOADREG).

Table 4.2: Memory and compute resources in 28nm Xilinx FPGA technology [98].

Product	Family	Part Number	BRAM (18 Kb each)	DSP48E1	GMAC/s	BRAM/ DSP
Standalone	Artix-7	XC7A200T	730	740	929	0.99
Standalone	Kintex-7	XC7K480T	1,910	1,920	2,845	0.99
Standalone	Virtex-7	XC7VX980T	3,000	3,600	5,335	0.83
Zynq SoC	Artix-7	XC7Z020	280	220	276	1.27
Zynq SoC	Kintex-7	XC7Z045	1,090	900	1,334	1.21

and memory (BRAM) resources, and present raw performance in GMAC/s (giga multiply-accumulates per second) across the largest FPGA devices covering both standalone and Zynq MPSoC chips [98]. A new metric BRAM/DSP ratio is introduced to quantify the balance between compute and memory resource and reported in Table 4.2. In Zynq MPSoC devices, the BRAM/DSP ratio is higher than standalone devices because more memory is required to implement substantial data buffers to exchange data between FPGA fabric and the host processor

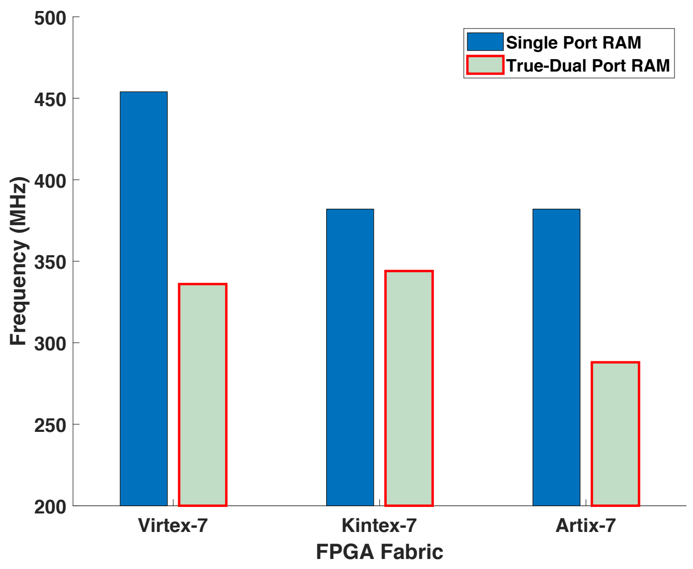


Figure 4.2: Impact of BRAM configurations on the maximum achievable clock frequency (f_{Max}) of Artix-7, Kintex-7 and Virtex-7 FPGAs for single and true-dual port RAM configurations.

while it is close to unity for standalone devices. This comparison shows that BRAM/DSP ratio can be used to quantify the area efficiency of FPGA designs.

4.3.2 FPGA-based soft-core processor functionality vs performance trade-off

A system composed of light-weight and high-performance soft-core processor architecture that supports modular computation with fine and coarse-grained functional granularity is more feasible than fixed dedicated hardware accelerators. A light-weight soft processor shall allow populating more programmable hardware accelerators onto a single MPSoC chip which would lead to better acceleration possibilities by exploiting data and task level parallelism.

Evaluation of processor functionality and dataflow models

This section presents the a design exploration approach to analyse and evaluate functional granularity of FPGA-based soft-core datapaths while correlating each model with their realistic dataflow model. Table 4.3 lists three models driven by previous work [35], [99] which functionally corresponds to soft-core datapath models ①, ② and ③. These models are used to find a trade-off between the functionality of soft-core processor and f_{Max} . They also laid the foundation to find the suitable soft-core datapath to map and execute the dataflow specification. Gupta *et al.* have reported different dataflow graph models [99], as illustrated in Figure 4.3. The input/output interfaces are marked in red while, the grey box represents the mapped functionality onto the datapath models shown in Figure 4.4.

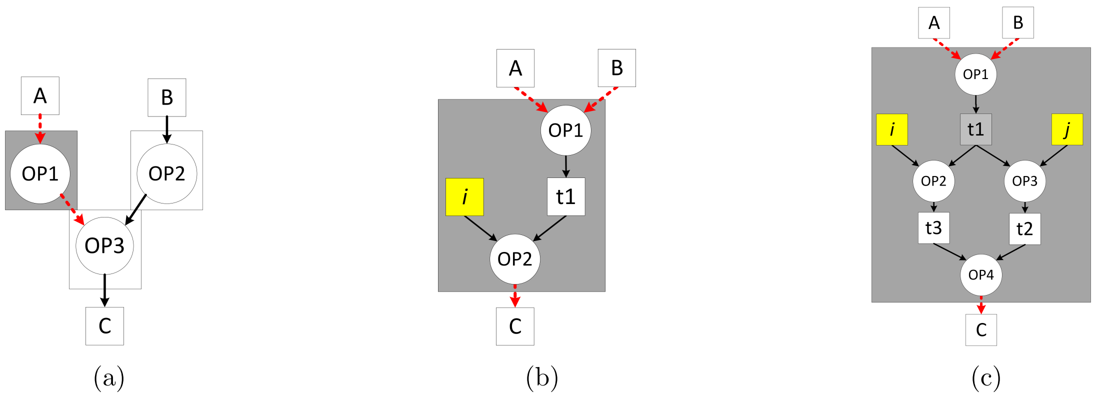


Figure 4.3: Dataflow models [35], [99] (a) DFG node without internal storage ① (b) DFG actor without internal storage $t1$ and constant i ② (c) Programmable DFG actor with internal storage $t1, t2$ and $t3$ and constants i and j ③.

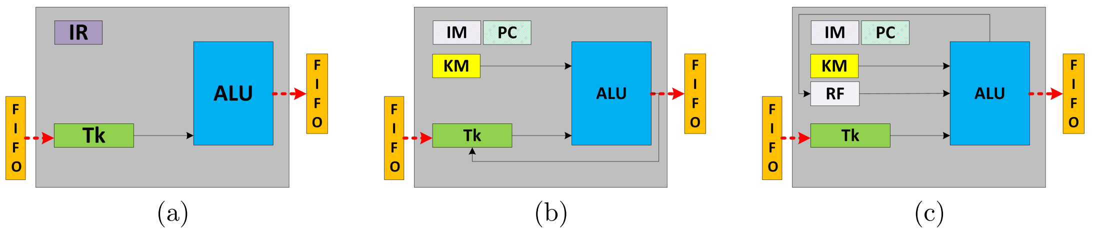


Figure 4.4: FPGA datapath models (a) Programmable ALU ① (b) Fine-grained processor ② (c) Coarse-grained processor ③.

The first model ① exhibits datapath of a programmable ALU as shown in Figure 4.4(a). It has *instruction register* (IR) that defines a DFG node (OP1) programmed at system initialisation. At each clock cycle, the datapath explicitly reads a token from the input FIFO, process token based on the programmed operation and stores into the output FIFO that are consumed by the following dataflow node (OP3). This model only allows mapping of data independent fine-grained dataflow nodes as shown in Figure 4.3(a) which limits its applicability due to lack of control and data dependent execution commonly found in image processing applications where the output pixel depends on the input or neigh-

Table 4.3: Correlation of FPGA-based soft-core datapath and dataflow models with increasing functionality and memory.

Model#	**Datapath model**	**Dataflow model**
①	Programmable ALU	Programmable node without memory
②	Fine-grained processor	Programmable actor without memory
③	Coarse-grained processor	Programmable actor with memory

bouring pixels. Table 4.4 list specific dataflow features supported by ①. This model is only suitable for mapping a single dataflow node.

The second model ② increases the datapath functionality to a fine-grained processor by including BRAM-based *instruction memory* (IM), *program counter* PC and *kernel memory* (KM) to store constants as shown in Figure 4.4(b). Conversely, ② can support mapping of multiple data independent dataflow nodes as shown in Figure 4.3(b). The node (OP2) requires a memory storage to store variable (t1) to compute output token (C) which feeds back from the output of the ALU needed for next instruction in the following clock cycle. This model supports improved dataflow mapping functionality over ① by introducing IM which comes at the cost of variable execution time and throughput proportional to the number of instructions required to implement the dataflow actor. Table 4.4 list supported dataflow features of ②. This model is suitable for accelerating combinational logic computations.

Table 4.4: Details of supported dataflow features and processor datapath memory elements in each presented model.

Model	**Dataflow features**					**Datapath memory elements**		
	Dataflow	***Control flow***	***Node mapping***	***Execution pattern***	***Token P/C***	***IM***	***KM***	***RF***
①	✓	✗	Single node	Feed-forward	Fixed	✗	✗	✗
②	✓	✗	Multiple nodes	Feed-forward	Fixed	✓	✓	✗
③	✓	✓	Multiple actors	Feed-forward, split, merge, feedback	Variable	✓	✓	✓

The third model ③ increases the datapath functionality to map and execute data dependent dataflow actor as shown in Figure 4.3(c). The datapath has memory element as *register file* (RF) which represents a coarse-grained processor shown in Figure 4.4(c). The RF stores intermediate results to execute data dependent operations, implements (feed-forward, split, merge and feedback) dataflow execution patterns and facilitates dataflow transformations (actor fusion/fission, pipelining *etc.*) constraints by the size of RF. It can implement modular computations which are not possible in ① and ②. In contrast to ① and ②, the token production/consumption (P/C) rate of ③ can be controlled through soft-core program code as listed in Table 4.4 and allow software controlled scheduling and load balancing possibilities.

Functionality vs Performance trade-off analysis

The presented models show that the processor datapath functionality significantly impacts the dataflow decomposition, mapping and optimisation possibilities, but at the same time increases the processor critical path length and affects f_{Max} by incorporating more memory elements and control logic. Table 4.4 lists the datapath memory elements in each presented model by incrementally allocating more memory resource (IM, KM, RF). Each presented model has coded in Verilog HDL, synthesised and place-and-route using Xilinx Vivado Design Suite v2015.2 on the Xilinx chips installed on widely available development kits which are Artix-7 (Zedboard), Kintex-7 (ZC706) and Virtex-7 (VC707). The obtained f_{Max} results are reported in Figure 4.5.

In this analysis, f_{Max} is considered as a performance metric for each processor datapath model. The implementation result shows that increasing datapath

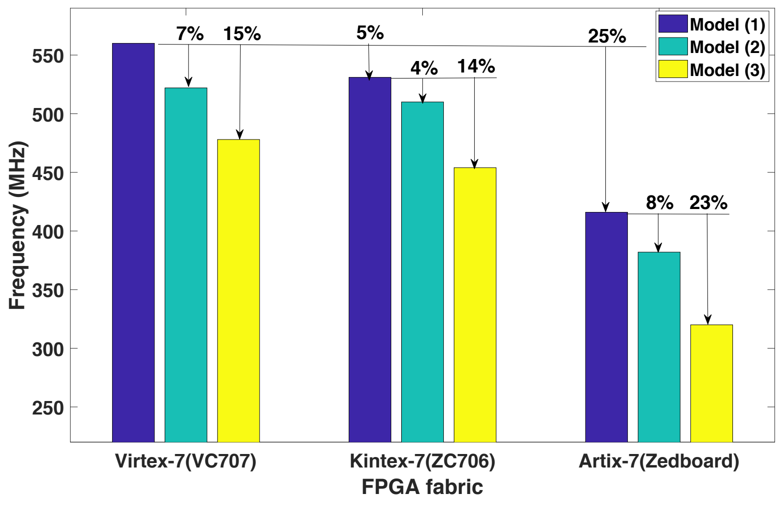


Figure 4.5: Impact of datapath models ①, ②, ③ on f_{Max} across FPGA fabrics.

functionality resulted in a reduction of f_{Max} by a maximum of $\approx 8\%$ and 23% for ② and ③ compared to ① using same FPGA technology. For ②, the addition of memory elements specifically IM realised using dedicated BRAM affected f_{Max} by $\approx 8\%$ compared to ①. Nevertheless, the instruction decoder (ID) which is a combinational part of a datapath significantly increases the critical path length of the design. A further 15% f_{Max} degradation from ② to ③ has resulted by adding memory elements KM and RF to support control and data dependent execution, which requires additional control logic and data multiplexers. Comparing across different FPGA fabrics, f_{Max} reduction of $\approx 14\%$ and 23% is observed for Kintex-7 and Artix-7. When ③ is ported from Virtex-7 to Kintex-7 and Artix-7, maximum f_{Max} reduction of $\approx 5\%$ and 33% is observed.

This analysis has laid firm foundations by comparing different processor datapath and dataflow models and how they impact the raw computation rate (f_{Max}) of the resultant soft processor. The trade-off analysis shows that an area-efficient, high-performance softcore processor architecture can be realised that supports requirements to accelerate image pre-processing applications. Among the presented models, ③ provides the best balance among functionality, flexibility, dataflow mapping and optimisation possibilities, and performance. This model is used to develop a novel IPPro architecture in Section 4.4.

4.4 Image Processing Processor (IPPro)

This section presents the novel Image Processing Processor (IPPro) datapath by mapping it onto FPGA resources. Image pre-processing functions requires grey-level image where the value of pixel represents the colour contrast. For specific functions, *e.g.* image filtering that involves multiply and multiply-accumulate operations, it is essential to maintain precision. Therefore, IPPro designed as 16-bit, signed, *reduced instruction set* (RISC), pipelined soft-core architecture shown in Figure 4.6.

The IPPro datapath exploits DSP48E1 and BRAM blocks and supports stream processing using blocking input/output FIFOs that handle a stream of pixels. On the contrary to out-of-order processor architectures, IPPro is designed as a five-stage, in-order pipelined processor because: 1) It consumes fewer area resources and can achieve better timing closure leading to the higher processor operating frequency f_{Max} . 2) The in-order pipeline execution is predictable and simplifies scheduling and compiler development. In fact, the area hungry out-of-order processor architectures are suitable for ASIC or custom designs where chip resource are not technologically bounded. Based on the exploration of processor datapath and dataflow models and evaluation of their functionality and performance trade-off analysis presented in Section 4.3, following memory areas are supported

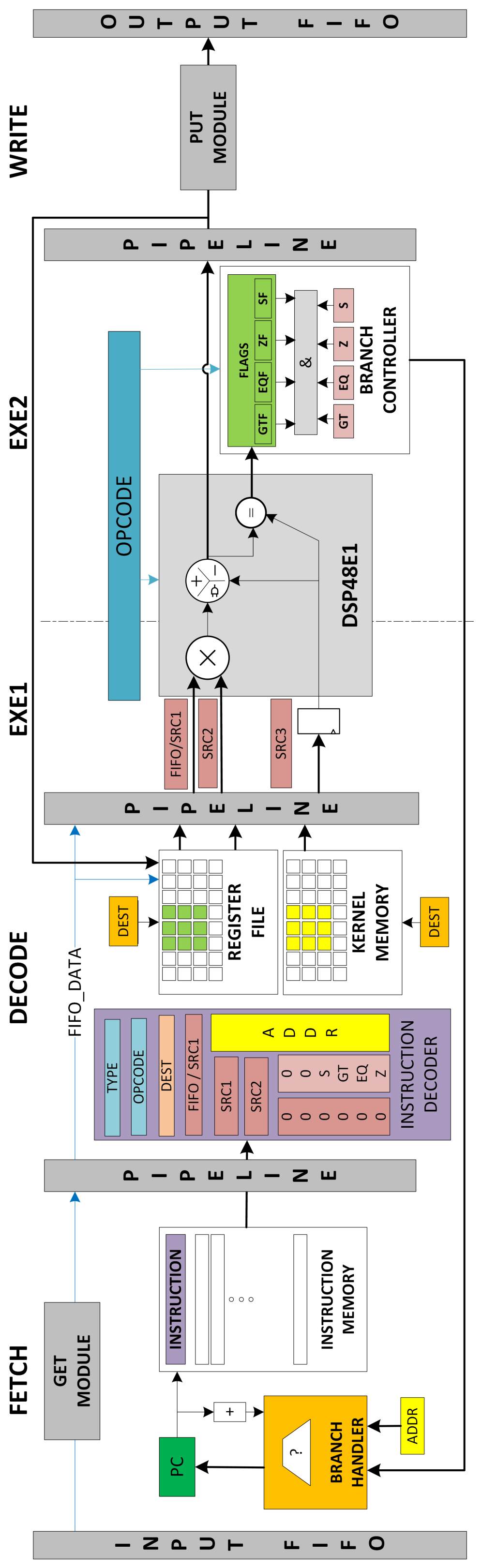


Figure 4.6: Block diagram of FPGA-based soft-core processor IPPro datapath.

in the IPPro datapath:

- Instruction memory (IM) (512x32) to store the dataflow actor functional description in the form of IPPro program code.
- Register file (RF) (32x16) to map fine and coarse-grained dataflow actors by storing intermediate results and provide random access to a stream of tokens or window of pixels stored inside the RF, *e.g.* 3x3, 3x4, 4x4 *etc.*
- Kernel memory (KM) (32x16) to save the parameters that are reusable such as filter coefficients and constant values.
- The blocking input/output FIFOs to buffer data tokens between a producer and a consumer to realise pipelined processing stages.

4.4.1 Datapath

RISC architecture performs computation on register values in contrast to stack-based complex instruction set (CISC) architecture. RISC-based architectures have faster memory access to the registers which involves random access to variables rather than access of stacked operands [100]. Therefore, a *Register file* (RF) of size 32x16 bits is implemented using Xilinx RAM32M primitive that uses look-up tables (LUT) resources. It provides a quad-port RAM with synchronous write and three asynchronous read ports compared to dual-port RAM supported by BRAM primitive. It supports three operand operations such as multiply-add commonly used for pixel processing. Figure 4.6 shows the detailed IPPro datapath. It has BRAM-based instruction memory (IM) configured as true dual-port RAM which stores program code. IPPro has a dedicated KM that can store

32x16 bit constant values to accelerate *area* operations by maximising memory reuse and avoid reloading of filter coefficients. The input FIFO stores the incoming stream of data, the GET instruction reads and stores them in the RF. $PUSH$ instruction reads the processed data from specified RF location and stores it in the output FIFO.

The IPPro datapath has no stack memory and therefore, does not support recursive function call as it requires context switching (which involves passing of parameters between functions and storing the function state/variable). But as long as the memory requirement of the calling function (number of critical function variables) matches the size of the register file, limited recursive function call can be implemented using the branch instructions (JUMP and BZ). From image processing perspective, the IPPro datapath has been designed to implement *point* and *area* image processing operations which only require neighbouring pixels and can be stored in the register file.

4.4.2 Branch and conditional execution

IPPro supports branch instructions to handle control flow graphs previously discussed in Table 4.4 to implement commonly known constructs such as *if-else* and *case* statements. The DSP48E1 block has a pattern detector that compares the input operands or the generated output results depending on the configuration and sets/resets the PATTERNDETECT (PD) bit. IPPro datapath uses the PD bit along with some additional control logic to generate four flag *zero (ZF), equal (EQF), greater than (GTF) and sign (SF)* bits. When IPPro encounters branch instruction, the *branch controller* (BC) compares the flag status and *branch han-*

Table 4.5: IPPro instruction frame structure.

BITS						
31 30	**29 25**	**24 20**	**19 15**	**14 10**	**9 5**	**4 0**
$INSTR\_TYPE$	$OPCODE$	R_D	R_B/K_n	R_A	R_C	0

dler (BH) updates the PC as shown in Figure 4.6.

4.4.3 Instruction set architecture

IPPro has a 32-bit instruction set architecture (ISA). Table 4.5 shows the simplified IPPro frame structure where R_A, R_B, R_C, R_D and K_n represents 5-bit address fields to point a location in RF or KM. R_A, R_B, R_C, K_n are source registers while R_D is a destination register. The 5-bit $OPCODE$ field represents a unique IPPro instruction. The 2-bit $INSTR\_TYPE$ field differentiates between supported addressing modes listed in Table 4.6. (for details on supported instruction set see Appendix B Table B.2).

4.4.4 Pipelined stream processing

The IPPro datapath is a five stage pipeline soft-core processor composed of *fetch, decode , execute#1 (EXE1), execute#2 (EXE2)* and *write-back (WB)* stages as shown in Figure 4.6. It starts execution by fetching the instruction from the *instruction memory*, the *instruction decoder* decodes the fetched instruction and generates required control signals to control the datapath. During this stage

Table 4.6: IPPro supported addressing modes and instructions.

Addressing Mode	**Data abstraction**	**Instructions**
FIFO handling	Stream access	get, push
RF - RF	Randomly accessed data	str, add, sub, mul, mulacc, muladd *etc.*
KM - FIFO	Stream and fixed data	addkm, subkm, mulkm, muladdkm, *etc.*

based on addressing mode (Table 4.6), IPPro read data operands either from input FIFO, RF or KM and stores into the pipeline registers and forwards to DSP48E1 block in EXE1 stage. The DSP48E1 is dynamically reconfigured on a cycle-to-cycle basis by ID. The configuration of DSP48E1 control signals to implement IPPro instructions (for details see Appendix B Table B.1). The DSP48E1 processes the data operands and store the results back to the register file in WB stage. Both EXE1 and EXE2 are DSP48E1 internal pipeline stages. The GET and PUSH modules shown in Figure 4.6 makes sure that input/output FIFOs are not empty/full. If any of the conditions persist, the IPPro stop processing and waits until both input and output FIFO have enough space to store the tokens.

4.4.5 Dataforwarding

Data hazards are common in pipelined processors, IPPro supports internal data forwarding by exploiting multiply-accumulate (MACC) feature of DSP48E1 to avoid pipeline stalls and NOP fillers. During instruction decoding, the datapath checks if the source address of the next instruction is equal to the destination address of the decoded instruction. If it is true, the dataforwarding path is

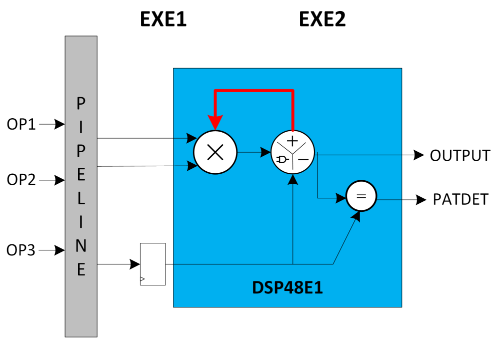


Figure 4.7: Implementation of dataforwarding exploiting MACC functionality of DSP48E1.

enabled by configuring a DSP48E1 control register and the result of DSP48E1 is forward to the next instruction as shown in Figure 4.7.

To demonstrate the impact of dataforwarding on execution time (clock cycles) and the program code size, consider the following equation, and the corresponding IPPro code listed in Table 4.7.

$$A = func(z - (x + y) + (y * z)) \tag{4.1}$$

This function requires three data dependent computations as listed in Table 4.7. In case of no dataforwarding, the NOP fillers are required to avoid data hazards due to lack of available data independent instructions which can fill the pipeline. On the other hand, in case of dataforwarding, the computed data can be forwarded directly to the next instruction as highlighted by blue and red in Table 4.7. The IPPro processor with and without dataforwarding takes 18 and 10 clock cycles respectively to process the function. Mathematically, it can be represented by:

Table 4.7: IPPro code to implement *func* with and without dataforwarding.

Instr.	No Data Forwarding	Description	Data Forwarding	Description
1	GET R1	R1=x	GET R1	R1=x
2	GET R2	R2=y	GET R2	R2=y
3	GET R9	R9=z	GET R9	R9=z
4	NOP		NOP	
5	NOP		NOP	
6	NOP		NOP	
7	ADD R3,R1,R2	R1+R2	ADD R3,R1,R2	R1+R2
8	NOP		SUB R4,R3,R9	(R1+R2) - R9
9	NOP		MULACC R5,R4,R2	(R9*R2)+(R1+R2)-R9
10	NOP		PUSH R5	
11	NOP			
12	SUB R4,R9,R3	(R1+R2)-R9		
13	NOP			
14	NOP			
15	NOP			
16	NOP			
17	MULADD R3,R4,R2,R9	(R9*R2)+(R1+R2)-R9		
18	PUSH R10			

Table 4.8: IPPro implementation results on selected Xilinx development boards.

Resources	**VC707**	**ZC706**	**Zedboard**
FFs		447	
LUTs		484	
BRAMs		1	
DSP48E1		1	
Freq. (MHz)	372	337	187

$$t = (n - 1) * 4 \tag{4.2}$$

Where n the number of consecutive data dependent instructions and t is the saved number of clock cycles per iteration. The impact of dataforwarding become significant when processing images consist of hundreds of thousands of pixels. Saving tens of clock cycles per pixel results in a significant saving of processing time. Nevertheless, it also reduces the code size which is $\approx 45\%$ for the presented case.

4.4.6 Implementation results

IPPro soft-core processor architecture has written in Verilog, synthesised and implemented using Xilinx Vivado Design Suite v2015.2. Table 4.8 reports the implementation results obtained using tool's default settings. The implementation results show that IPPro consumes $< 1\%$ of Kintex-7 (ZC706) FPGA resources and delivers 337 MIPS while maintaining BRAM/DSP ratio equal to unity. The IPPro design has ported to various FPGA fabrics to analyse the potential performance, by implementing it on widely available Xilinx development boards used by research community which are ZedBoard (XC7Z020CLG484-1), ZC706 (XC7Z045FFG900-2) and VC707 (XC7VX485T-2). Table 4.8 shows the maximum possible frequency f_{Max} on the selected Xilinx development boards.

IPPro running on Virtex-7 (VC707) and Kintex-7 (ZC706) can deliver $\approx$ 2.00 and 1.80 times improved f_{Max} compared to Artix-7 (Zedboard) by porting on different FPGA fabric. The obtained results closely correspond to the results reported in Table. 4.5 where it was expected to be $\approx$ 19% and 48% for ZC706 and Zedboard respectively in Section 4.3.

4.5 IPPro Optimisations

To evaluate performance and identify limitations of the developed IPPro, two group students have accelerated colour, morphology [101] and two-stages of the *histogram of gradient* [102]. Russell *et al.* have reported 9.6 times performance improvement for morphology operations using native IPPro instructions compared to ARM processor-based implementation. He identified that supporting dedicated *minimum* and *maximum* instructions will improve performance. Kelly *et al.* have profiled and explicitly translated the first two stages of HOG algorithm from mathematical expressions to native IPPro instructions. He reported that 77.3% of the total instructions belong to the normalise overlapping spatial blocks function, out of which 72.2% of the IPPro instructions belong to the division calculation. He indicated that *division* function is the computational bottleneck and off-loading division from IPPro to dedicated coprocessor could significantly improve the acceleration results. To this end, this section presents IPPro optimisations by extending IPPro datapath capabilities beyond DSP48E1 supported instructions to enhance the performance further.

4.5.1 Minimum and maximum instructions

In image processing applications, morphological operations are applied to the filtered image to clean up small holes in objects and remove small groups of pixels which saves processing time for later stages. Morphology involves finding either the maximum (dilation) or minimum (erosion) value in a set of pixels contained within a masked region around the input pixel. Russel *et al.* have reported that implementation using native IPPro instructions takes $\approx$ 48 cycles for a 3x3 kernel or 81 cycles for a 5x5 kernel. To include dedicated *MIN* and *MAX* instruction, the additional control logic and a 4-1 multiplexer to select the minimum or maximum result are added into the datapath as shown in Figure 4.8. The *MIN* and *MAX* registers externally hold the operand values. The DSP48E1 block compares the operands and updates the sign flag (SF), which is used to select either MIN/MAX value and store it into the RF.

Table 4.9 shows the IPPro code to compare the impact of optimised MIN/-MAX on the execution time. It shows that native implementation first compares the operands using subtraction followed by branch evaluation to find the minimum and maximum value which takes $\approx$ 13 - 20 clock cycles per pixel depending

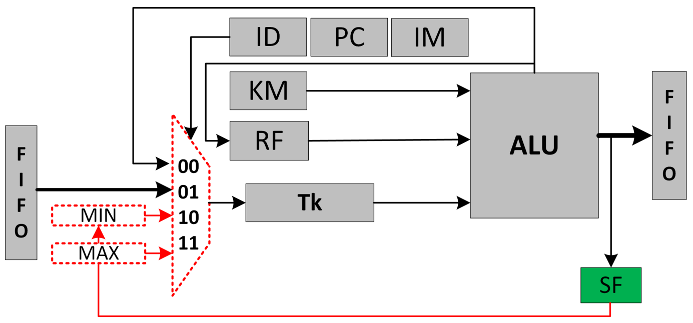


Figure 4.8: Optimisation of IPPro datapath to support dedicated minimum and maximum instructions.

Table 4.9: Implementation of Min/Max using native and optimised IPPro instructions.

Instr.	Native	Description	Dedicated	Description
	FUNCTION:		**FUNCTION:**	
1	GET R1	R1=a	GET R1	R1=a
2	GET R2	R2=b	GET R2	R2=b
3	NOP		NOP	
4	NOP		NOP	
5	NOP		NOP	
6	NOP		NOP	
7	SUB R3,R2,R1	R3 = a-b ? +ve/-ve	MIN R3, R2, R1	R3 = min(a,b)
8	BS MAX		MAX R4, R2, R1	R4 = max(a,b)
9	NOP		PUSH R3	
10	NOP		JUMP FUNCTION	
11	NOP			
12	NOP			
13	PUSH R2	send minimum value		
14	JUMP FUNCTION			
15	...			
16	...			
17	...			
18	**MAX:**			
19	PUSH R1	send maximum value		
20	JUMP FUNCTION			

on whether the branch has taken or not. On the other hand, optimised implementation takes ten clock cycles per pixel irrespective of pixel value resulting approx. 50% reduction in execution time which is significant for pixel processing.

4.5.2 Coprocessor extension

In image processing, some of the algorithms require arithmetic operations which are not supported by the IPPro. For such applications, IPPro has a coprocessor interface that allows a transparent integration of a custom coprocessor into the IPPro datapath. In this section, the example of a division coprocessor will be discussed as Kelly *et al.* have reported that it is appropriate to off-load computationally expensive functions to a coprocessor which adds complexity to the processor architecture. In case of IPPro, it is adding a coprocessor interface into the datapath and balancing the pipelined execution while dispatching the operands, collecting the processed results and storing them into the RF such that coprocessor shall execute in parallel and not stall the IPPro to achieve best possible improvement.

Figure 4.9(a) shows the block diagram of the pipelined division coprocessor and Figure 4.9(b) shows the IPPro coprocessor extension datapath. Four 16-bit registers (C_IR1, C_IR2, C_OR1 and C_OR2) are incorporated between input/output interface of coprocessor and the IPPro datapath. The coprocessor enable signal (C_ENABLE) is asserted by instruction decoder once the IPPro encounters the dedicated coprocessor instruction and writes the input operands to C_IR1 and C_IR2 registers. These registers isolate coprocessor and IPPro datapath and ensure transparent exchange of data and independent parallel execution of coprocessor and IPPro. The coprocessor process input operands and stores result into the output registers (C_OR1 and C_OR2). The IPPro reads the coprocessor generated results from these output registers by executing a particular coprocessor read instruction and stores them into the RF as illustrated in Figure 4.9(b).

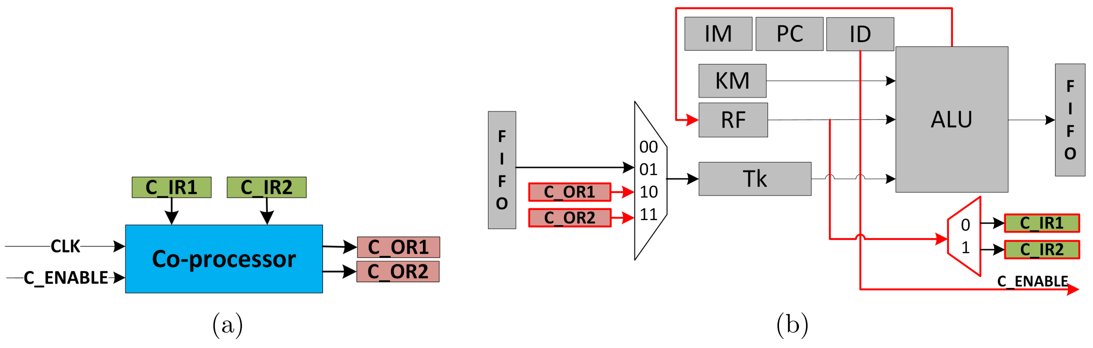


Figure 4.9: (a) Input/output interfaces of division coprocessor (b) Coprocessor extended IPPro datapath.

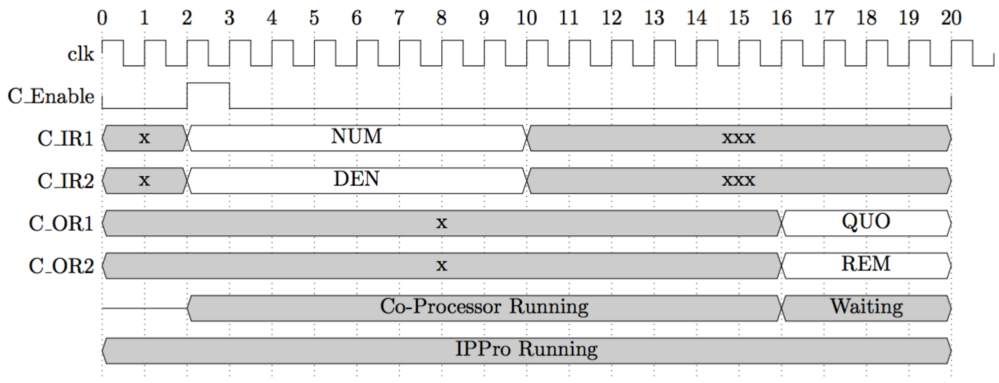


Figure 4.10: Pipelined execution of division coprocessor.

A division coprocessor has been incorporated into the extended IPPro datapath to evaluate the coprocessor extension. The division coprocessor takes two input operands (*numerator* and *denominator*) and generates (*quotient* and *reminder*), which mapped into IPPro datapath via (C_IR1, C_IR2, C_OR1 and C_OR2) registers respectively as shown in Figure 4.9(a). In this implementation, the coprocessor clock (CLK) is synchronised to the IPPro datapath. Figure 4.10 shows the timing diagram of parallel execution of IPPro and the division coprocessor. The operands (C_IR1, C_IR2) are exchanged and they become valid once *C_Enable* is asserted. The coprocessor takes a fixed number of clock cycles to process input data, generate results and store them into output registers. These processed tokens are then collected using the process described earlier.

The coprocessor extended datapath has implemented using Xilinx Vivado De-

Table 4.10: Implementation results of optimised IPPro datapath to support coprocessor extension on ZC706 (Kintex-7).

Resources	**Standalone**	**Coprocessor extension**
FFs	447	481
LUTs	484	573
BRAMs	1	1
DSP48E1	1	1
Freq. (MHz)	**337**	**302**

sign Suite v2015.2. Table 4.10 shows the area and performance results. f_{Max} degradation of 11% is observed compared to standalone IPPro caused by the addition of multiplexers to feed operands to coprocessor and store generated results back into the register file. Kelly *et al.* have reported that the division coprocessor implementation reduced the instruction count for the division from 160 to 19 instructions which caused 82% reduction in the normalisation function [103]. This saving attributed to the introduction of the coprocessor at the cost of 89 LUTs, 34 FFs per core and 11% reduction in f_{Max} as shown in Table 4.10. This solution allows the IPPro core to execute in parallel with the coprocessor giving the scheduler considerable freedom to organise the fine-grained tasks of the algorithm efficiently.

4.6 Comparison of IPPro results

Kapre *et al.* have proposed GraphSoC, a custom soft processor for accelerating graph algorithms using Zynq MPSoC [16]. It is a 3-stage pipelined processor that supports graph semantics (node, edge operations). The graphs were stored in on-chip BRAM for fast local access. A compilation framework developed including assembler to configure the processor instruction and data memories where each core uses 9 BRAMs and operates at 200 MHz. Andryc *et al.* presented an FPGA-based FlexGrip architecture for compute-intensive streaming applications [36]. It is composed of an array of streaming multiprocessors (SMs), each SM contains multiple 5-stage pipelined scalar processor (SP) cores connected in a SIMD computing paradigm. The framework maps pre-compiled CUDA kernels on SP that operates at 100 MHz.

Table 4.11: Comparison of IPPro against other FPGA-based soft-core processor architectures.

Resources	**IPPro**	**GraphSoC** [16]	**FlexGrip 8 SP** [36]	**MicroBlaze**
FFs	447	551	12,972=103,776/8	518
LUTs	484	974	8,915=71,323/8	897
BRAMs	1	9	15=120/8	4
DSP48E1s	1	1	19=156/8	3
No .of Stage	5	3	5	5
BRAM/DSP ratio	1.0	9.0	0.76	1.3
Freq. (MHz)	**337**	**200**	**100**	**211**

* Scaled to a single streaming processor.

Table 4.11 compares the implementation results of IPPro processor against other processors. The reported area utilisation results of FlexGrip is normalised to single processing core as each SP is composed of 8 cores connected in SIMD. The results show that IPPro is compact and delivers ≈ 1.6x - 3.3x times better performance, considering f_{Max} . The reported area results show that the FFs utilisation is relatively similar except FlexGrip uses 18 times more FFs. While comparing LUTs, IPPro uses 50% fewer LUT resources compared to both MicroBlaze and GraphSoC. Analysing design area efficiency, a significant difference 0.76 - 9.00 in BRAM/DSP ratio is observed which makes IPPro an area-efficient design-based on the proposed metric.

4.7 Application use cases

Two different comparison approaches are adopted to evaluate the area and performance of IPPro architecture by comparing it against HLS, programmable FPGA-based architecture and softcore processor. Firstly, a set of chosen *point* and *area* operations image pre-processing functions are implemented using IPPro and compared against the hand-coded HLS implementations. Secondly, the chosen image pre-processing functions will be compared against programmable FPGA-based

Table 4.12: Mathematical representation of image pre-processing functions.

Function	**Mathematical representation**
Thresholding	$P_{(output)} = P_{input} > P_{threshold} ? 255 : 0$
Gaussian	$P_{(output)} = \begin{bmatrix} P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 \\ P_7 & P_8 & P_9 \end{bmatrix} * \begin{bmatrix} K_1 & K_2 & K_3 \\ K_4 & K_5 & K_6 \\ K_7 & K_8 & K_9 \end{bmatrix} = \sum_{i=1}^{9}(P_i * K_i)$
Sobel	$P_{(output)} = \begin{bmatrix} P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 \\ P_7 & P_8 & P_9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$
Gradient calculation	$P_{Gradient} = \lvert P_x \rvert + \lvert P_y \rvert$

architecture and lastly, a set of micro-benchmarks are selected to analyse the IPPro performance against well established MicroBlaze softcore processor.

Image pre-processing functions Image pre-processing algorithms are used extensively for feature detection, image analysis and noise reduction which includes image filtering functions [94], [95], [96]. *Convolution* operation is central to filtering algorithms that use *area* image processing operations as previously identified in Table 4.1. On the other hand *point* operations are commonly used for image segmentation. Functions from both classes, i.e. (thresholding, Gaussian, Sobel and gradient calculation) are accelerated using IPPro processor to evaluate the performance of IPPro architecture. These functions are commonly used front-end image processing operations [104], [105], [106], [107], [108]. Table 4.12 presents the mathematical representation of chosen functions. These operations are accelerated by developing a system that implements real-time video pipeline composed of a camera and VGA output. The obtained acceleration results are compared against fixed HLS approach.

Processor Micro-benchmarks The performance of a processor can be measured in many ways, often it is reported in *millions-instructions-per-second* (MIPS). Though it is not always a good metric as one processor may accomplish more work than an instruction on another processor by issuing a single instruction or negatively impact the performance due to the branch penalty. One of the commonly used performance metric is the time required to accomplish a defined task. Therefore, a set of commonly used micro-benchmarks [86], [109] have been chosen and implemented on the IPPro and the obtained results are compared against well established MicroBlaze soft-core processor. Each of the chosen micro-benchmarks are fundamental kernels of larger algorithms and often the core computation of more extensive practical applications. The following are the details of each chosen micro-benchmark and the architecture aspects tested by each:

- Digital filter is an important function that signal processors use to modify and improve signals. In image processing they are used to improve the appearance of an image by smoothing, blur and removing noise. It allows to analyse 1D stream processing capabilities of the IPPro architecture. The implementation of **5-tap FIR** function reads an element from an input stream, computes FIR and push the output to the FIFO for 50 samples.
- **Convolution** is a stream processing micro-benchmark extensively used in the image processing. It allows to analyse 2D data processing capability of a processor. For the IPPro architecture, it would help to analyse the impact of single cycle MULACC optimisation. The implemented micro-benchmark

reads 3 data elements per iteration, computes the convolution function and push output to the FIFO.

- **Polynomials** are one of the most fundamental types of functions generally used in mathematics as well as in image processing to realise non-linear filters used for contrast enhancement, texture segmentation and edge extraction. Usually they are formed entirely by repeated multiplications and addition. For the IPPro architecture, it allows to analyse the impact of dataforwarding optimisation. The implementation of **degree-2 polynomial** function reads an element from an input stream, computes $y(x) = ax^2 + bx + c$ and push output to the FIFO.

- **Matrix multiply** is a widely used operation in digital signal processing applications and its non-linear complexity is often the critical part of many algorithms. It is computational expensive as it requires extensive data independent multiplications and data dependent additions. This micro-benchmark allows to analyse the computation capability (MULACC) and the memory limitations of the IPPro architecture. The implementation of matrix multiply function reads two matrices from the register file, computes the product and stores the resultant matrix into the local memory.

- In digital image processing, the **Sum Of Absolute Differences** (SAD) is a measure of the similarity between image blocks. It calculates by taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison. It is used for object recognition, disparity map and motion estimation. This micro-benchmark allows to analyse the impact of branch operations necessary to compute the absolute value. The implementation of SAD function reads a window of elements stored in local memory, computes the absolute difference and pushes the results to the output FIFO.

- **Fibonacci** sequence requires adding of the two preceding numbers to generate the output number which makes it extensively data dependent computation. It allows to analyse the impact of both the data dependent execution and the branch penalty on the IPPro architecture. The implementation of Fibonacci function calculates first 50 numbers of the series and pushes into the output FIFO.

Section 4.7.1 presents the system architecture used to accelerate the chosen image pre-processing operations.

4.7.1 System architecture

The system architecture is composed of OV7670 camera (to capture real-time video stream), single core IPPro (to process the incoming video stream) and VGA output (to display processed video stream). Figure 4.11 shows the developed system architecture used to accelerate the chosen *point* and *area* operations by feeding pixel or window of pixels configured during system initialisation. This system architecture is implemented and tested on Avnet Zedboard development board has an on-board Xilinx Zynq SoC (XC7Z020-CLG484-1). The Zynq heterogeneous MPSoC has on-chip *programmable system* (PS) tightly-coupled with *programmable logic* (PL). The AXI-AMBA communication protocol is supported between PS and PL. The AXI-Lite interface is used to program the IPPro instruction memory, and control register during system configuration.

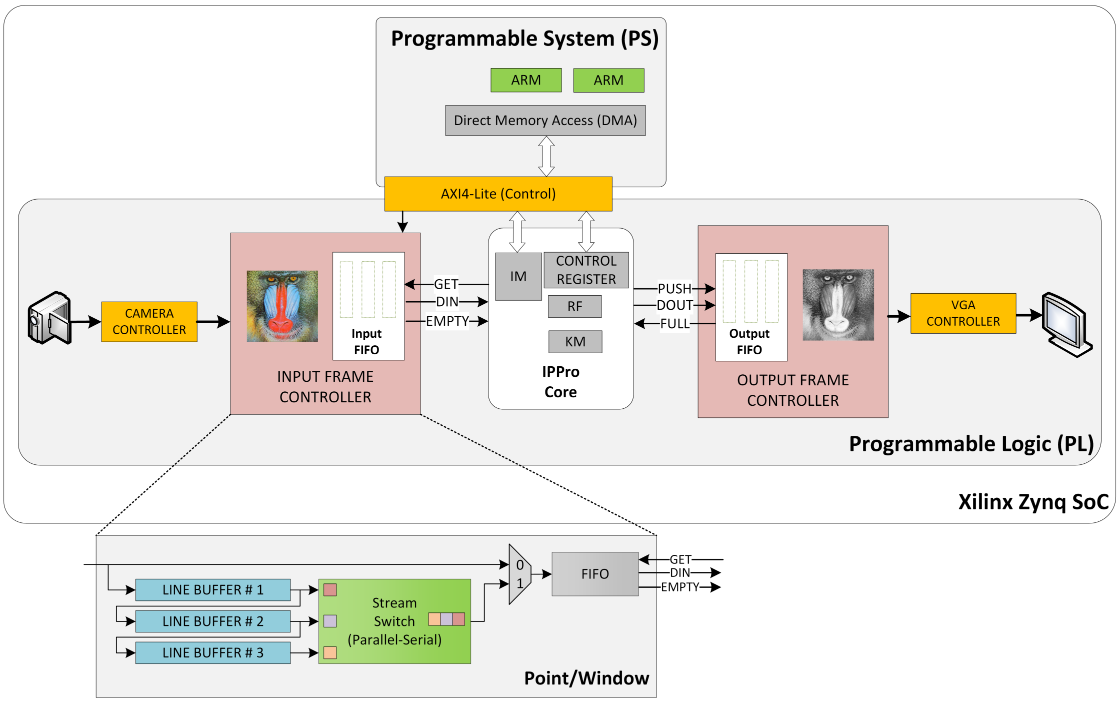


Figure 4.11: Block diagram of programmable video processing platform to implement case-studies using single-core IPPro.

The OV7670 is a CMOS colour image sensor that supports configurable VGA and CIF video resolutions, and RGB 565/555, YUV(4:2:2) and YCbCr(4:2:2) pixel formats. The camera module is directly connected to the Zynq PL using PMOD-A and PMOD-B interface on Zedboard. The VGA resolution and YUV(4:2:2) pixel format is selected where (Y) grey-scale component is used to accelerate the chosen front-end image processing operations. A dedicated *camera controller* handles camera initialisation sequence and configurations using I^2C protocol. It captures the incoming video stream and stores them into input frame buffer. The *input frame controller* sequentially reads the video frame (starting from address $0 \rightarrow 307200$) from the input frame buffer and converts it into a stream of pixels based on configured point or window (using line buffers) then

Table 4.13: Area utilisation results of IPPro hardware accelerator.

Module	**Resources**			
	FFs	LUTs	BRAM	DSP
Datapath	695	815	1	1
Point/area	349	275	3	0
Total	**1044**	**1090**	**4**	**1**

store into the input FIFO as shown in Figure 4.11.

The input FIFO isolates the camera and IPPro clock domains which allows IPPro to run at higher operating frequency 187 MHz than the camera interface. It also provides handshaking mechanism, to propagate the ripple effect and halts the input frame controller to avoid data corruption when IPPro executes an unbalanced actor. As soon as the pixels are available in the input FIFO, IPPro reads the stream of pixels, sequentially processes and store them into the output FIFO. The *output frame controller* reads the processed pixels and converts into video frame by sequentially storing pixels (starting from address $0 \rightarrow 307200$). The *VGA controller* reads the processed video frame, generates the required VGA control signals (V-SYNC and H-SYNC) to display it on the VGA monitor.

Table 4.13 reports the implementation results of datapath and point/area module. The reported datapath is composed of the necessary control logic composed of AXI-Lite control registers. The point/area module composed of line buffers to organise video data into a *point* or *window* of pixels. The point/area module uses three BRAMs to implement three line buffers required to generate 3x3 pixel window as reported in Table 4.13. Additionally, to support AXI4-Lite control and configuration register, IPPro datapath consumes 1.55 and 1.68 times more FFs and LUTs compared to the reported results in Table 4.10.

4.7.2 Comparison of IPPro with HLS approach

The acceleration results of the proposed IPPro-based programmable approach are compared against high-level synthesis (HLS) approach. The chosen image processing operations are hand-coded in C++ and compiled using Xilinx Vivado HLS. The implementations exploit pipeline optimisation and the designs are synthesised and implemented using Xilinx Vivado Design Suite v 2015.2. The Vivado HLS has generated each operation as intellectual property (IP) which has AXI4-Lite and AXI4-Stream interfaces for easy integration into the previously presented system architecture in Figure 4.11. In system architecture, the IPPro core is replaced with Vivado generated IP.

The HLS implementations achieved 28 and 15 times better than IPPro due to higher computation rate *(MPixel/s)* as reported in Table 4.14 at the cost of software-centric edit-compile-run design flow. In case of IPPro, the computation rate *MPixels/s* is inversely proportional to *cycles/pixel* which depends on the complexity of the function. Therefore, further comparison of the proposed IPPro-based programmable approach against other programmable FPGA-based architecture is presented and analysed in Section 4.7.3.

Table 4.14: Comparison of hardware acceleration results obtained from HLS and IPPro using Avnet Zedboard (Artix-7).

Acceleration approach	**Dedicated accel.**			**Proposed IPPro**			
Performance results	*fps*	*MPixel/s*	*Freq. (MHz)*	*fps*	*cycles/pixel*	*MPixel/s*	*Freq. (MHz)*
Thresholding	651	200	200	17	42	5.22	187
Gaussian (3x3)	488	150	150	35	20	10.80	187
Sobel (3x3)	488	150	150	43	16	13.20	187
Gradient calculation	651	200	200	24	32	7.37	187

4.7.3 Comparison of IPPro against programmable FPGA-based architecture

Reichenbach *et al.* have presented a programmable image processing architecture for smart cameras [110]. The architecture is based on programmable coarse grained application specific processing elements (PE) that enables fine-grained configurability to realise algorithmic peculiarities of image processing applications. Each PE only supports a set of application-specific assembly instructions that can be used to compute that specific image processing function such as Gaussian, Sobel and Gradient operations. The architecture had been implemented on heterogeneous Xilinx Zynq XC7Z020 SoC platform where the programmable logic is used to populate the PEs and process video frames.

Table 4.15: Comparison of IPPro performance results against programmable FPGA-based architecture.

Function	**[110]**			**IPPro**	**Speed-up**
	# of cores	fps	fps/core	fps	
Gaussian	12	295	24	46	1.87
Sobel	6	180	30	54	1.80
Gradient	20	120	6	35	5.83

To compare the performance and resource utilisation results of this architecture against IPPro, the performance and resource utilisation numbers have been reported in Table 4.15 and Table 4.16 has been normalised to single-core. Focusing on area utilisation numbers, the PE implementing a Sobel filter consumed 2.8 and 2.6 times less FFs and LUTs respectively than a Gaussian by exploiting kernel coefficient optimisation. IPPro has achieved 5.8 and 1.8 times better performance at the cost of approximately equal number of FFs and 1.5 times less LUT resources over [110] for gradient calculation and Sobel filter respectively. This performance improvement at reduced area cost by IPPro architecture has

Table 4.16: Area comparison of IPPro against programmable FPGA-based architecture. The normalised per core resource utilisation are reported in the brackets.

Resources	[110]			**IPPro**
	Gaussian	Sobel	Gradient	
FFs	6177 (1029)	4360 (363)	592 (30)	1044
LUTs	10017 (1669)	7718 (643)	1782 (90)	1090
BRAMs	2	2	2	4
DSPs	-	-	-	1

been achieved by exploiting DSP block optimisation over [110] which can be clearly observed in Table 4.16.

4.7.4 Comparison of IPPro with MicroBlaze

The selected micro-benchmark results are compared against well established Xilinx MicroBlaze soft-core processor. The micro-benchmarks are written in standard C and implemented using Xilinx Vivado SDK v2015.1. MicroBlaze has been configured for performance with no debug module, instruction/data cache and single AXI-Stream link enabled to stream data into the MicroBlaze using *getfsl* and *putfsl* instructions in C which are equivalent to (get and put) in assembly.

Table. 4.17 reports the performance results of micro-benchmarks implemented using IPPro and MicroBlaze soft-core processors using Kintex-7 FPGA fabric. Table. 4.18 shows the area utilisation of proposed IPPro and MicroBlaze soft-core

Table 4.17: Comparison of micro-benchmarks on IPPro and MicroBlaze.

Processor	**MicroBlaze**	**IPPro**	
FPGA Fabric	Kintex-7		
Freq (MHz)	287	337	
Micro-benchmarks	**Exec. Time (us)**		**Speed-up**
Convolution	0.60	0.14	4.41
Degree-2 Polynomial	5.92	3.29	1.80
5-tap FIR	47.73	5.34	8.94
Matrix Multiply	0.67	0.10	6.7
Sum of Absolute Differences	0.73	0.77	0.95
Fibonacci	4.70	3.56	1.32

Table 4.18: Area comparison of IPPro and MicroBlaze processors.

Processor	MicroBlaze	IPPro	Ratio
FFs	746	422	1.77
LUTs	1114	478	2.33
BRAMs	4	2	2.67
DSP48E1	0	1	0.00

processors. IPPro consumes ≈ 1.7 and 2.3 times fewer FFs and LUTs respectively. It can be observed that for streaming functions (3x3 filter, 5-tap FIR and Degree-2 Polynomial), IPPro has achieved 1.80, 4.41 and 8.94 times better performance compared to MicroBlaze due to support of single cycle multiply-accumulate with dataforwarding and get/push instructions in IPPro processor. However, as IPPro datapath does not support branch prediction that impacts IPPro performance implementing data dependent or conditional functions (Fibonacci and Sum of absolute differences), where SAD implementation using IPPro resulted in 5% performance degradation compared to Microblaze. On the other hand for memory-bounded functions such as Matrix Multiplication, IPPro performed 6.7 times better than MicroBlaze due to higher operating frequency.

4.8 Summary

This chapter has presented a FPGA-based soft-core processor architecture to achieve programmable hardware acceleration of front-end image processing operations and compared the obtained performance and area results against fixed HLS design approach. The proposed approach has achieved software recompilation of FPGA by avoiding synthesis, place and route. It has achieved by developing a FPGA-based soft-core Image Processing Processor (IPPro) architecture tailored to accelerate front-end image processing operations. The architecture is developed after detailed insight analysis of FPGA resources, processor functionality and dataflow models. The architecture exploited FPGAs dedicated computing and memory resources to achieve best balance between performance f_{Max} and area utilisation.

The IPPro datapath supports is a 16-bit signed, 5-stage pipelined RISC processor that supports basic arithmetic, logical and branch instructions with dataforwarding to implement data dependent point and area operations. It is light-weight soft-core processor that consumes less than 1% of Kintex-7 (ZC706) FPGA fabric resources and delivers 337 MIPS. IPPro running on Virtex-7 (VC707) and Kintex-7 (ZC706) can deliver $\approx$ 2.00 and 1.80 times improved f_{Max} compared to Artix-7 (Zedboard) by porting IPPro to different FPGA fabric. The area and performance results make it viable to be used as basic processing element for programmable many and multicore architectures.

To evaluate the performance and identify limitations of the developed IPPro architecture, Russell and Kelly has accelerated *morphology filtering* and first two-stages of *histogram of gradient* (HOG) using native IPPro supported instructions. They reported that significant performance improvements by extending the datapath capabilities beyond supported instructions offered purely by the DSP48E1 block. Two IPPro optimisations are implemented which are; supporting MIN/-MAX instruction; and coprocessor extension which resulted in $\approx$ 82% reduction in IPPro instructions for HOG.

In the end, three comparison approaches are adopted to evaluate the performance and area of the IPPro architecture. The obtained results have compared against HLS, FPGA-based programmable architecture and well established MicroBlaze soft-core processor. The acceleration of point and area image pre-processing functions using HLS delivered significant performance compared to IPPro at the cost of programmability. IPPro has achieved 5.8 and 1.8 times better performance over FPGA-based programmable architecture that uses dedicated programmable processing elements by exploiting DSP block optimisation. On the other hand, IPPro delivered up to 8.94 times better performance, and 1.7 and 2.3 times fewer FFs and LUTs resources compared to MicroBlaze. Analysing the micro-benchmarks, IPPro has outperformed implementing data independent streaming functions due to the pipelined support of single-cycle multiply-accumulate operation and dataforwarding. For data dependent micro-benchmarks, reduction in performance is due to lack of branch prediction. Although IPPro delivered better performance and results than MicroBlaze, the results presented in this chapter uses a single-core IPPro. In Chapter 4 further investigation is carried out to explore performance improvement by exploiting data and task parallelism in streaming applications.

Chapter 5

IPPro-based acceleration of dataflow actor

5.1 Introduction

Chapter 4 presented the IPPro as a FPGA-based soft-core processor architecture to achieve programmable hardware acceleration of image pre-processing by exploiting the FPGAs dedicated computing and memory resources. This chapter extends this work by looking at the dataflow MoC and how it can effectively be used to accelerate dataflow actors by supporting it in the IPPro datapath. Initially, the chapter covers support of a dataflow actor at core-level focusing on firing actors, handling multi-port dataflow, the impact of FIFO implementation on the timing results (f_{Max}) and hardware constraints of mapping dataflow actor onto the IPPro core. It also present the benefits of the IPPro-based programmable approach over HLS. Then it focuses on a system architecture by integrating multiple IPPro accelerators to exploit dataflow parallelism. To evaluate the performance of discussed core and system level features, a detailed implementation of a k-means case study is presented, and compared against an equivalent implementation using an embedded CPU and GPU. The major contributions of this chapter are:

- Creation of an optimised IPPro core architecture which supports mapping and execution of static dataflow actor. The architecture is an independent, self-managed and area-efficient dataflow accelerator.
- Design and development of IPPro-based hardware accelerator models to analyse the management and provisioning policies of IPPro as a programmable dataflow accelerator and their impact on system design and control requirements to exploit parallelism.
- Design and implementation of a configurable system architecture that facilitates flexible decomposition and mapping of dataflow actors onto multiple IPPro cores using scatter-gather data distribution and a collection mechanism for image processing.
- Acceleration of *distance calculation* and *averaging* stages of the k-means clustering algorithm using four different IPPro accelerators exhibiting different actor-core mappings on an Avnet Zedboard. Performance, power, and resource efficiency have been compared against embedded CPU and GPU implementation.

Section 5.2 presents the IPPro core that supports dataflow components, execution patterns and stream-based producer-consumer model while maintaining a balance between area and performance. Section 5.3 explores different IPPro management and provisioning possibilities when incorporated in a heterogeneous system. It evaluates the impact on the host, inter-core communication and resource utilisation. Section 5.4 presents coarse and fine-grained mapping possibilities of dataflow actors onto multiple IPPro cores. It also presents a configurable system architecture tailored to accelerate image processing applications. Section 5.5 presents a case study acceleration of k-means clustering computing stages using IPPro accelerators. The solution uses data and task level parallelism by pipelining multiple stages. The results achieved with the IPPro accelerators are compared with the equivalent embedded CPU and GPU implementation in Table 5.15.

5.2 IPPro: A dataflow processor

A CAL dataflow application is a collection of computing units known as *actors*, which are composed of components, operations and memory elements as discussed in Section 2.2 and listed in Table 3.1 Figure 5.1(a) shows a CAL actor representation consisting of an action, state variables and a *finite-state-machine* (FSM). An actor exchange stream of tokens coming from unidirectional data buffers and starts execution as soon as the actor firing rule is satisfied. Once this happens, the actor reads token from the input buffer, processes it and stores it into the output buffer. It is essential that these functional requirements must be supported by the IPPro datapath to map and execute the dataflow actor. Table 5.1 lists one-to-one mapping of dataflow semantics onto the IPPro datapath.

The IM stores the functional description of a dataflow actor which contains the actor's description and its interaction with other actors, state variables and an

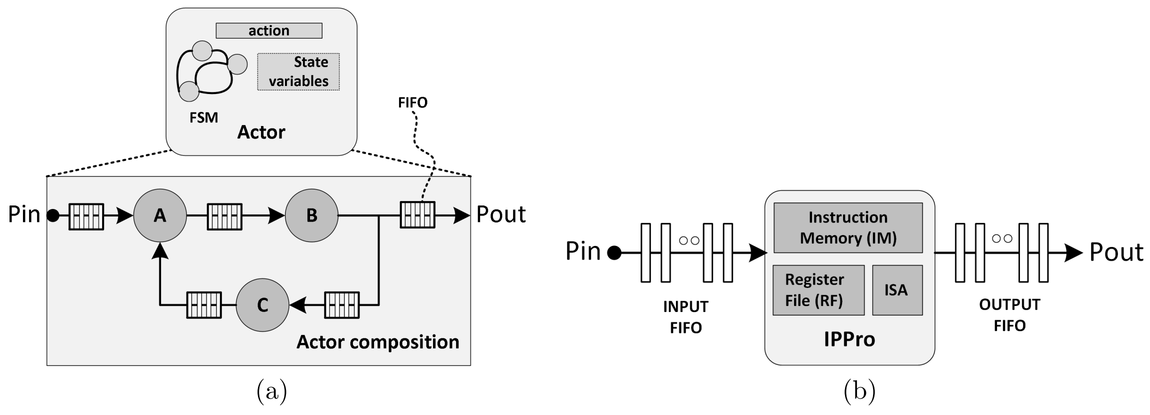


Figure 5.1: (a) Representation of a CAL dataflow actor (b) Mapping of dataflow actor onto IPPro datapath.

FSM which is stored in the form of IPPro program code. The IPPro instruction set architecture (ISA) implements the dataflow compute nodes defined within the action using arithmetic, logic and dedicated instructions *(MUL, MULACC, MULADD, MULSUB, MIN, MAX, ADD, SUB, etc.)*. The branch instructions *(BZ, BNZ, BS, BNS, etc.)* implements conditional, relational and data dependent nodes of the actor. RF is a memory element that stores state-variables, intermediate tokens and results of dependent nodes. One of the benefits of processor-based dataflow processing is *modularity*, as it allows fine and coarse-grained hierarchical decomposition and mapping of an actor onto IPPro core [30]. Figure 5.1(b) illustrates the mapping of an actor onto the IPPro datapath.

Section 5.2.1 presents the support of actor firing in the IPPro datapath while

Table 5.1: One-to-one mapping of dataflow semantics onto IPPro datapath.

No.	**Dataflow semantics**	**IPPro datapath (component)**	**Description**
1)	Actor	Instruction memory (IM)	Functionality of dataflow *actor*
2)	State variable	Register file (RF)	Stores intermediate data for data dependent node
3)	Operator node	Instruction set (ALU)	Arithmetic, logical and conditional operations
4)	Input buffer	Input FIFO	Stores input tokens
5)	Output buffer	Output FIFO	Stores output tokens

Section 5.2.2 extends it to support a data-driven computing model. It gives an analysis of realising FIFO's using different FPGA memory resources and their impact on the overall timing (f_{Max}) of the IPPro datapath. Section 5.2.4 presents the implementation of basic dataflow execution patterns using IPPro. Usually, dataflow actors support multiple data ports which are not feasible for IPPro architecture due to inefficient utilisation of FPGA resources which is covered in Section 5.2.5.

5.2.1 Notion of firing an actor

The notion of firing an actor is essential for functional correctness due to the un-timed behaviour of dataflow MoC. The token consumption and production rate depends on the functional description of an actor, and it is only known once the application use case has been chosen by the algorithm developer. Therefore, the IPPro must provide a flexible/programmable approach to handle actor firing and support a data-driven control mechanism to exchange data among actors. The initial IPPro datapath does not support the exchange of data tokens among multiple actors and is only suitable to map and execute an independent actor. It uses GET and $PUSH$ instructions to read and write data tokens.

To this end, an *actor firing* module and a $TEST$ instruction has been added into the IPPro datapath and instruction set as shown in Figure 5.2. The $TEST$ instruction allows the algorithm developer to specify the actor's consumption rate as a part of the actor firing and defined inside the IPPro program code. This instruction checks the number of tokens available for consumption by reading $TOKEN\_COUNT$ value of the input FIFO and comparing it with the expected

Table 5.2: IPPro code implementing dataflow actor firing rule.

#	Instructions		Description
1	**MAIN:**		*MAIN routine to check actor firing rule*
2	STR	R1, 4	*Set no. of tokens required to fire the actor*
3	TEST	R2, R1	*Check FIFO has more than 4(R1) tokens?*
4	BZ	FIRE_ACTOR	*If YES fire actor*
	..		
	..		
10	JMP	MAIN	*else wait until firing rule is satisfied!*
	..		
	..		
15	**FIRE_ACTOR:**		
	..		
	..		
30	JMP	MAIN	*The execution of actor is finished. Go back to MAIN to check firing rule again for next iteration*

consumption rate (passed as an argument with the instruction). The result of this comparison either grants or restricts the execution of the actor.

Table 5.2 presents IPPro code that implements the actor firing rule by initialising R1 (STR R1,4), where the value stored in R1 represents the actor's consumption rate. During program execution, the processor jumps between the $MAIN$ and $FIRE\_ACTOR$ sub-routines. In case, the input FIFO has four or more tokens, the program execution jumps to the $FIRE\_ACTOR$, executes a single iteration and returns to the $MAIN$ sub-routine. Otherwise, the program execution returns to the $MAIN$ and checks the firing rule. The $TEST$ instruc-

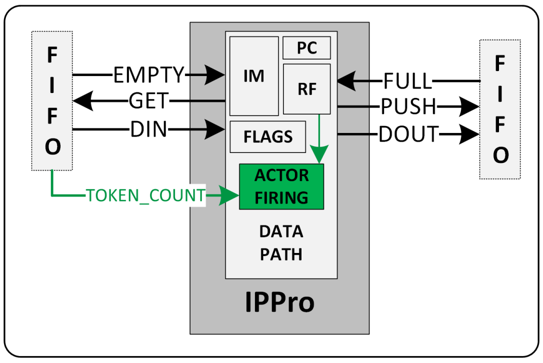


Figure 5.2: IPPro datapath supporting firing of dataflow actor.

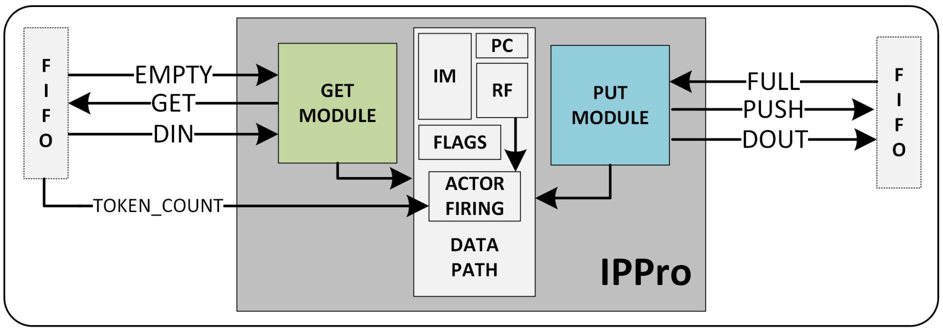


Figure 5.3: Producer-consumer data-driven execution using IPPro core.

tion allows programmable implementation of actor firing. It combines both actor's functional description and control (firing rule) in the IPPro code which avoids the need for an external controller synchronisation mechanism to implement dataflow actor and its code generation.

5.2.2 Producer-consumer computing model

This section discusses the problem of realising programmable multicore architecture using IPPro as basic programmable computation unit where some cores are producers and others are consumers requires control/handshake mechanism. These control mechanism ensures a continuous flow of data tokens between pipelined processing stages. The input and output FIFO provides isolation that could be used to exploit task level parallelism by minimising the maximum execution time of all stages of a pipeline and improves acceleration by keeping the cores busy in processing data.

For this purpose, dedicated PUT and GET hardware modules are included at the input and output data interfaces of the datapath as illustrated in Figure 5.3. They use *EMPTY* and *FULL* signals to check the status of a FIFO to identify whether input FIFO is empty or output FIFO is full. If this happens, the core stops execution and only resumes if both the output FIFO has empty space to store processed tokens and the input FIFO has tokens available for processing. Therefore, executing unbalanced actors, the slowest actor of an algorithm defines the worst-case execution time due to the ripple effect. However, there are different dataflow optimisations that could improve results by exploiting data parallelism and chosing a suitable decomposition [24], [30] which will be discussed in Section 5.4.

5.2.3 Evaluation of FIFO configurations

In an FPGA, FIFO can be realised using *Block RAM* (BRAM), *lookup-table* (DistRAM) or *shift register* (SR). BRAM is suitable for realising large FIFO structure similar to a line buffer that stores line of pixels, while DistRAM and SR are efficient for smaller FIFO realisation [111]. Realisation of FIFO using shift register exploits the LUT resources of a *configurable logic block* (CLB) as a shift register instead of a dual-port RAM. The CLB can be configured either as distributed 64-bit RAM or as 32-bit shift registers (SRL32) or as two 16-bit shift registers (SRL16). From a hardware perspective, FIFOs isolate processing elements running at different clock frequencies, hence are available in two configurations: *common-clock* (CC) or *independent-clock* (IC) depending on write and read clock sources. Thus, different FIFO configurations have been implemented on different FPGA fabrics using Xilinx Vivado v2015.2, and the results are reported in Figure 5.4.

Comparing common-clock (CC) implementations, the DistRAM delivers best f_{Max} followed by SR and BRAM where degradation of 8% and 17% have ob-

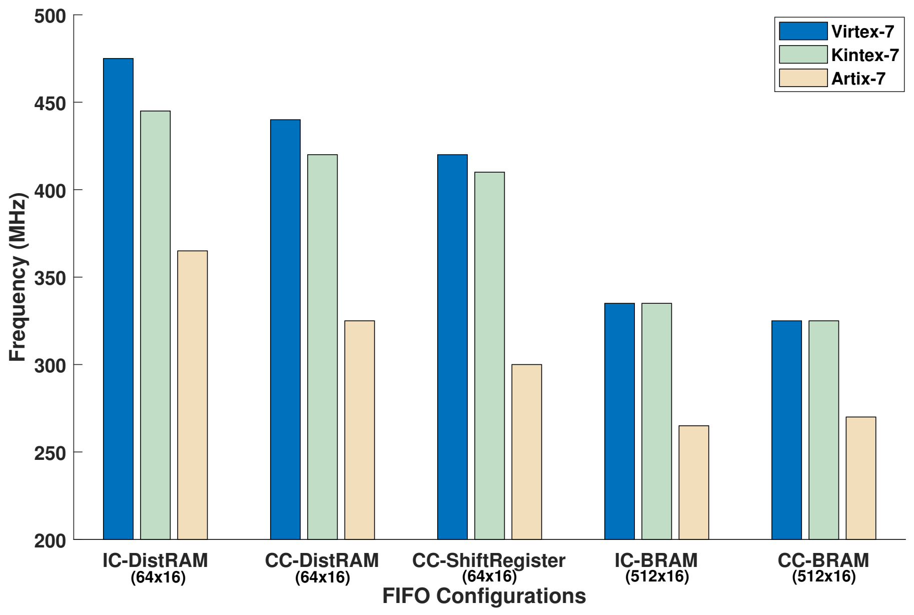


Figure 5.4: Impact on f_{Max} of realising FIFOs using different resources and configurations.

served on Artix-7 FPGA fabric. Moreover, comparing independent clock (IC) implementations, the DistRAM delivered best f_{Max} compared to BRAM which resulted in $\approx 27\%$ degradation. Realisation of FIFO using DistRAM is only feasible when deployed in the middle of the processing pipeline to store intermediate data tokens. On the other hand, BRAM-based FIFO are suitable and resource efficient for larger memory data structures such as line buffers (640, 1024, 2048, etc.). The result reported in Figure 5.4 shows the impact of FIFO configurations across FPGA technologies and can be used to find suitable FIFO configuration for the IPPro datapath.

For this purpose, the input and output FIFOs of the processor are realised using BRAM, DistRAM and SR. These designs are implemented using Xilinx Vivado v2015.2, and the area and timing results are reported in Table 5.3. In

Table 5.3: Implementation results of processor datapath using different FIFO configurations on Artix-7 FPGA fabric.

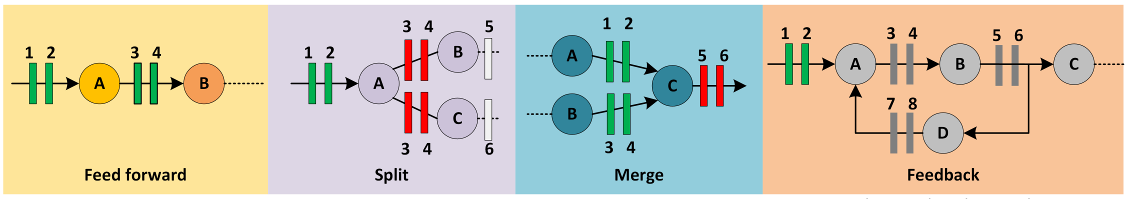
FIFO (size)	FF	LUT	LUTRAM	BRAM	DSP48E1	Frequency (MHz)
BRAM (512x16)	478	422	66	1.5	1	195
Shift Register (64x16)	510	411	90	1	1	237
DistRAM (64x16)	416	459	119	1	1	242

case of DistRAM, the processor datapath can operate up to 242 MHz giving a raw computation of 242 MIPS utilising 8% more LUTs compared to BRAM design. A reduction of 3% and 19% in processor operating frequency have observed for SR and BRAM designs respectively at the cost of 18% more FFs and 10% less LUTs. The presented processor datapath results show the impact of different FIFO configurations on timing and area utilisation. The design choice to realise FIFO depends on the deployment scenario and the application use case. DistRAM is efficient for small data buffers usually in the middle of an image processing pipeline. On the other hand, BRAM is resource efficient for large data buffer commonly found at the beginning or end of the image pipeline.

5.2.4 Mapping and execution of static dataflow actor

A static dataflow actor could represent a single operation node, a set of multiple operation nodes or a complex dataflow graph depending on the chosen decomposition. Each dataflow node can also have different execution patterns [90], [17], [25], [112]. These execution patterns include *feed-forward, split, merge and feedback* as illustrated in Figure 5.5 using dataflow nodes A, B, C and D. Figure 5.6 presents the pseudo IPPro program codes to implement each execution pattern using IP-Pro core.

In *feed-forward*, the GET reads the data tokens and stores them into $R1$



(R3,R4)=A(t1,t2)	(R3,R4)=A(R1,R2)	(R1,R2)=A(x)	(R3,R4)=A(R1,R2)
B=func(R3,R4)	R5=B(R3,R4)	(R3,R4)=B(y)	(R5,R6)=B(R3,R4)
	R6=C(R3,R4)	(R5,R6)=C(R1,R2,R3,R4)	C=func(R5,R6)
			(R7,R8)=D(R5,R6)

Figure 5.5: Mapping of dataflow execution patterns on IPPro core.

```
MAIN:
GET R1
GET R2
R3,R4=A(R1,R2)
PUSH R3
PUSH R4
JUMP MAIN
```

```
MAIN:
GET R1
GET R2
(R3,R4)=A(R1,R2)
R5=B(R3,R4)
PUSH R5
R6=C(R3,R4)
PUSH R6
JUMP MAIN
```

```
MAIN:
(R1,R2)=A()
(R3,R4)=B()
(R5,R6)=C(R1,R2,R3,R4)
PUSH R5
PUSH R6
JUMP MAIN
```

```
MAIN:
GET R1
GET R2
(R3,R4)=A(R1,R2,R7,R8)
(R5,R6)=B(R3,R4)
(R7,R8)=D(R5,R6)
PUSH R5
PUSH R6
JUMP MAIN
```

Figure 5.6: Pseudo IPPro code to implement dataflow execution patterns.

and $R2$ register of RF and executes function A, stores result into $R3$ and $R4$, and PUSH results to the output FIFO. In case of a *split*, the tokens produced by function A ($R3$ and $R4$) are fed to B and C. In case of a *merge*, A and B produce tokens ($R1$ and $R2$) and ($R3$ and $R4$) respectively which are fed to C that computes output tokens $R5$ and $R6$. The benefit of supporting these execution patterns with the help of RF in IPPro core is that, it not only allows implementation of a dataflow actor but also to provides flexible decomposition and mapping options to the user and software framework to explore and exploit dataflow optimisations.

5.2.5 Supporting multi-port dataflow actor

The dataflow programming languages support multi-port actors. The HLS driven hardware architectures support an input *interface* for each computation block, where each data-port is directly translated into a FIFO structure [43], [17]. However, the application use case or algorithm to be implemented is known in-advance before the hardware design is synthesised and implemented. It allows the HLS tool to profile and find optimal memory requirements for the chosen application. On the contrary, in a processor-based approach, the underlying hardware architecture is pre-implemented using generic processing and memory requirements of the class of applications. Because of this, the number of input/output interfaces supported by the IPPro datapath must be fixed. The higher number of ports could lead to inefficient utilisation of resources and small number of ports could limit the actor mapping possibilities. Thus, this section discusses this design problem by increasing number of ports and analysing their impact on resource requirements and the execution time of an actor.

Figure 5.7 depicts increasing input data interfaces to identify the architec-

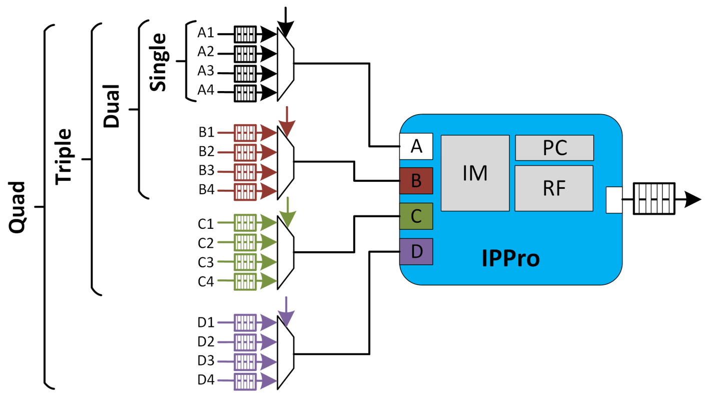


Figure 5.7: Block diagram of multi-port input data interface of IPPro datapath.

tural requirements and theoretically estimate their impact on the actor execution time. The datapath can be composed of *single*, *dual*, *triple* and *quad* input ports (A, B, C, and D). Each input port can receive tokens produced by different producers via ports A_n, B_n, C_n and D_n, where n distinguishes each unique producer. For functional correctness, the order of tokens is important. Therefore, a dedicated FIFO channel is required for each producer core to avoid token re-ordering problem.

Table 5.4 lists the architectural and control requirements for the input interface illustrated in Figure 5.7. A number of FIFO channels and multiplexers are required to connect the cores and receive data produced by the connected cores. It can be observed that the required number of FIFO channels are multiple of producers and input ports and the number of multiplexers required are directly proportional to the input ports. In FPGA design, a multiplexer is implemented using combinational logic which increases the critical path length of the design which affects the timing results. Therefore from both resource utilisation and timing point-of-view, a multi-port IPPro datapath is not a suitable design choice.

Figure 5.8 depicts cycle-based execution of $func(X)$ using *Single, dual, triple and quad* input ports. A *single port* design sequentially reads token from input FIFO compared to *dual*, *triple* and *quad* port designs. The DFG node processing time t_x (execution time of single iteration) is greater than time to read/write

Table 5.4: Hardware resource and control requirements to map multi-port actors onto IPPro core.

Resource	**Dataflow Actor**			
	Single-port	Dual-port	Triple-port	Quad-port
Producer cores	4	4	4	4
FIFO channels	4	8	12	16
No. of multiplexers	1	2	3	4
Source port addressing	No	1-bits	2-bits	3-bits

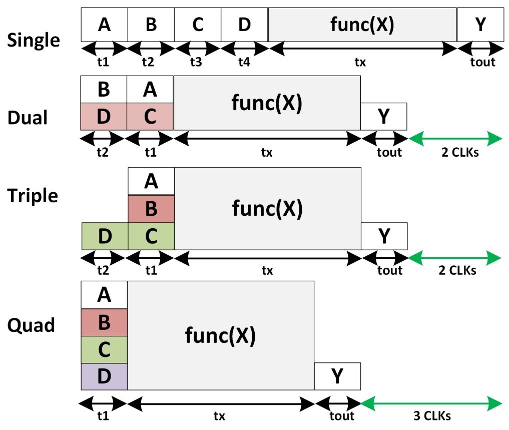


Figure 5.8: Impact of multi-port IPPro datapath on execution time (in clock cycles) of dataflow actor.

token $t1$, $t2$, $t3$, $t4$ and t_{out} (which is single clock cycle each) and has a negligible impact on the total execution time of an actor. In the best case scenario, the multi-port designs could save maximum of two or three clock-cycles as illustrated in Figure 5.8, at the cost of using more resources. Therefore, a single input port datapath is selected that can handle multiple operands using time multiplexing.

5.2.6 Discussion on hardware acceleration using IPPro over HLS

Usually, FPGA-based dataflow programming frameworks and HLS-based tools take a dataflow description, using static timing analysis techniques to profile and find a suitable decomposition that meets the application requirements. After finding proper decomposition, further FPGA/hardware specific optimisations are carried out and then equivalent HDL circuit is generated. On the contrary, in IPPro approach, the dataflow specification is statically profiled based on the IPPro mapping constraints. The following are major IPPro mapping constraints:

- *The number of instructions to implement a dataflow actor - $inst_{(actor)}$*. The IPPro has IM of 512x32 bit (because it efficiently exploits the distribution of BRAM resources by using exactly half of the BRAM block (18 KB) and allows a maximum of 512 instructions. This metric drives the level of decomposition of an actor. The framework must describe actor operations within 512 or less IPPro instructions.

- *Actor execution time - $t_{(exec.)}$* It is a measure of time needed for IPPro to execute a single iteration of an actor. f_{Max} is the IPPro system maximum clock frequency where each instruction takes one clock cycle to complete. This metric facilitates the framework during decomposition, when balancing actors and avoids blocking.

$$t_{(exec.)} = \frac{inst_{(actor)}}{F_{max}} \tag{5.1}$$

- *Register utilisation - $RF_{(util)}$*: It is the measure of registers used by single execution of an actor. It covers storage of input, intermediate and output variables used in a single iteration. This metric can aid the algorithm developer to find a suitable actor decomposition.

This section has presented IPPro core as FPGA-based soft-core dataflow accelerator supporting flexible mapping and execution of static multi-port dataflow actor. Besides, IPPro specific mapping constraints have been outlined that are essential for software profiling, mapping and compilation of dataflow actors onto IPPro. Section 5.3 investigates IPPro accelerator from a system level perspective, where multiple IPPro cores are connected and exchange tokens. The focus is to identify the system level management and control requirements, the inter-core communication mechanisms and their impact on the resource utilisation.

5.3 Management and provisioning of IPPro hardware accelerators

Hardware accelerators are used in data intensive computing systems, including many and multicore processors architectures [22], [23], [12], [43], [17]. Generally, in MPSoC-based systems, hardware accelerators are managed by a host/master processor. It handles system configuration, communication, data and control among accelerators that impacts the performance [113], [114]. It is vital to minimise host intervention not only in managing control and data transfer but also managing the hardware accelerators to achieve better acceleration. The hardware accelerators can be classified based on management policies into three classes [113]:

- Class I: Host managed dependent accelerator
- Class II: Host managed independent accelerator
- Class III: Self-managed independent accelerator

Table 5.5 lists the core, multicore and system level control and management requirements of each class of accelerators. To identify the desired synchronisation and inter-core communication mechanisms and analyse their impact on the area of each class of accelerator, four multiple IPPro core designs Ⓐ, Ⓑ, Ⓒ, Ⓓ have been implemented as shown in Figure 5.9.

Table 5.5: Impact of accelerator classes on IPPro-based core, multicore and system requirements [113].

Accelerator		**IPPro architecture**						
		Functional requirements					**Parallelism**	
Management	**Parallel Skeleton**	**Core level**	**Multicore level**	**Memory model**	**Control management**	**Code Sync.**	**Data**	**Task**
I) Dependent host managed	Pipeline	Host manages both data and control mechanisms	All accelerators are directly connected and managed by host processor (No inter-core communication)	Shared memory	Host controlled (complex host application)	Yes (host shall synchronise order of execution)	No	Yes
II) Independent host managed	Pipeline, Split-compute-merge	Instruction driven data mechanism TEST, GET, PUSH	Programmable inter-core communication controller	Message passing	Separate code for inter-core communication controller	Yes (between inter-core controller and each actor)	Yes	Yes
III) Independent self-managed	Pipeline, Split-compute-merge	Instruction driven data and control mechanism TEST, GET CH#, PUSH CH#	Self-managed inter-core communication	Message passing	Embedded within IPPro code	No	Yes	Yes

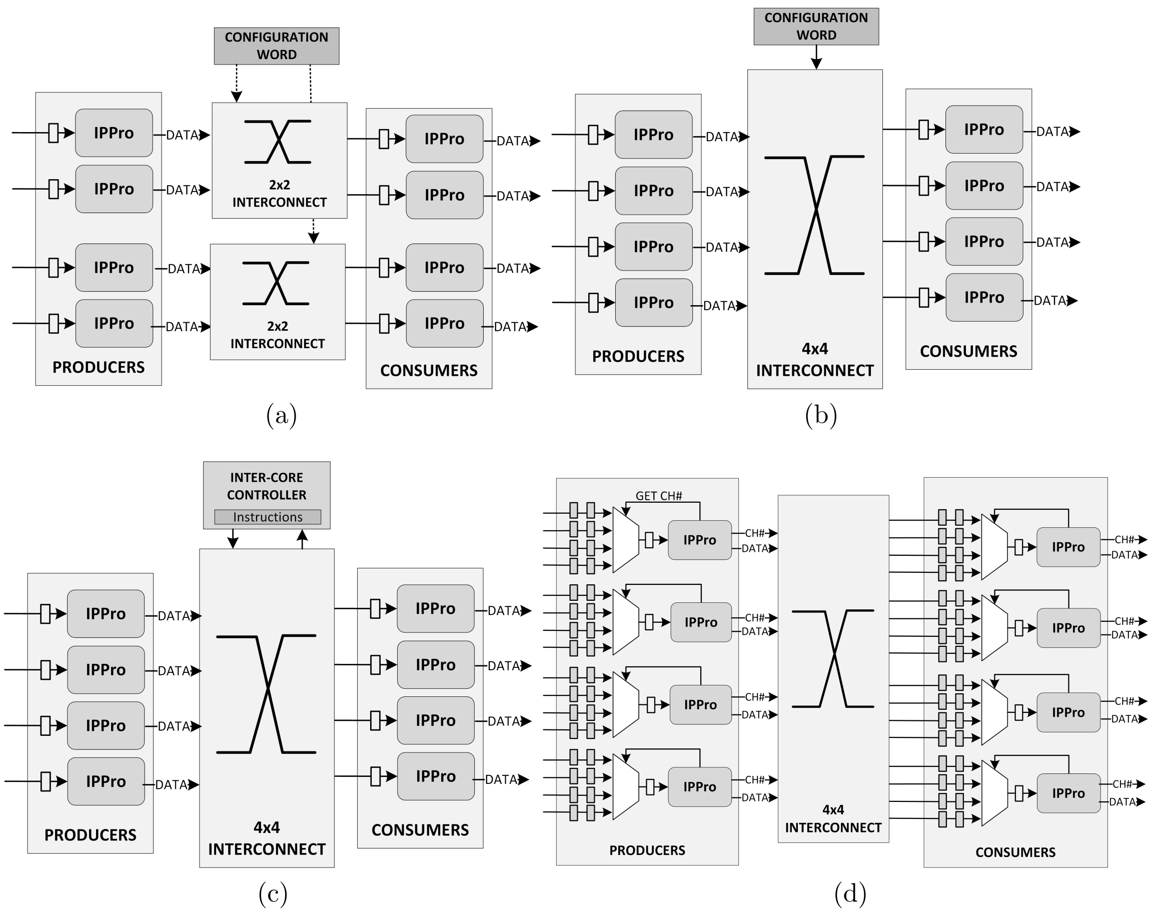


Figure 5.9: Multiple IPPro core-based hardware accelerator designs (a) Design Ⓐ (b) Design Ⓑ (c) Design Ⓒ (d) Design Ⓓ.

The *Class I* accelerator represents a "host managed dependent accelerator" which has been common in hardware solutions where the compute intensive part of the application is off-loaded on the dedicated IPs. The host assigns a job to the worker and is solely responsible for managing data distribution via shared memory using an appropriate control mechanism. IPPro is a stream accelerator that uses GET and PUSH instructions and does not require explicit data management by the host; therefore, *Class I* accelerators is not relevant. The designs Ⓐ, Ⓑ and Ⓒ functionally exhibit *Class II* accelerators as per Table 5.5. The difference

Table 5.6: IPPro-based multiple core architectures and their impact on system requirements and inter-core communication.

Design	**System requirement**			**Inter-core communication**			
	No. of cores	Host-core Synchro-nisation	Communication Management	Progra-mmble	Token re-ordering needed	Token deter-ministic	Inter-core connectivity
Ⓐ	8	Yes	Static configuration	No	Yes	No	1-1, 1-2, 2-1
Ⓑ	8	Yes	Static configuration	No	Yes	No	4-way split, merge,
Ⓒ	8	Yes	Inter-core controller	Yes	Yes	No	
Ⓓ	8	No	Self-managed	Yes	No	Yes	

between Ⓐ and Ⓑ is the level of inter-core connectivity of 2x2 and 4x4 between producer and consumer cores as shown in Figure 5.9(a) and (b). In Ⓐ and Ⓑ, the host processor statically configures the multiplexers during system configuration by setting a *configuration word* which remains fixed for the rest of the system operation. To map a tree-based dataflow actor, Ⓑ requires 4^N computing stages compared to Ⓐ, where N is the level of connectivity between cores which is 2x2 for Ⓐ and 4x4 for Ⓑ. When multiple cores are exchanging data simultaneously, both designs need collision avoidance mechanisms.

Kelly *et al.* have proposed a solution to address this issue by scheduling actors with a fixed offset [60]. This mechanism is common in HLS-based fine-grained architectures where the connectivity of dataflow actors are identified at compile time before realising hardware [22], [25], [43], [115]. On the other hand, Ⓒ supports dynamically configurable inter-core connectivity of 4x4 managed by an external *inter-core controller* as shown in Figure 5.9(c) which allows runtime configuration of the inter-core communication using routing program produced by the compiler extracted from the XDF. However, this increases hardware complexity as it requires synchronisation between IPPro cores and the controller. Lastly, design Ⓓ illustrates a *Class III* hardware accelerator where each IPPro core itself manages the inter-core communication. At the input interface, each core has FIFO queues (equal to the number of producers) which ensures deterministic token, resolve token re-ordering and avoids the collision. The design minimises host intervention and system level control compared to previous designs due to the absence of an external controller as shown in Figure 5.9(d). It has achieved this by attaching additional information (FIFO channel) along with a data token and forwarded to the interconnect where, each FIFO channel number represents the producer of data token.

This solution simplifies the system architecture by avoiding distributed control and data mechanisms and integrating them into a single point of control. It has been achieved by making IPPro an independent self-managed dataflow accelerator. It provides flexibility to explore and implement applications only by changing the IPPro program code that contains information related to data processing, control/synchronisation mechanism and exchange of tokens among multiple producer and consumer. Therefore, the application developer or software compiler has to generate only the IPPro code, instead of additional code for the inter-core controller as required by Ⓒ.

Implementation Results

Table 5.7 reports the implementation results obtained from Xilinx Vivado Suite v2015.2. Statically managed inter-core communication designs Ⓑ consumes 1.25 and 1.94 times more FFs and LUTs compared to Ⓐ by increasing the level of core connectivity from 2x2 to 4x4. On the other hand, using an *inter-core controller* to dynamically manage the inter-core communication further increases the FFs and LUTs utilisation by 1.07 and 1.20 times compared to Ⓑ in addition

Table 5.7: Impact on area utilisation of different accelerator configurations.

Design	FF	LUT	DSP	BRAM
Ⓐ	1902	709	8	8
Ⓑ	2381	1376	8	8
Ⓒ	2549	1632	8	8
Ⓓ	7616	5989	8	8

to complex system level synchronisation mechanism. Besides, Ⓓ result in $\approx$ 2.30 and 3.67 times increased in FFs and LUTs. Though, comparing Ⓓ with previously reported IPPro core results in Table 5.3, the presented functionalities and reduced management overhead come at the maximum cost of approx. 2.88 and 1.54 times of FFs and LUTs.

The presented area results in Table 5.7 shows that the increasing level of connectivity and avoiding off-loading host management tasks come at the cost of higher resource utilisation while the BRAM/DSP ratio remains constant.

5.4 Dataflow parallelism and multiple IPPro

Dataflow is a stream driven MoC that allows exploiting data and task level parallelism using different parallel computing paradigms as previously discussed in Section 2.2.1. IPPro is a light-weight programmable architecture that can use to realise programmable parallel dataflow computing system architecture by connecting multiple IPPro cores to exploit parallelism. In contrast to the pipelined parallel architectures, the iterative execution of a dataflow actor is a sequential operation which could take a variable number of clock cycles depending on the complexity of an actor. Therefore, to achieve acceleration, the *computation load* and *data transfer load* are chosen as application constraints which are defined as the actor execution time and token production-consumption rate. These application constraints shall be used by the compiler framework to find out the suitable application decomposition and mapping on the IPPro cores for the user. Frames per second fps has been chosen as a performance metric for image processing applications. Because it will be used as the input parameter to the compiler framework to start profiling and optimising the application. Mathematically, it can be represented using Equation 5.2.

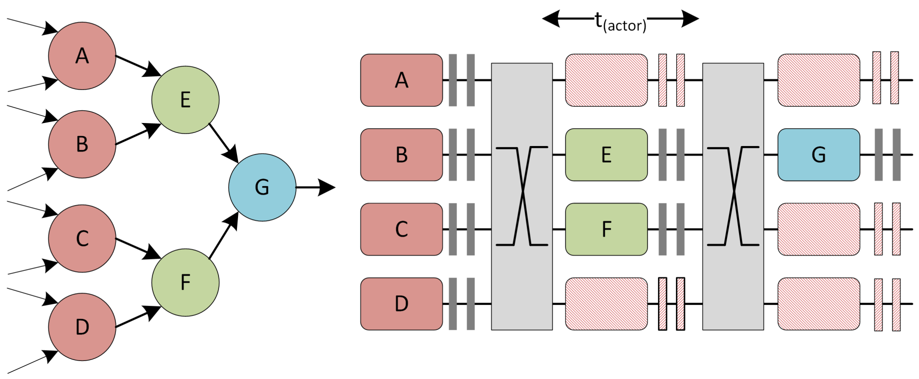
$$fps = \frac{f_{(IPPro)}}{t_{(actor)} * \frac{N_{(total\_pixels)}}{N_{(pixel\_consumption)}}} \tag{5.2}$$

where $f_{(IPPro)}$ is IPPro operating frequency (extensively discussed previously as performance metric in IPPro core level discussions and analysis development of IPPro core), $t_{(actor)}$ is the execution time (in clock cycles) of the slowest dataflow actor, $N_{(total\_pixels)}$ the number of pixels in a frame and $N_{(pixel\_consumption)}$ the number of pixels consumed by an actor in each iteration. To improve the fps, the following options are possible as depicted in Figure 5.10:

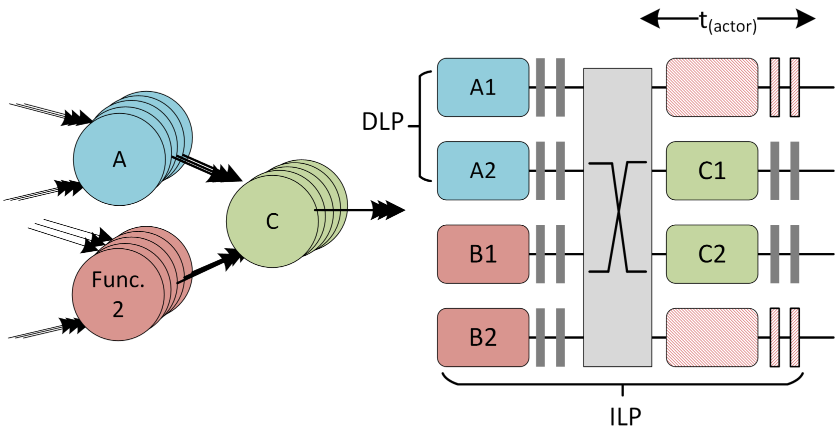
- **Reducing the actor's execution time** by decomposing it into multiple pipelined stages, thus reducing $t_{(actor)}$ to improve fps. Shorter actors can be merged sequentially to minimise the data transfer overhead by localising data into FIFOs between processing stages.

- **Vertical scaling to exploit data parallelism** by mapping an actor on multiple IPPro cores thus, reducing $(n * \frac{N_{(total\_pixels)}}{N_{(pixel\_consumption)}})$. Though, it requires an additional system level data distribution, control, and collection mechanisms.

Figure 5.10 shows two actor-core mapping examples to elaborate both optimi-



(a)



(b)

Figure 5.10: Multiple IPPro cores as dataflow accelerators deploying dataflow optimisations (a) One-to-one actor-core mapping (b) 2-way SIMD mapping per actor.

sations. The first example focuses on the pipelined one-to-one actor-core mapping of dataflow actors as shown in Figure 5.10(a) where individual actors $A - G$ are mapped on separate IPPro cores. The actors are unbalanced and have different execution times represented by $t_{(actor)}$. The inter-core communication architecture is used to exchange data among cores. This example illustrates pipelined mapping of dataflow actors using IPPro cores. It enables implementation of dataflow optimisation by dividing complex actor into multiple small actors and reduce the overall actor execution time.

The second example focuses on exploiting parallelism using vertical scaling of IPPro cores as shown in Figure 5.10(b). An actor is replicated onto multiple IPPro cores to exploit data parallelism. The level of connectivity supported by the interconnect defines the exploitable degree of data parallelism. This issue will be further discussed in Chapter 6.

5.4.1 Configurable data distribution and collection architecture

To realise parallel computing paradigms, scatter-gather is used to exploit data and task level parallelism [116], [71]. It uses the static decomposition of data where data is divided up into many equal-sized parts where each part can be processed by a separate processing core as shown in Figure 5.11. The research community has reported various image data distribution patterns driven by row, column and block-based static decomposition that result in row-strip, column-strip, row-cyclic, column-cyclic, block-wise and window-wise distributions [40], [117], [118]. In this thesis, the row-cyclic data distribution has been chosen because it allows buffering of pixels in a pattern suitable for *point* and *area* operations after storing them into the *line buffers*. It simplifies the reading process of pixels from the image buffer. The system level architecture composed of *line buffers*, a *scatter* module to distribute the buffered pixels, *gather* module to collect the processed pixels and a *finite state machine* to manage and synchronise these modules as shown in Figure 5.12.

The host processor uses control and data interfaces to configure, manage and

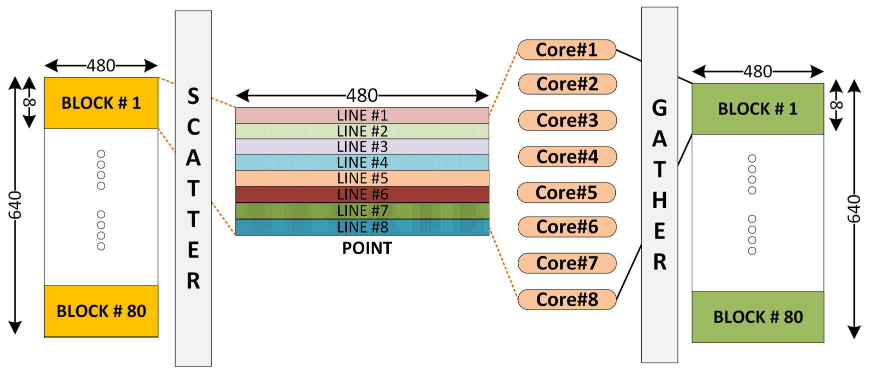


Figure 5.11: Cyclic row-wise image/video pixel distribution.

distribute pixels through a programmable host application. The host sequentially feeds the pixels into the *line buffers* using *IN* interface as shown in Figure 5.12. The width of the line buffer is configurable by loading a suitable value in $LINE\_WIDTH$ register using *AXI4-Lite* interface. It makes the system infrastructure adaptable to various image sizes. As soon as line buffers fill, the *Scatter* starts feeding data to the cores by storing it into the input FIFOs. The cores begin to process data as soon the actor firing rule is satisfied and pushes the processed data into the output FIFO. *Gather* reads processed data and feed it back to host processor using OUT interface. Figure 5.12 shows *Control* interface that is used to control the FSM presented in Figure 5.13 by the host processor and relevant output control signals for each state listed in Table 5.8. The following are the details of FSM states:

- **RESET** resets the programmable logic, i.e. IPPro cores, multicore interconnect and data distribution and collection mechanisms.

- **CONFIGURE_SYSTEM** enables the system SYS_EN and assigns a user-defined value to $LINE\_WIDTH$ register (as defined in the host application) which configures the *line buffer, scatter and gather* modules. The

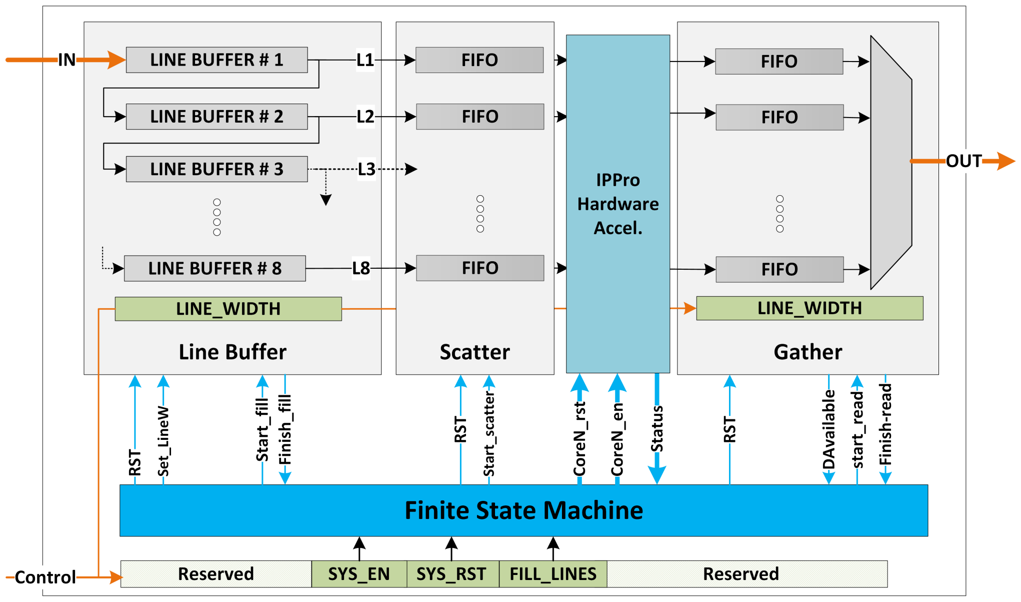


Figure 5.12: System level data distribution and control architecture.

value stored in the $LINE\_WIDTH$ register specifies the number of pixels stored in each line buffer.

- **IDLE** waits for host program to dispatch data by asserting $FILL\_LINES$.
- **FILL_BUFFERS** initiates filling of the line buffers by asserting $start\_fill$, waits until all line buffers are filled and assert $Finish\_fill$.
- **SCATTER** asserts *start_scatter* and IPPro *core_en* signals. The scatter module reads the line buffers and loads data into the input FIFOs. The core process data in parallel and stores processed data into respective output FIFOs. The *gather* compares the FIFO token count with LINE_WIDTH value and asserts $DAvailable$ signal which triggers the $GATHER$ state.
- **GATHER** asserts *start_read* signal and starts reading the output FIFOs of each core. It controls the multiplexer based on the defined $LINE\_WIDTH$

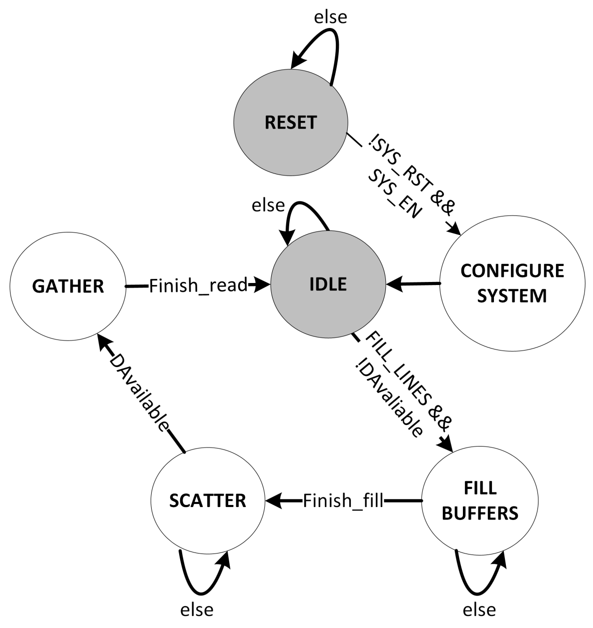


Figure 5.13: FSM used to control the architecture of Fig. 5.14.

value by checking the FIFO token count. The host reads processed data via OUT interface.

The presented stream-based data distribution and collection architecture abstracts the low-level hardware implementation details from the user and simplifies the application development process by providing underlying functionality via a control register. This approach provides task-level optimisations by pipelining multiple computing stages and localising data within the programmable logic

Table 5.8: Output signals of FSM for each state.

FSM output	**FSM State**					
	RESET	CONFIGURE SYSTEM	IDLE	FILL BUFFERS	SCATTER	GATHER
set_LineW	0	LINE_WIDTH				
start_fill	0	0	0	1	0	0
start_scatter	0	0	0	0	1	0
start_read	0	0	0	0	0	1
coreN_rst	1	0	0	0	0	0
coreN_en	0	0	0	0	1	1

before sending it back to a host processor thus reducing data transfer overhead.

To evaluate IPPro as a dataflow accelerator and implement some of the discussed dataflow optimisations (data and task level parallelism), Section 5.5 present the acceleration of k-means clustering.

5.5 Case Study: *k*-means clustering

Image segmentation is the process of partitioning an image into multiple segments. Three recognised methods by scientists and researchers are image thresholding, edge detection and clustering [119]. k-means belongs to *image clustering*, which is an unsupervised image segmentation method that classifies the image into a finite number of clusters. It has been chosen because of its simple control flow, data dependent execution and inherent fine-grained parallelism which makes it suitable for FPGA-based hardware acceleration [120]. It involves two stages which are *Distance Calculation* and *Averaging*. The distance calculation is mathematically represented as:

$$Y = \sum_{i=1}^{n} \sum_{j=1}^{k} (||P_i - C_j||)^2 \tag{5.3}$$

Where, $(||P_i - C_j||)$ is the *Euclidean distance* between a data point (pixel) P_i and a centroid value C_j, iterated over n points in the cluster for all k clusters. *Averaging* is used to calculate the updated centroid values for the next iteration by finding the average of clustered data/pixels in the dimension to find the new centroid value. In this case study, 512x512 resolution of images have been clustered by accelerating both stages of the k-means algorithm. To explore different

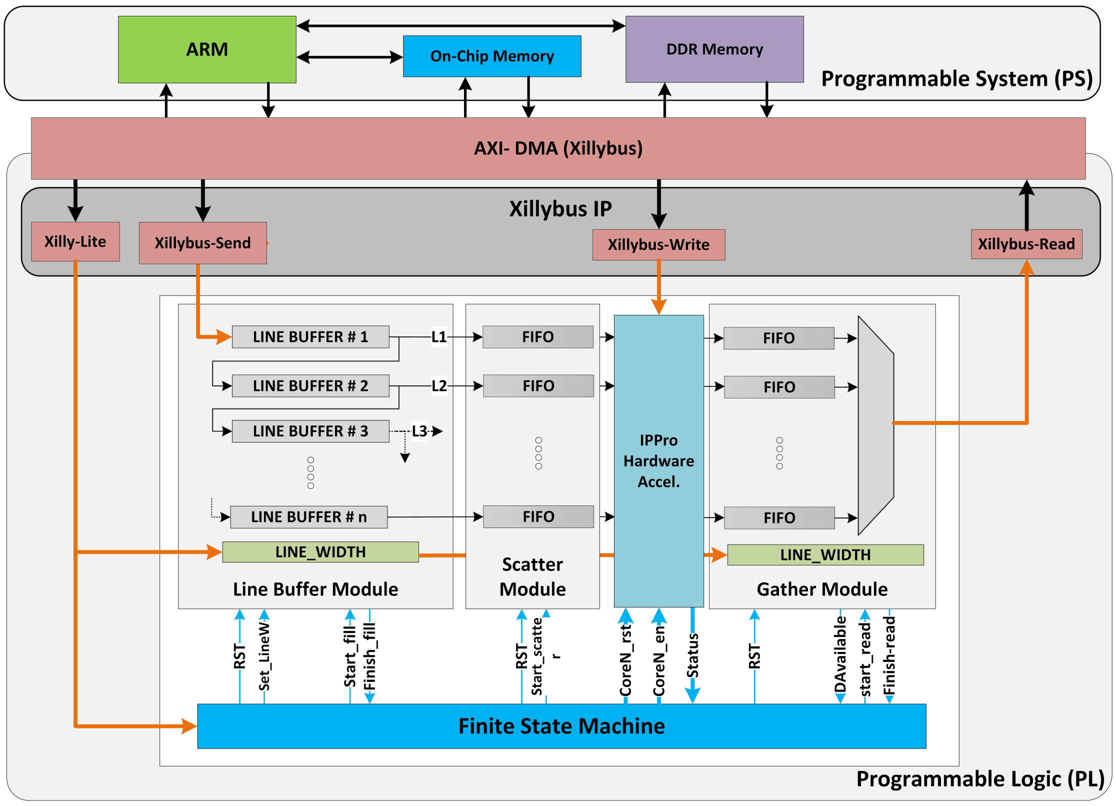


Figure 5.14: Block diagram of implemented system architecture for case study.

data and task parallelism and actor-core mapping possibilities, four *IPPro hardware accelerator* designs have been implemented. These designs cover single-core, dual-core, 8-way SIMD and dual 8-way SIMD-based IPPro acceleration architectures and allow evaluating the impact of exploiting data and task parallelism on area and performance. The system has implemented on Avnet Zedboard (XC7Z020CLG484-1) and the same *k*-means implementation has been realised on the desktop NVIDIA GTX980 GPU, embedded ARM Mali-T628 GPU and ARM Cortex-A7 CPU to compare the technologies.

5.5.1 MPSoC-based heterogeneous system architecture

Xilinx Zynq MPSoC is composed of a host processor known as *programmable system* (PS) and FPGA programmable logic (PL). The system architecture is used to accelerate the *distance calculation* and *averaging* using IPPro as shown in Figure 5.14. PS configures and controls the underlying architecture while PL is used to implement image processing pipeline and *IPPro hardware accelerator* as illustrated in Figure 5.14. The AMBA-AXI bus transfers the data between PS and PL using the AXI-DMA protocol. The Xillybus IP core [121] is deployed as a bridge between PS and PL to feed data into the image processing pipeline. It gives an intuitive DMA-based end-to-end turnkey solution for transporting data between PL and PS while running the Linux Operating System (OS) on an ARM host processor thus reducing engineering and device driver development effort [121]. The *IPPro hardware accelerator* interacts with the Xillybus IP core via FIFOs. The Linux application running on PS streams data between the FIFO and the file handler opened by the host application. The Xillybus-Lite interface allows control registers from the user space program running on Linux to manage the underlying hardware architecture.

Figure 5.14 shows the implemented system architecture which consists of the necessary control and data infrastructure. The data interfaces involve stream (Xillybus-Send and Xillybus-Read); uni-directional memory mapped (Xillybus-Write) to program the IPPro cores; and Xillybus-Lite to manage Line buffer, scatter, gather, IPPro cores and the FSM. Xillybus Linux device drivers are used to access each of these data and control interfaces. An additional layer of C functions is developed using Xillybus device drivers to configure and manage the system architecture, program IPPro cores and exchange pixels between PS and PL. Table 5.9 presents the developed C functions that the host application uses to program IPPro cores and control the system architecture are presented in Figure 5.14.

The Linux host application uses these C functions to feed image pixels into a *line buffer* module. These functions allow to control/manage the data distribution and collection architecture and program the IPPro cores using the process discussed in Section 5.4.1.

5.5.2 IPPro hardware accelerator designs

The case study is implemented to explore the different acceleration possibilities of *distance calculation* and *averaging*. Therefore, both stages are accelerated individually as an independent dataflow actor using single and multiple IPPro cores realised in design ① and ② as shown in Figure 5.15. Later, both stages are accel-

Table 5.9: Summary of the C functions running on the host processor to program and control the underlying architecture.

C function	**Description**

int open_system (void);
int close_system (int fd);
	The host uses *Linux User I/O* (UIO) interface to access the IPPro core as a device file by its memory map. This function is used to get the address of the PL hardware blocks by OS.

int system_reset (int fd, int addr);	It sets the SYS_RST bit
int system_enable (int fd, int addr);	It clears SYS_RST and set SYS_EN bit.
int set_line_size (int fd, int addr, short int value);	It sets the size of line buffer.
int fill_lines (int fd, int addr);	It sets the FILL_LINES bit.
int program_core (FILE *fp);	It programs the IPPro core by reading a .hex file using AXI-MM interface

int send_stream (short int *sdata, int len);
int read_stream (short int *rdata, int len);
	They are used to send/receive stream of data from host to PL using the Xillybus-Send and Xillybus-Read interfaces.

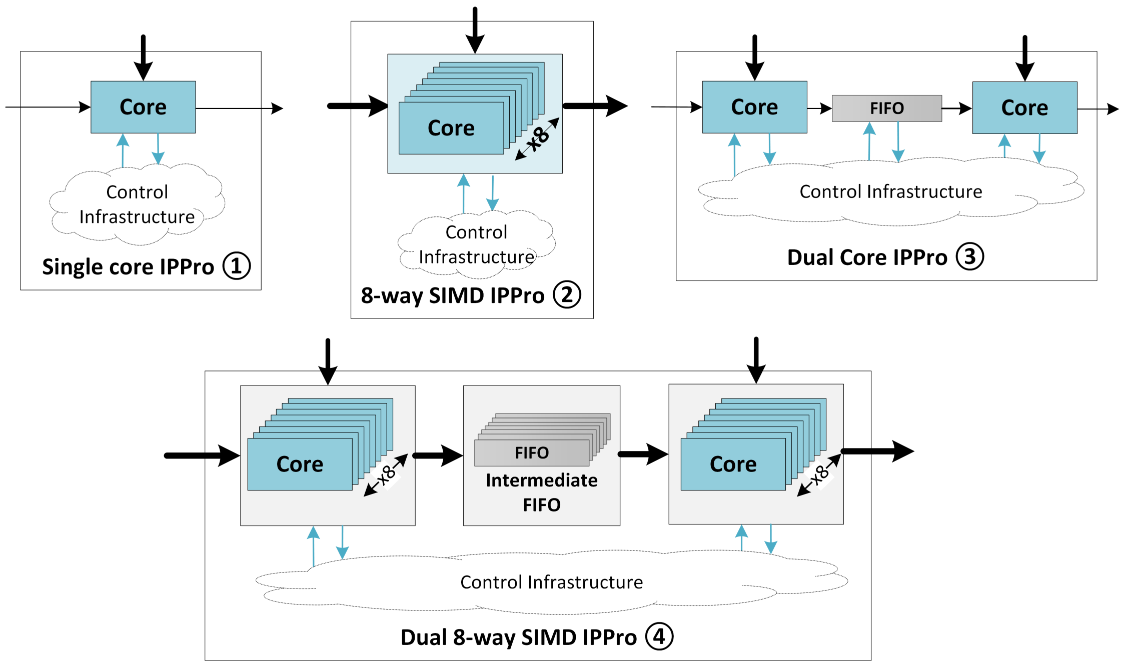


Figure 5.15: IPPro hardware accelerator designs to explore and analyse the impact of parallelism on area and performance. ① Single core IPPro, ② 8-way SIMD IPPro, ③ Dual core IPPro, ④ Dual core 8-way SIMD IPPro.

erated together as pipelined dataflow actors using dual and multiple-dual-IPPro cores realised in design ③ and ④. Figure 5.15 illustrates the block diagram of all four designs, and their data and control interfaces. Each design is used as a *IPPro hardware accelerator* illustrated earlier in Figure 5.14 and incorporated into the presented IPPro-based heterogeneous system architecture. These designs are selected as they enable different acceleration paradigms, dataflow actor mapping possibilities and parallelism options as listed in Table 5.10. Moreover, they allow the analysis of different algorithmic decompositions and their impact on the execution time and area utilisation.

Table 5.10: Dataflow actor mapping and supported parallelism of IPPro hardware accelerator design presented in Figure 5.15.

Design	Acceleration Paradigm	Dataflow mapping	Parallelism	
			Data	Task
①	Single core IPPro	Single actor	No	No
②	8-way SIMD IPPro	Single actor	Yes	No
③	Dual core IPPro	Dual actor	No	Yes
④	Dual 8-way SIMD IPPro	Dual actor	Yes	Yes

5.5.3 Acceleration results

The presented IPPro hardware accelerator designs have used different sample images for classification due to the data dependent characteristics of the clustering algorithm. Table 5.11 and Table 5.13 report the average execution time and fps numbers while, the area utilisation results have been reported in Table 5.12.

Table 5.11 reports the results obtained by individually accelerating the stages of k-means clustering using ① and ②. In each iteration, *distance calculation* takes two pixels and classifies them into one of the four clusters which take an average of 45 cycles/pixel. To classify the whole image, it takes 118.2 ms which corresponds to 8.45 fps. On the other hand, the *averaging* takes four tokens and produces four new cluster values, which takes an average of 55 clock cycles/pixel results in 145 ms or 6.88 fps. Both the stages involve point-based pixel processing. Therefore design ② is developed and used to exploit data level parallelism. As a result, the execution time is reduced to 23.32 ms and 27.02 ms for *distance calculation* and *averaging* respectively. This is an improvement of 5.06 and 5.37 times over ①. It came at the cost of 4.1, 2.3 and 8.0 times more BRAMs, LUTs and DSP blocks

Table 5.11: Performance measurements for design ① and ② of Figure 5.15.

Single Actor	① Single-core IPPro		② 8-way SIMD IPPro	
	Exec. (ms)	fps	Exec. (ms)	fps
Distance Calculation	118.21	8.45	23.37	42.78
Averaging	145.17	6.88	27.02	37.00

Table 5.12: FPGA area utilisation of various designs shown in Figure 5.15. The relative Zedboard area utilisation is also reported.

Design	**FF**		**LUT**		**BRAM**		**DSP**	
① Single-core IPPro	5197	(4.89)	4736	(8.90)	4.5	(3.21)	1	(0.45)
② 8-way SIMD IPPro	12279	(11.54)	10941	(20.57)	18.5	(13.21)	8	(3.63)
③ Dual-core IPPro	5737	(5.19)	5215	(3.21)	7.5	(3.21)	2	(0.90)
④ Dual 8-way SIMD IPPro	16106	(15.14)	13864	(26.06)	34	(13.21)	16	(7.27)

respectively as reported in Table 5.12. The major contributor to increased area utilisation is data distribution and control infrastructure. Theoretically, scaled up design has been expected to give eight times increase in performance, which is not achieved in ② because, the data transfer overhead involved in filling the line buffers, collecting the processed pixels and sending them back to the host is not negligible.

Table 5.13 reports the execution time and performance (fps) numbers of both stages together to exploit task-level parallelism using designs ③ and ④. The reported results of ① and ② obtained by combining the execution time of both stages previously reported in Table 5.11. Using design ③, the effect of task-level parallelism implemented via *intermediate FIFO* result in an average of 63 clock cycles/pixel which is 163 ms and 6 fps. By pipelining both actors, ③ has achieved 1.6 times better performance compared to ① at the cost of 1.6 and 2.0 times more BRAM and DSP blocks using the same Xillybus IP infrastructure as ①. The reason for the improvement is the localisation of intermediate data

Table 5.13: Performance with task-level parallelism using designs in Figure 5.15.

***k*-Means Acceleration**	**Performance**	
	Exec. (ms)	**fps**
① Combined stages using Single-core IPPro	263.38	3.8
② Combined stages using 8-way SIMD IPPro	50.39	19.8
③ Dual-core IPPro	163.2	6
④ Dual 8-way SIMD IPPro	35.9	28
Software implementation on ARM	286	3.49

within FPGA fabric using an *intermediate FIFO*, which hides the data transfer overhead to and from host processor as shown in Figure 5.15.

Analysing the impact of exploiting both task and data level parallelism using ④ results in average 14 clock cycles/pixel and execution time of 35.9 ms or 28 fps. It is 1.4, 4.5 and 7.3 times better than ②, ③ and ① respectively. For comparison, both stages are coded in C language and executed on an embedded ARM Cortex-A7 processor that achieved execution time of 286 ms and 3.49 fps which is 8 times slower than the performance achieved by ④.

5.5.4 Comparison against GPU implementations

This section presents the details of adopted power measurement methods and compares the IPPro-based implementation to the equivalent k-means GPU implementations. The IPPro power measurements obtained by running post-implementation timing simulation. A *Switch activity interchange format* (SAIF) file is used to record the switching activity of designs data and control signals of each presented IPPro designs. Xilinx Power Estimator (XPE) takes SAIF file and reports the power consumption. At *Queens University Belfast* (QUB) Minhas, a research student doing research on big data computing has coded an equivalent version of k-means in CUDA and OpenCL which is implemented and profiled on nVIDIA GeForce GTX980 and ODRIOD-XU3, due to in-house availability of both GPU platforms.

The nVIDIA desktop GPU card supports 2048 CUDA cores running at a base frequency of 1126 MHz. OpenCL and CUDA have used for programming the GPU, and both stages merged into the single kernel. For performance measurement, OpenCL's profiling function *clGetEventProfilingInfo* is used which returns the execution time of kernel in nanoseconds. The power consumption during kernel execution was logged using nVIDIA *System Management Interface* (nvidia-smi) which allows to measure the power consumed by the GPU and the host processor separately. It is a command line utility, based on top of the nVIDIA Management Library (NVML), intended to aid the management and monitoring of nVIDIA GPUs.

To set the base line figures and for fair comparison of the FPGA against the GPU technology, an embedded CPU (ARM Cortex-A7) and an embedded GPU (ARM Mali-T628) implementation has been carried out on ODROID-XU3 platform. This is a heterogeneous multi-processing platform that hosts 28nm Samsung Exynos 5422 application processor which has on-chip ARM Cortex-A7 CPUs and ARM Mali-T628 embedded GPU. The platform is suitable for power constraint application use cases where ARM Cortex-A7 CPU and mid-range ARM Mali-T628 GPU runs at 1.2 GHz and 600 MHz respectively. The platform have separated current sensors to measure the power consumption of ARM Cortex-A7 and ARM Mali-T628, thus allow component-level power measurement capability.

Table 5.14 shows the results of IPPro-based accelerator designs running on Zedboard where both data and task parallel implementation achieved 4.6 times better performance over task only implementation at the cost of 1.57 times higher power consumption. Table 5.15 shows the performance results of the k-means implementation on Kintex-7 FPGA and compares them against equivalent embedded CPU (ARM Cortex- A7), embedded GPU (ARM Mali-T628) and desktop GPU (nVIDIA GeForce GTX680) implementation. The presented embedded CPU results has been considered as baseline figures for the comparison.

Table 5.14: Power, resource and combined efficiency comparisons of IPPro-based k-means implementations on Zedboard.

Implementation	**Power (mW)**			**Freq.**	**Exec.**	**fps**	**Power**	**Approx.**	**Resource**	**Combined**
	Static	**Dynamic**	**Total**	**(MHz)**	**(ms)**		**efficiency (fps/W)**	**transistor utilised (TU)** ($x10^{6}$)	**efficiency (fps/TU)** ($x10^{-8}$)	**efficiency (fps/W/TU)** ($x10^{-9}$)
Dual-core IPPro	118	18	136	100	163.2	6	44.1	591 (9%)	1.0	74.6
Dual 8-way SIMD IPPro	122	92	214	100	35.9	28	130.8	1564 (23%)	1.8	83.6

Table 5.15: Power, resource and combined efficiency comparisons for k-means using Xilinx Zynq XC7Z045 Kintex-7 FPGA and GPU NVIDIA GTX980.

Platform	**Implementation**	**Power (mW)**			**Freq.**	**Exec.**	**fps**	**Power**	**Approx.**	**Resource**	**Combined**
		Static	**Dynamic**	**Total**	**(MHz)**	**(ms)**		**efficiency (fps/W)**	**transistor utilised (TU)** ($x10^{6}$)	**efficiency (fps/TU)** ($x10^{-8}$)	**efficiency (fps/W/TU)** ($x10^{-9}$)
FPGA	**Dual-core IPPro**	158	26	184	337	48.43	21	114.1	591 (9%)	3.6	193.1
	Dual 8-way SIMD IPPro	160	153	313	337	10.65	94	300.3	1564 (23%)	6.0	192.0
GPU	**OpenCL**	37000	27000	64000	1127	1.19	840	13.1	1331 (26%)	63.1	9.8
	CUDA	37000	22000	59000	1127	1.58	632	10.7	1227 (24%)	51.5	8.7
Embed. GPU	**ARM Mali-T628**	120	-	1560	600	3.69	271	173	-	-	-
Embed. CPU	**ARM Cortex-A7**	250	-	670	1200	286	3.49	5.2	-	-	-

Comparing the performance results (fps), both FPGA implementations achieved 6 and 27 times over the embedded CPU, while the embedded GPU delivered 6.7 times better performance over the FPGA by exploiting parallelism and higher operating frequency. Focusing on the power consumption results, the FPGA consumed 2.1 and 4.9 times less power than both the embedded CPU and embedded GPU respectively. It shows that the FPGA technology delivers power-optimised solution while, the GPU technology provides performance-optimised solution. Though by considering both performance and power together, the power efficiency (fps/W) numbers shows that FPGA and embedded GPU implementations are 57 and 33 times more power efficient than embedded CPU. These results shows that FPGA implementation is 24 times more power efficient than embedded GPU. Nevertheless, this power efficiency edge can be further improved by applying dataflow transformations and increasing the number of IPPro cores.

Table 5.15 also compares the FPGA results against desktop GPU and reports resource efficiency as a metric due to significant difference in the power consumption numbers. The resource efficiency has been presented in terms of frames-per-second-per-Transistor-Utilisation (fps/TU) which is 6 and 63 for 28nm FPGA and GPU technologies. For embedded CPU and GPU, these results are not reported due to unavailability of transistor count numbers by the ARM. The reported resource efficiency results shows that GPU utilises area resources more efficiently than FPGA when power is kept out of the equation. Combining all three metrics (fps/W/TU) shows that the advantage gained from FPGA designs is significant which is 22 times more efficient than GPU. This advantage becomes more valuable considering the fact that presented FPGA-based MPSoC design is adaptable, allows exploration, profiling and implementation of different dataflow transformation possibilities over dedicated FPGA approaches to accelerate image processing applications where energy is vital.

5.6 Summary

This chapter presented IPPro as a programmable dataflow accelerator that supports dataflow MoC. The presented IPPro architecture implements multi-port static dataflow actors supported with notion of firing actor and execution patterns using producer-consumer computing model. These execution patterns provide flexible mapping options to the user and software framework to explore and deploy dataflow optimisations. The input and output FIFOs of IPPro are realised and implemented using BRAM, DistRAM and SR. IPPro implementation using DistRAM-based FIFO has achieved timing of $\approx$ 242 MHz, utilising $\approx$ 8% more LUTs compared to BRAM-based implementation. On the other hand a degradation of 3% and 19% in operating frequency has observed for SR and BRAM-based FIFO implementations.

To use IPPro as basic dataflow computation unit in heterogeneous MPSoC-based system architecture requires communication between the accelerator and the host. Four multiple IPPro core designs have been implemented to evaluate the impact of host-accelerator communication and inter-core communication mechanism on area utilisation. These designs cover the different level of connectivity between producer and consumer cores, as well as static and dynamic handling of inter-core connectivity managed either by the host or the core itself. The last design Ⓓ has offered desired functionalities with reduced management overhead at the maximum cost of 2.88 and 1.54 times more FFs and LUTs.

To deploy dataflow optimisations (decomposition, mapping, and scheduling) using multiple IPPro cores different actor-core mapping possibilities are discussed supported with inter-core communication. A configurable stream-based data distribution and collection system architecture has proposed to deploy and realise the selected optimisations. The architecture abstracts the low-level hardware implementation details from the user and simplifies application development process by providing underlying functionality via C-APIs. The design facilitates exploitation of discussed dataflow optimisations including multi-stage pipelined and parallel computing models (split, compute, and merge).

To evaluate the proposed architecture deploying dataflow optimisations, distance calculation and averaging stages have implemented on Avnet Zedboard. Four IPPro hardware accelerator designs have realised that cover single-actor, dual-actor, data and task level parallelism. The obtained results show that by exploiting both data and task level parallelism, it is possible to achieve 7.3 times better performance than task parallelism alone. Comparing against other technologies, FPGA achieved 27 times better performance over the embedded CPU by exploiting parallelism and consumes 4.9 times less power than the embedded GPU. Moreover, the power efficiency (fps/W) numbers shows that FPGA implementation is 57 and 24 times more power efficient than embedded CPU and GPU respectively.

Chapter 6

FPGA-based programmable hardware acceleration platform

6.1 Introduction

Many and multicore hardware accelerators have been used in data intensive computing systems [22], [23]. Despite the efficiency of the heterogeneous system, the designers and system architects are facing challenges to quickly implement tailored applications on FPGA-based platforms to meet design goals [12], [43], [17]. One of the shortfalls in these parallel architectures is the scarcity of hardware abstraction, which makes it difficult for application designers to efficiently use the available FPGA compute resources [17], [122]. It requires a certain level of hardware knowledge which software and application developers lack in order to maximise efficiency and reusability of the available parallel architecture as, it involves handling of the low-level core, inter-core and system communication and system interfaces etc. To approach this problem is by designing an

FPGA-based multicore processor using an IPPro core that allows an efficient, high-performance, fine and coarse-grained mapping and execution of dataflow actors. This multicore processor extends the flexibility provided by the IPPro core and allows both pipelined and parallel execution of dataflow actors to realise *programmable streaming networks*. Multiple instances of these multicore processors can be cascaded together to achieve a *FPGA-based programmable hardware acceleration platform*. This platform facilitates exploration, profiling and acceleration of image processing applications to software and algorithm developers using software-centric *edit-compile-run* flow by avoiding *synthesis and place-and-route* design flow. In addition, it supports implementation of parallel computing skeletons that provide higher programming abstraction of parallel structures which can be efficiently realised on the underlying architecture to implement parallel applications. The resulting platform allows software-controlled adaptable execution of parallel skeletons, by abstracting the underlying hardware architecture to the developer which gives better granularity to the application programmer realising parallel applications using FPGA technology. The following are the main contributions :

- Creation of an optimised IPPro core architecture which supports *message passing* and *shared* data models to process uniform and non-uniform distributed data. These data models enable realisation of *split, compute and merge*, *pipeline* and *farm* parallel skeletons.

- A novel multicore IPPro architecture that supports dynamic routing of data streams among cores, exploiting parallelism using horizontal and vertical scaling. It includes one-to-many, many-to-one, many-to-many producer-consumer data passing patterns for flexible actor-core mapping possibilities.

- A software configurable data distribution and collection architecture to realise parallel implementation on heterogeneous architecture. It handles different image resolutions, provides flexible control on data stream generation and distribution and can be integrated in direct and buffered video pipelines.
- Software abstraction of the proposed programmable platform and its hardware supported features to realise software driven parallel implementations.

This chapter presents IPPro core-level optimisations in Section 6.3. It covers incorporation of data and control mechanisms required to implement parallel skeletons, hardware-optimised implementation of dataflow actor firing rule to minimise control overhead, and implementation results of the optimised IPPro core architecture. Section 6.4 presents the multicore IPPro architecture focusing on the identification of multicore architectural features, exploration of a suitable stream-based multicore interconnect design and their impact on performance and core utilisation. Section 6.5 presents the FPGA-based programmable hardware acceleration platform with focus on dynamic data distribution and collection requirements for parallel implementations. Section 6.6 discusses the performance results of the chosen image processing functions exploiting data/task parallelism, and heterogeneous computing to evaluate the flexibility of the platform. Each IPPro acceleration result is compared against the equivalent optimised ARM implementation.

6.2 Programmable realisation of parallel skeletons on FPGAs

Parallel *skeletons* are pre-defined generic components derived from higher-order functions which can be parametrised in sequential problem-specific code and can be efficiently implemented on hardware architectures [116], [123]. In this research, a data-driven producer-consumer computing paradigm has been adopted which can be used to exploit data and task parallelism. Therefore, the underlying architecture must support the desired data exchange and synchronisation mechanisms and the functional requirements of skeletons. For this purpose, Figure 6.1 presents three-layer programming (actor, parallel actors and parallel skeletons) and hardware (IPPro core, Multicore IPPro, System Infrastructure) abstraction.

From bottom-up, a *programmable streaming unit* supports the functional requirements of a dataflow *actor* and, a *programmable streaming network* supports dataflow driven data exchange patterns across multiple actors to enable flexible mapping possibilities to implement *parallel actors.* The top layer allows parametric implementation of a *parallel skeleton* by supporting stream and non-stream data access and control mechanisms that are necessary to exploit parallelism.

Figure 6.1 shows the *hardware abstraction* to realise the *programmable hardware acceleration platform.* The IPPro core is used to implement a programmable dataflow actor, the multicore IPPro gives algorithm exploration possibilities using different actor-core mappings of multiple actors. The *system infrastructure* allows the necessary software configurable data distribution and collection mechanisms to support control and data requirements of parallel skeletons.

To this end, the IPPro core already supports some architectural features as

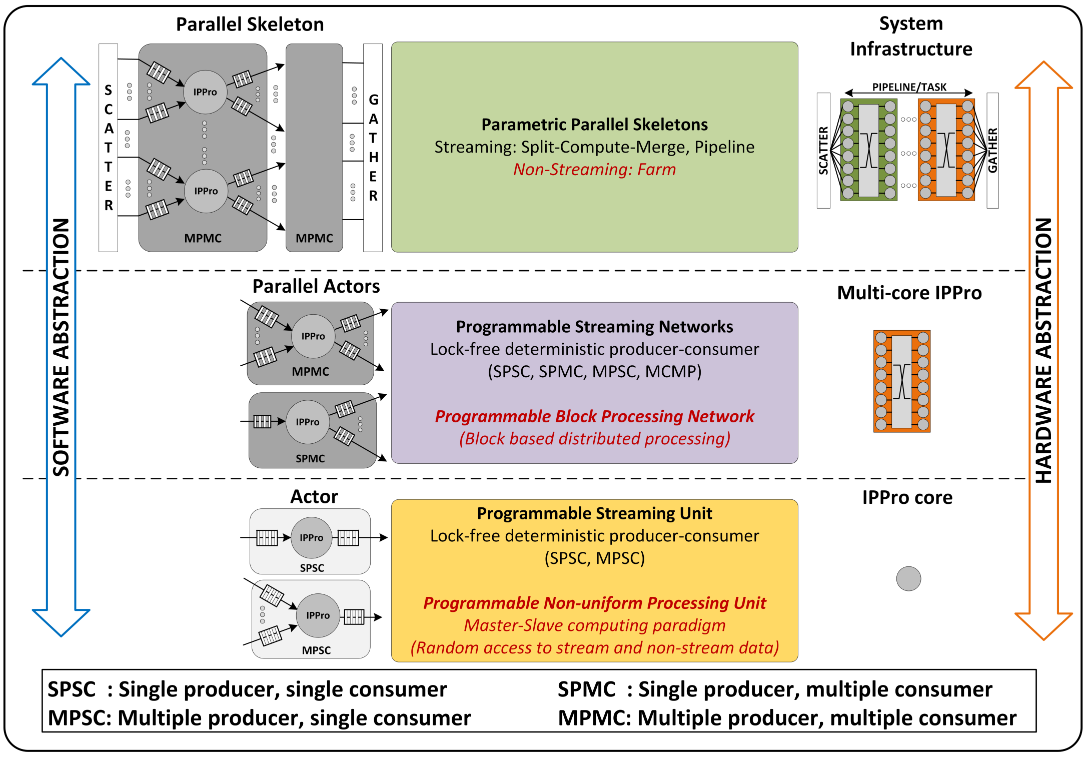

Parallel Skeleton
SCATTER
IPPro
MPMC
MPMC
GATHER
Parametric Parallel Skeletons
Streaming: Split-Compute-Merge, Pipeline
Non-Streaming: Farm
System Infrastructure
PIPELINE/TASK
SCATTER
GATHER
Parallel Actors
IPPro
MPMC
SPMC
Programmable Streaming Networks
Lock-free deterministic producer-consumer
(SPSC, SPMC, MPSC, MCMP)
Programmable Block Processing Network
(Block based distributed processing)
Multi-core IPPro
Actor
IPPro
SPSC
MPSC
Programmable Streaming Unit
Lock-free deterministic producer-consumer
(SPSC, MPSC)
Programmable Non-uniform Processing Unit
Master-Slave computing paradigm
(Random access to stream and non-stream data)
IPPro core
SOFTWARE ABSTRACTION
HARDWARE ABSTRACTION
SPSC : Single producer, single consumer
MPSC: Multiple producer, single consumer
SPMC : Single producer, multiple consumer
MPMC: Multiple producer, multiple consumer

Figure 6.1: Software and hardware abstraction of the platform.

presented in Chapter 4 and 5. The architectural features which are required and not supported by existing IPPro datapath are *highlighted* in Figure 6.1. Section 6.3 presents IPPro core optimisations focusing on data and control mechanisms needed to implement parallel skeletons and the hardware-optimised implementation of dataflow actor firing rule.

6.3 IPPro core architectural optimisations

The existing IPPro datapath supports a message-passing data communication model which is only suitable for stream and uniformly distributed data processing to realise *split, compute, merge* and *pipeline* skeletons. On the other hand, the *farm* skeleton requires access to non-uniform distributed data which need data memory. This memory would serve as a data exchange path between master (host) and worker (IPPro) as abstracted in Figure 6.1. It facilitates the implementation of *global* functions (subject to the size of data memory) using IPPro cores. Based on the functional requirements, the following optimisations have been identified:

1. Optimisation of dataflow actor firing rule minimising control overhead to implement multiple-consumer and multiple-producer dataflow actors (Section 6.3.1).
2. IPPro scratchpad memory to exchange data between IPPro and the host processor to realise a *farm* skeleton (Section 6.3.2).
3. IPPro core interfaces compliance with industry standard MPSoC communication protocols for easy integration and portability within SoC and other systems as IP (Section 6.3.3).

6.3.1 Dataflow actor firing rule optimisation

Chapter 5 presented a programmable software solution to handle actor firing rule which is not suitable for multi-port actors (MPMC and MPSC). It adds execution overhead directly proportional to the number of producer nodes as shown in Listing 6.1. During code execution, the core iteratively checks the firing rule dedicated to each producer node using branch instructions. As a result, the actor's execution time is dependent on the number of producer nodes.

A hardware *actor firing module* has been designed and integrated into IPPro datapath to reduce execution overhead as shown in Figure 6.2. The control interface allows configuration of eight *set token count registers* (STC_Q0 - STC_Q7) to check the number of tokens available from each producer. From the multicore architecture perspective, it allows actor mapping opportunities for up to eight producers feeding an actor by storing tokens into their appropriate FIFO queues. Also, an *actor firing mask* (AFMR) register holds the information about the number of producer nodes connected to the actor while values stored in (STC_Q0 - STC_Q7) registers define the number of tokens expected from each producer.

Listing 6.1: IPPro code of un-optimised actor firing rule.

```
; Check if the expected number of tokens (1, 2, 1) in FIFO queues coming from
; source nodes (0, 1, 2) are available? If yes, fire the actor
# Store expected number of tokens from each producer node
STR R1, 1 ;
STR R2, 2 ;
STR R3, 1 ;
...
# Check producer#1 rule
CHECK_RULE1:
TEST R20, R1, #0
BNZ  CHECK_RULE1
...
# Check producer#2 rule
CHECK_RULE2:
TEST R20, R2, #1
BNZ  CHECK_RULE2
...
# Check producer#3 rule
CHECK_RULE3:
TEST R20, R3, #0
BNZ  CHECK_RULE3
...
ACTOR_FIRED:
...
```

These registers are initialised by the host. During actor execution, the *actor firing module* concurrently reads the token counts of input FIFO queues, compares them against (STC_Q0 - STC_Q7), masks it with AFMR, and updates the result in *firing status register* (FSR) as shown in Figure 6.2. This allows software integration of an actor firing rule into the IPPro code using TEST instruction. Individual bits of the FSR shows the availability of expected number

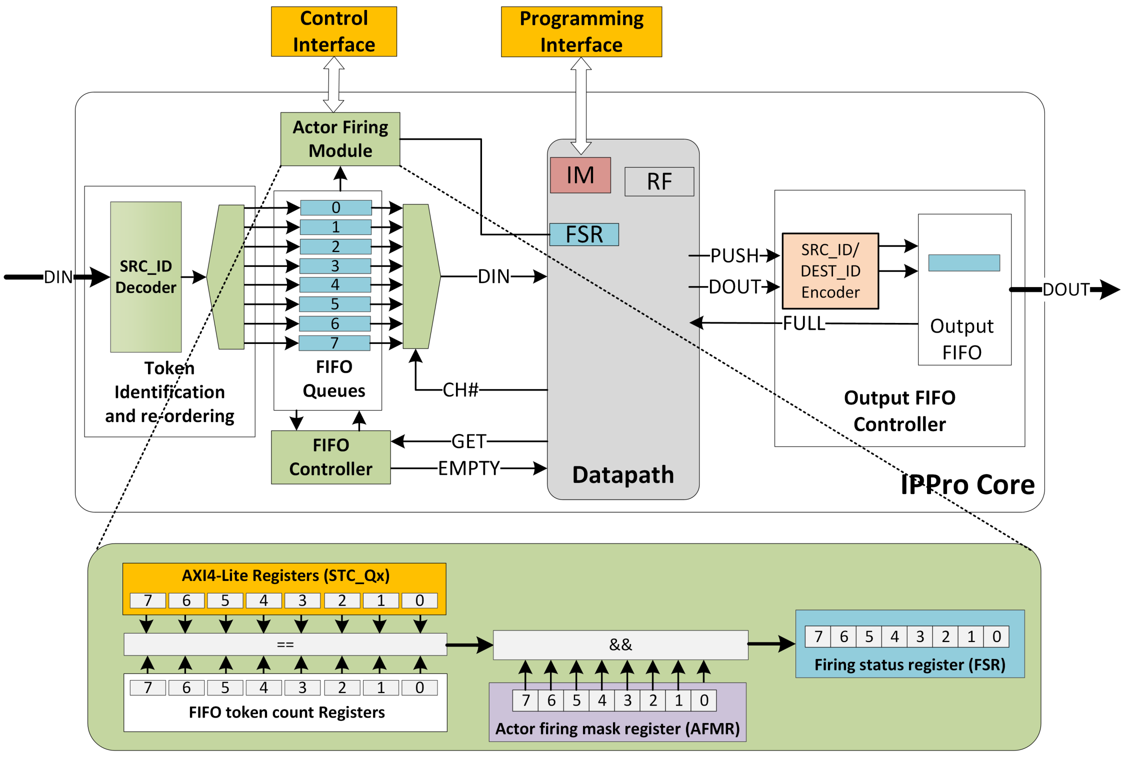


Figure 6.2: Block diagram of hardware dataflow actor firing module.

of tokens from each producer. It compares the value of FSR against the set ACTOR_FIRING_MASK defined in IPPro code as shown in Listing 6.2.

Once an actor has fired, execution of $GETRx, CHANNEL\#$ reads the token from the addressed FIFO queue and stores it into the addressed location of the register file. Similarly, $PUSHRx, CHANNEL\#$ reads token from the register file and forwards it to the output FIFO. The *output FIFO controller* shown in Figure 6.2 encodes SRC_ID and DEST_ID tags, required to re-order and route tokens to the different consumer node. The SRC_ID and DEST_ID specifies a source node (producer) and a destination node (consumer) of the token.

By comparing the execution time of the presented IPPro code Listing 6.1 and 6.2 shows that optimised implementation takes a fixed number of clock cycles which is independent of the number of producer nodes. The proposed hardware *actor firing module* enables programmable implementation of both fixed and multi-rate actor firing rule using IPPro by merely changing the program code. On the contrary, high-level synthesis approaches generates a fixed architecture [25], [43], [124] that needs design recompilation, synthesis and place-and-route to deploy small changes such as actor firing rule.

Listing 6.2: IPPro code of optimised actor firing rule.

```
; Check if the expected number of tokens set by the host STK_Qx expecting from
; source node 0, 1, 2, 3, 4, 5 are available in respective FIFO queues?
; If yes, fire the actor
CHECK_FIRING_RULE:
        STR  R15, #0000_0000_0011_1111   ; Set ACTOR_FIRING_MASK
        TEST R30, R15                    ; Check FSR
        BNZ  CHECK_FIRING_RULE
         ...
ACTOR_FIRED:
        GET  R10,#0      ; Read token from node # 0
        PUSH R20,#1      ; Send token to node # 1
         ...
        JMP CHECK_FIRING_RULE
```

6.3.2 Scratchpad memory to access non-streaming data

Section 6.3 outlined the importance of *data memory* in the IPPro datapath, providing a path between the host processor and the IPPro core to implement *farm* parallel computing skeleton. For this purpose, a *scratchpad memory* of size 512x16 bits configured as true dual-port RAM has been added into the IPPro datapath. This design choice has been made to efficiently utilise the BRAM resources as, 512x16 bits size maps well on 18KB BRAM block (half of the BRAM). The other half of the BRAM has been used for the instruction memory. One of the port is connected to the host processor via an AXI4 interface, and the other to the datapath using a native interface as shown in Figure 6.3.

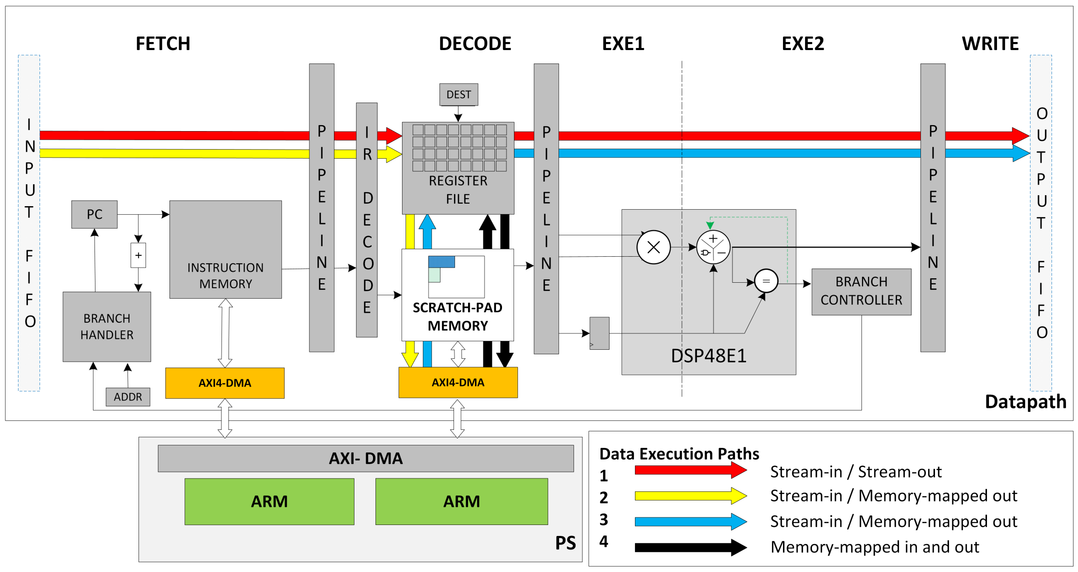


Figure 6.3: Data processing paths of the IPPro using scratchpad.

To maintain a balance among better functionality, area and timing, six additional instructions have been supported by IPPro to access scratchpad memory as listed in Table 6.1. These instructions allow reading and writing data into the *scratchpad memory* and return assigned task status to the host processor. *Direct* (LDSP, STSP) and *in-direct* (LDSPI, STSPI) addressing modes facilitates iterative access to memory locations using loops and offsets, which are commonly practised by software programmers. Listing 6.3 shows the example code using direct and indirect addressing modes to access the scratchpad memory.

This optimisation has also improved the data processing capabilities of IPPro core by processing stream and non-streamed data simultaneously using four sup-

Table 6.1: IPPro instructions to access scratchpad memory.

IPPro Instruction	**Description**
TASK_FINISHED	Inform host that task is completed
SP_VALID	Inform host that scratchpad is valid
LDSP, STSP	Load/store data to/from directly addressed location
LDSPI, STSPI	Load/store data to/from indirectly addressed location

ported data execution paths as highlighted in Figure 6.3. These data execution paths facilitate a flexible dataflow actor to IPPro core decomposition and mapping options. It has been done in such a way that the stream execution path has minimal data transfer overhead compared to non-stream execution path. This is because the FIFO-based transfers exploits pipelining compared to memory-based transfers via a host processor where cache coherency latencies can be significant.

Listing 6.3: Code demonstrating direct and in-direct access to the scratchpad.

```
# Indirect access to scratchpad memory using loop
INIT:
        STR R31,#1      ; Loop initial value / indirect address pointer
        STR R1, #1      ; Loop increment constant value
        STR R20, #10    ; Loop terminate count value
        ...
LOOP:
        LDSPI R21, R31          ; R21 <= SP[R31]
        ADD R31, R31, R1        ; Increment loop count
        SUB R22, R31, R20       ; Check whether Loop condition
        BNZ LOOP
        ...
# IPPro core as a hardware accelerator (farm worker)
# c = FUNC(a*b)
# It is pre-defined that SP(0) = a ; SP(1) = b ; SP(3) = c
FUNC:
        LDSP R1, #1     ; Load a
        LDSP R2, #2     ; Load b
        ...
        MUL  R3, R1, R2
        STSP R3, #3     ; Store c
        SP_VALID
        ...
        JMP FUNC
```

6.3.3 Host management of IPPro core using AMBA-AXI4

A vital aspect of any SoC solution is not only the hardware components it houses, but also the way these components are connected. The ARM *Advanced Microcontroller Bus Architecture* (AMBA) is an open-standard on-chip interconnect specification. Most leading SoC chips supports the fourth generation *AMBA-AXI4*. In these systems, a host processor configures, manages and in some cases

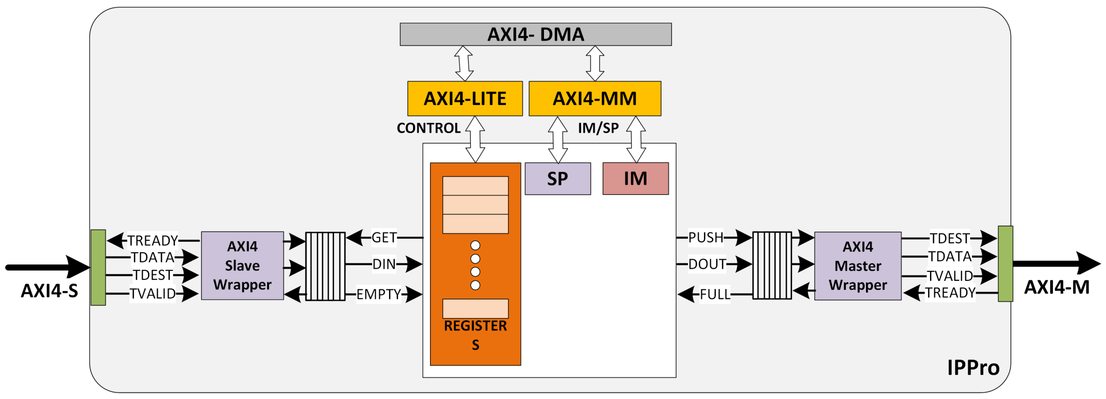


Figure 6.4: AMBA-AXI4 compliant management interfaces of the IPPro.

feeds data to the slaves. AMBA-AXI4 specification supports three protocols: 1) AXI4-Lite to provide register-based control mechanisms 2) AXI4-Stream to feed a stream of data 3) AXI4-memory mapped to exchange random access data between the host and the underlying architecture.

IPPro supports all three AMBA-AXI4 protocols where AXI4-Lite interface is used to configure *actor firing module*, *SRC_ID decoder* and *DEST_ID encoder* using nine AXI4-Lite IPPro registers (for details see Appendix B Table B.3). It has two AXI4-memory-mapped interfaces that allow the host processor to program instruction memory and access scratchpad memory. Two AXI4-Stream interfaces allow sending/receiving a data stream into the core, which can be either the host processor via direct memory transfer or system architecture. The *AXI4 Slave and Master wrapper* modules are added into the IPPro datapath as shown in Figure 6.4 that convert a native FIFO handshaking to AXI4-Stream interface. They use native EMPTY and FULL handshake signals to generate respective AXI4 master and slave handshake signals (TREADY and TVALID). The modules also handle separation of the data payload (TDATA), routing tags (TDEST), and the generation of reading and writing control signals to a native DIN, DOUT

Table 6.2: Implementation results of the optimised IPPro on Kintex-7 fabric.

Resources	**Initial IPPro**	**Optimised IPPro**
Flip Flops	447	884
LUTs	484	755
BRAMs	1	1
DSP48E1	1	1
Freq. (MHz)	337	300

ports. C-APIs have been developed to abstract control and management of the core (for details see Appendix B).

6.3.4 Implementation results of optimised IPPro core

The optimised IPPro datapath is synthesised and implemented using Xilinx Vivado v2016.4 design suite. Table 6.2 summarises the results and compares them against the initial IPPro core indicated in Table 1.12. The critical path has increased approx. 11% and resulted in operating frequency of 300 MHz. This f_{Max} reduction come at the cost of fix actor firing execution time (Section 6.3.1), and data compute capability of both stream and non-streamed data (Section 6.3.2). The optimised datapath consumes 1.9 and 1.5 times more FFs and LUTs, while the BRAM/DSP ratio remains constant. Generally, an FPGA fabric has two times more FFs than LUTs and therefore, the maximum number of cores that can be populated on the chip will be affected by the FF/LUT ratio. Regarding mapping possibilities, an actor with up to eight producer nodes which has been reflected in the reported LUT utilisation. This increase in LUT utilisation is caused by eight 16x32 FIFO queues to re-order received data tokens from multiple producers. Similarly, increase in FF utilisation occurred due to FIFO count registers used by the hardware *actor firing module*, and AXI4-Lite registers which were absent in initial IPPro. However, the area utilisation represents $< 1\%$ of

Table 6.3: Comparison of IPPro against other FPGA-based soft-core processors.

Resource	IPPro	Graph-SoC [16]	FlexGrip [36]*	MicroBlaze
Flip-flops	884	551	12972	518
LUTs	755	974	8916	897
BRAMs	1	9	15	-
DSP48E1	1	1	19.5	3
Stages	5	3	5	5
Freq.	300	200	100	211

* Scaled to a single streaming processor.

the available on-chip resources.

Table 6.3 compares the results of optimised IPPro core against other FPGA-based soft-core processors. The optimised IPPro delivers 1.4 - 3.0 times better f_{Max} compared to other processors. Comparing area utilisation numbers, IPPro has used 37% and 41% more FFs than GraphSoC and MicroBlaze but lower than FlexGrip. On the other hand, IPPro consumed $\approx$ 15% and 22% less LUTs than MicroBlaze and GraphSoC.

Section 6.3 has presented IPPro datapath optimisations to minimise execution overhead to implement multi-port actor and achieve essential data and control mechanisms to map and execute stream and non-stream data processing. The control of supported mechanisms is abstracted by developing C-APIs to maintain flexibility.

6.4 Multicore IPPro

The low-level communication and synchronisation mechanisms must be managed by the multicore architecture itself that created the need of a flexible *multicore interconnect*. It facilitates adaptability to exploit different dataflow transformations, provide flexible level of connectivity and essential data exchange patterns among cores to map parallel dataflow actors. It will help not only to map different pipelined dataflow graphs onto multicore architecture but also to exploit data and task parallel implementation adopting a horizontal and vertical scaling approach. Considering these architectural features the following design requirements are identified:

- Software controlled connectivity among cores of multicore IPPro to realise one-to-many, many-to-one, many-to-many consumer-producer data passing patterns, to have flexible actor-core mapping possibilities (Section 6.4.1).
- Dynamic routing of data streams among IPPro cores to achieve area-efficient horizontal and vertical scaling of the architecture (Section 6.4.1).

6.4.1 Exploration of multicore interconnect architecture

In a multicore architecture, the *multicore interconnect* defines connectivity across IPPro cores. In the open literature, the research community has proposed and analysed different types of interconnect architectures such as *bus, crossbar and network-on-chip (NoC)* [125]. Each interconnect architecture has pros and cons based on the supported connectivity, flexibility, area and performance [112], [125]. From an application mapping point-of-view, the chosen level of connectivity can limit data exchange possibilities among cores, leading to a restrained actor-core mapping and realising parallel possibilities. From a hardware design point-of-view, it could significantly impact performance and area utilisation.

This section discusses the dataflow data passing patterns and highlight their importance to achieve better actor-core mapping possibilities, and horizontal and vertical scaling. Besides, it presents the dynamic routing of the data streams approach to achieve flexible mapping possibilities onto multicore IPPro.

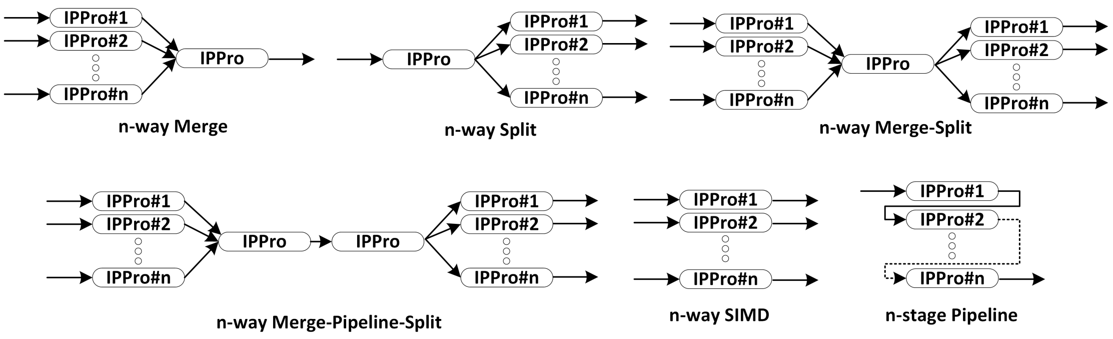


Figure 6.5: Theoretical mapping of data exchange patterns on IPPro cores.

Data passing patterns and core connectivity

Section 6.2 has emphasised the significance of supporting dataflow data passing patterns which must be supported by the multicore architecture to realise adaptable implementations. It includes multiple actor (many-to-one, one-to-many and many-to-many) data passing patterns (MPSC, SPMC, MPMC) [92], [93], [91].

Figure 6.5 models the required connectivity among cores that shall be supported by the multicore interconnect to map and execute different dataflow graphs. This architecture will provide both horizontal and vertical connectivity among cores which was absent in the 4x4 interconnect architecture. A *split* and *merge* can be expressed by SPMC and MPSC in producer-consumer model, or used to implement *data parallel* computation. Similarly, a *feed-forward* can be represented by SPSC in producer-consumer model, or used to achieve *pipelining* or *task parallel* computation. Since these patterns are reusable, different nested data passing patterns can be derived such as *merge-pipeline-split* or *split-pipeline-merge* as shown in Figure 3.3.

It shows that the multicore interconnect should support the identified data exchange patterns to improve actor mapping possibilities and maximise core utilisation for parallel implementations.

Dynamic routing of data streams

One of the set design requirement of multicore interconnect is the dynamic routing of data streams across multiple cores by sharing resources. It requires data-channel arbitration to avoid data collision, resource starvation and to ensure balanced bandwidth distribution across cores. For this purpose, a Xilinx *AXI4-Stream switch* IP is chosen as multicore interconnect that supports M x N crossbar connectivity between AXI-Stream master and slave channels. It uses an address control signal (TDEST) to route a stream of data between a master and a slave. It supports slave decoding and master arbitration mechanisms (fixed and round-robin) where each master is statically assigned a TDEST value.

Figure 6.6 illustrates the realisation of identified dataflow data passing patterns in Section 6.4.1 using TDEST signal. To maintain the balance between area utilisation and the level of connectivity among cores, the maximum support

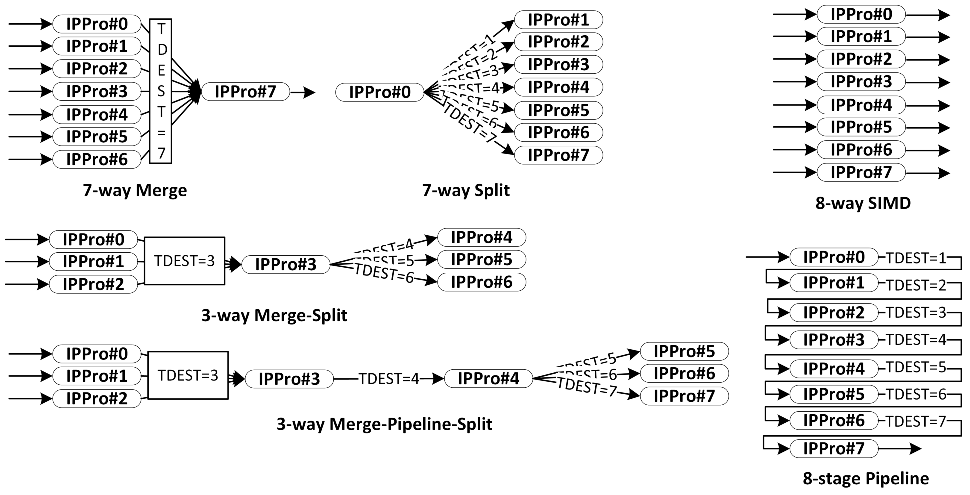


Figure 6.6: Realisation of data exchange patterns using stream interconnect.

of up to eight cores (IPPro#0 - IPPro#7) is considered as shown in Figure 6.6. This configuration would allow realising parallel implementations up to the 7-way split, 7-way merge, 8-way SIMD, 8-stage pipeline or a combination of thereof. The arrow shows the flow of data from producer to the consumer core. Section 6.3.1 has detailed the process of tagging tokens with *DEST_ID* whenever IPPro encounters PUSH CHANNEL# instruction. This tag specifies the destination core (consumer) and is used as TDEST.

This multicore interconnect architecture compliments, the features supported by IPPro and extend actor-core mapping possibilities using dynamic routing of data streams. The level of core connectivity supported by the interconnect defines the granularity of exploitable parallelism by the resultant multicore IPPro which in this case is 8-way SIMD.

6.4.2 Impact of interconnect's core connectivity and core utilisation on area and performance

Three designs have been selected using 4x4, 8x8 and 16x16 cross-bar configurations to accommodate 2, 4 and 8 IPPro cores as illustrated in Figure 6.7. These designs express an increasing level of core connectivity allowing better actor-core mapping possibilities by providing both horizontal and vertical connectivity necessary to realise tree expansion and reduction while maximising *core utilisation* (CU) as illustrated in Figure 6.6.

Each design has AXI4-stream master and slave interfaces (M_x and S_x). Half of the interfaces of each design, are assigned to the number of supported IPPro cores while remaining interfaces are used to feed data in and out of the multicore IPPro.

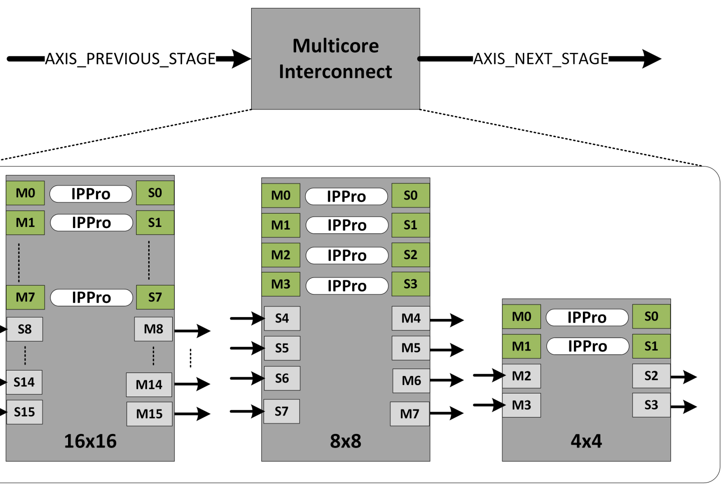


Figure 6.7: Stream interconnect architectures with increasing core connectivity.

Each input and output interface has an internal 32x24-bit FIFO realised using FPGA's LUT resources that buffer data locally to avoid congestion during channel arbitration while, the interconnect is serving other cores. The data payload of each channel is three bytes TDATA (2-bytes data token, 1-byte source/destination tag). The interconnect uses round-robin scheduling to avoid resource starvation and provide equal bandwidth to all cores. The size of TDEST has been fixed to 2, 3 and 4-bits for 4x4, 8x8 and 16x16 designs respectively to uniquely address each slave channel (input interface of IPPro core). The designs have been synthesised and implemented using Vivado v2016.4 for Artix-7 and Kintex-7 FPGA. The area and timing results are reported in Table 6.6.

Impact of scaling on the interconnect architecture Table 6.4 details area and CU of *stream interconnect* and compares it against *4x4 interconnect*. The *stream interconnect* provides a software controlled implementation of data passing patterns as illustrated in Figure 6.6.

Table 6.4 presents resource utilisation where both data parallel mappings have achieved 100% *core utilisation* (CU). The task parallel mappings of 4x4 interconnect have achieved 25% CU due to lack of vertical connectivity among cores. It shows that *stream interconnect* provides flexible actor-core mapping options to exploit both data and task parallelism using the same underlying architecture.

Table 6.5 presents the normalised area results of Table 6.4. The normalised FF and LUT utilisation is close to unity for *data parallel* implementations and consumes twice the number of BRAMs and DSP48E1s. On the other hand, a significant difference approx. 1.67 to 2.19 times in LUTs and FFs utilisation, is observed for *task parallel* implementations and four times number of BRAMs and DSP48E1s. The results show that the *stream interconnect* architecture is flexible, supports better actor-core mapping possibilities suitable for data and task parallel implementations, and area efficient than the *4x4 interconnect* architecture.

Performance Analysis Table 6.6 compares the implementation results of *stream interconnect* on Artix-7 and Kintex-7. Using Artix-7, 4x4 connectivity has resulted in f_{Max} 200 MHz, which reduced $\approx$ 1.33 and 1.66 times when connectivity is scaled-up to 8x8 and 16x16 respectively due to larger cross-bar connections implemented using multiplexers. When the same 4x4 connectivity is ported to Kintex-7, the design has achieved f_{Max} 285 MHz which is 1.09 and 1.29 times lower when scaled-up to 8x8 and 16x16 respectively. It can be observed that Kintex-7 delivered 1.45 times better timing than Artix-7 because, Kintex-7 FPGA technology is optimised for performance, which comes at higher chip cost.

It is important that the cores do not require full bandwidth of the interconnect as they sequentially process data and their execution time is directly proportional

Table 6.4: Implementation results to evaluate scaling of 4x4 and stream interconnect architectures on area and core utilisation to realise data (vertical) and task (horizontal) parallel implementations.

Impl.	4x4 Interconnect							Stream Interconnect						
	Cores	**Conn.**	**FFs**	**LUTs**	**BRAMs**	**DSP48E1**	**Core Util.**	**Cores**	**Conn.**	**FFs**	**LUTs**	**BRAMs**	**DSP48E1**	**Core Util.**
4-stage	16	4x4	15266	11991	16	16	4/16	4	4x4	6950	6434	4	4	4/4
8-stage	32	4x4	30528	23988	32	32	8/32	8	8x8	15338	14306	8	8	8/8
4-way	8	4x4	7616	5989	8	8	8/8	4	4x4	6950	6434	4	4	4/4
8-way	16	4x4	15232	11978	16	16	16/16	8	8x8	15338	14306	8	8	8/8

Table 6.5: Normalised area utilisation numbers of 4x4 with respect to stream interconnect realising parallel implementations.

Resource	Task Parallel		Data Parallel	
	4-stage	**8-stage**	**4-way**	**8-way**
FF	2.19	1.99	1.09	0.99
LUTs	1.86	1.67	0.93	0.83
BRAM	4	4	2	2
DSP48E1	4	4	2	2

Table 6.6: Implementation results of scaled-up stream interconnect designs with increasing core-connectivity on Artix-7 and Kintex-7 fabrics. The normalised area utilisation numbers of each design with respect to single-core IPPro are reported within the brackets.

Connectivity	FFs	LUTs	LUTRAM	f_{Max} (MHz)	
				Artix-7	**Kintex-7**
4x4	1524 (1.7)	1092 (1.9)	160 (0.9)	200	285
8x8	3414 (3.9)	2840 (5.0)	320 (1.7)	150	260
16x16	8266 (9.4)	8335 (14.6)	768 (4.2)	120	220

to the complexity of the actor. The implementation of a simple dataflow actor on IPPro requires atleast approx. 12 instructions. The *stream interconnect* arbitrates data channels and routes data from source to destination in a round-robin fashion on a cycle-to-cycle basis. Due to this reason, the bandwidth requirements per core is less than what is usually expected in a fully pipelined FPGA architectures (where a slower data transfer rate could limit the performance). Moreover, deployment of input/output FIFOs at interconnect boundaries allow data buffering and isolates clock boundaries which allow interconnect and IPPro cores to operate at different operating frequencies. Therefore, the operating frequency of the multicore interconnect ($f_{Interconnect}$) is not necessarily required equal to the operating frequency of the IPPro core (f_{IPPro}). Based on this fact, the maximum f_{Max} degradation of 1.66 and 1.29 times at the cost of flexible core connectivity among cores is a viable choice.

Area Analysis Table 6.6 reports the area utilisation of 4x4, 8x8 and 16x16 designs. The difference margin between LUTs and FFs of 5.00 and 3.19 is higher due to FIFO buffers realised using LUT resources. The normalised area utilisation of 4x4, 8x8 and 16x16 interconnect to single-core IPPro has been reported in Table 6.6. They consume 1.7, 3.9 and 9.4 times more FFs, and 1.9, 5.0 and 14.6 times LUTs respectively. This show that *stream interconnect* fulfils the identified requirements of the multicore interconnect identified in Section 6.4, and provides a balance between area and performance.

6.4.3 Multicore IPPro architecture

Considering the performance and area analysis results of multicore interconnect, the multicore IPPro is composed of eight IPPro cores, connected through 16x16 *stream interconnect* as shown in Figure 6.12. The AXI4-Lite interface allows to manage and AXI-MM to program dataflow actors onto IPPro cores. The interfaces (S8 - S15) and (M8 - M15) allow data in and out of the multicore IPPro. Depending on TDEST value, the incoming data stream is dynamically routed to the destination core realising multi-level *split*, *merge* implementations using the same underlying hardware architecture.

The interconnect interfaces (S0 - S7) and (M0 - M7) connected to the IPPro cores has 32x24 bits FIFO buffers. These buffers serve three purposes: 1) It temporarily stores data tokens produced/consumed by the cores which keep cores in processing due to data buffering. 2) It gives interconnect necessary time to arbitrate and route data streams among cores. 3) It isolates the clock domain boundaries allowing IPPro cores and multicore interconnect to run on independent clock frequencies. The buffering of data hides the data transfer time between cores by storing data tokens at input and output interfaces of the cores. It is possible to run IPPro cores (f_{IPPro}) at a maximum of 300 MHz while the multicore interconnect ($f_{Interconnect}$) can run up to 220 MHz which is 1.83 and 1.90 times higher compared to Artix-7 respectively as reported in Table 6.2.

6.4.4 Example: Mapping of dataflow graph onto multicore architecture

The chosen dataflow graphs cover the parallel and pipeline dataflow transformations. Consider an example dataflow graph composed of actors (A, B, C, D, E, F and G) as shown in Figure 6.8. The graph is decomposed such that A, D, E and F are mapped onto separate cores but, actor B, C and G require different data parallel granularity 3-way and 4-way SIMD to implement (B1, B2, B3), (C1, C2, C3) and (G1, G2, G3, G4) which needs *split* and *merge*. Figure 6.9 shows the mapping onto multicore IPPro, the interconnect interfaces used by each core are shown explicitly for a clear understanding of data execution flow. The dataflow graph is decomposed and mapped onto two multicore IPPro to demonstrate scalability and parallel implementation of actors.

A receives input data stream at M0 routed from input interface S8. The stream is processed by core#0 as defined by A and fed to B1, B2, B3 when encountering (PUSH Rx, 1, PUSH Rx, 2 and PUSH Rx, 2) instructions. Each core has a dedicated FIFO queue to receive tokens from other cores (Section 6.3.1

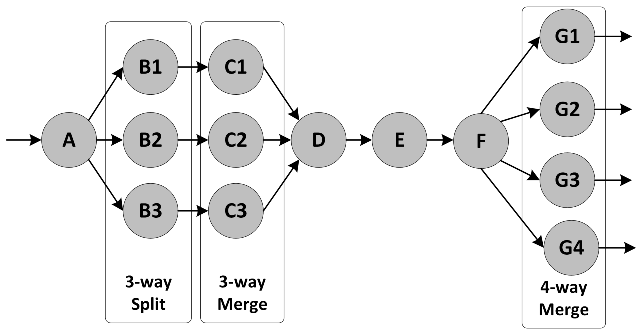
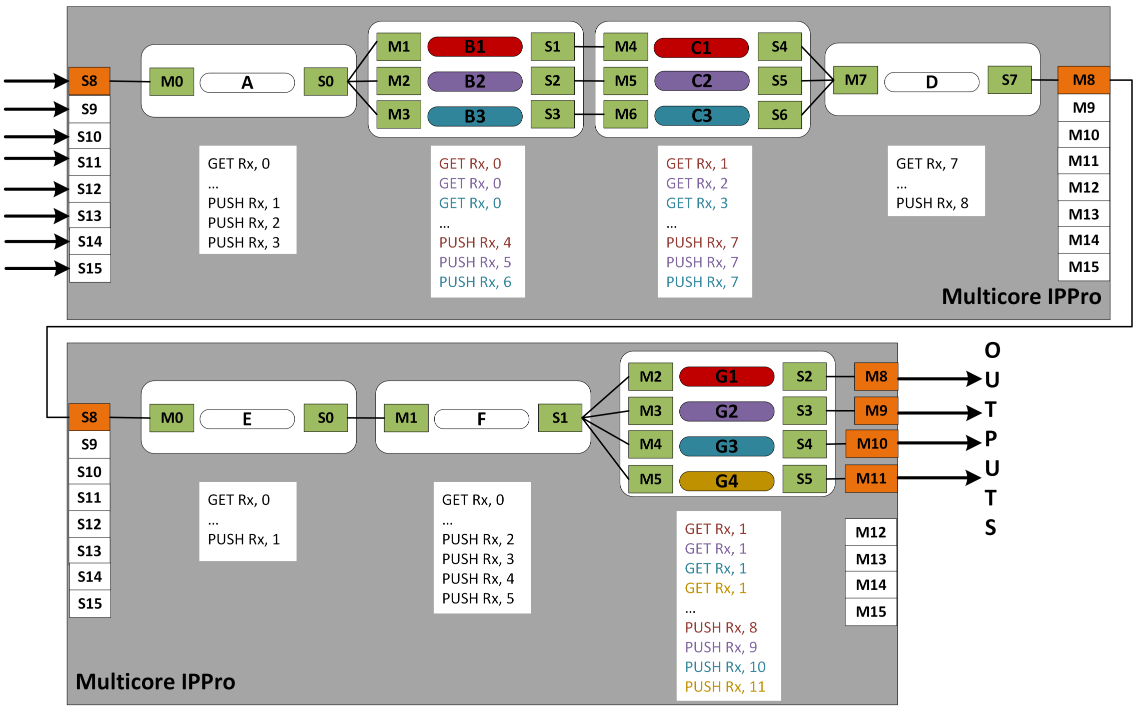


Figure 6.8: A dataflow graph example that covers pipelining of multiple data parallel actors.

and Figure 6.2) residing within multicore IPPro. B1, B2, B3 can concurrently read tokens (processed by A) into CHANNEL#0 of their respective FIFO queues using (GET Rx, 0). This process continues until D push the processed tokens to M8 output interface of the multicore interconnect. This interface is statically connected to S8 of the following multicore IPPro as indicated in Figure 6.9. Therefore, the tokens processed by D are received by E at S8, routed to M0 by the interconnect. The execution continue till reach the split (G1, G2, G3, G4) where the cores concurrently process the tokens and send processed tokens out of multicore IPPro using output interface (M8, M9, M10, M11).



```
GET Rx, 0
...
PUSH Rx, 1
PUSH Rx, 2
PUSH Rx, 3
```

```
GET Rx, 0
GET Rx, 0
GET Rx, 0
...
PUSH Rx, 4
PUSH Rx, 5
PUSH Rx, 6
```

```
GET Rx, 1
GET Rx, 2
GET Rx, 3
...
PUSH Rx, 7
PUSH Rx, 7
PUSH Rx, 7
```

```
GET Rx, 7
...
PUSH Rx, 8
```

```
GET Rx, 0
...
PUSH Rx, 1
```

```
GET Rx, 0
...
PUSH Rx, 2
PUSH Rx, 3
PUSH Rx, 4
PUSH Rx, 5
```

```
GET Rx, 1
GET Rx, 1
GET Rx, 1
GET Rx, 1
...
PUSH Rx, 8
PUSH Rx, 9
PUSH Rx, 10
PUSH Rx, 11
```

Figure 6.9: Flat illustration of mapping and execution of pipelined multiple data parallel actors exploiting parallelism using multicore IPPro. The listed IPPro code shows the read, write and tagging of tokens for each actor. These tags are used by the interconnect to route token among cores of the multicore IPPro.

6.5 FPGA-based programmable hardware acceleration platform

The data distribution and collection requirements depend on the application in-hand, and the adopted decomposition and mapping which are not known at design time [32], [40]. A flexible hardware-based data distribution and collection architecture is needed so the following design requirements are supported by the *system infrastructure* to parallel skeletons:

- *Split, compute and merge*, and *pipeline* skeleton require parallel streams which raises the need of parametrised distribution of multiple data streams.
- *Farm* skeleton require access to parallel data blocks which needs programmable distribution of data blocks into the cores scratchpad memories.

6.5.1 Parallel distribution and collection of data streams

Scatter-gather has widely adopted as a parallel data distribution and collection paradigm for regularly distributed data which makes it suitable for pixel processing [116], [126]. It uses static decomposition and divides data into multiple equal-sized blocks as illustrated in Figure 6.10 for parallel processing using multiple cores. In open literature, various image processing data distribution patterns driven by row, column and block-based static data decomposition are reported [32], [40], [117], [118]. However, these hardware architectures handle fixed image sizes and parallel distribution of streams. The software or application developer needs granular control on both stream generation and distribution using software APIs without dealing with low-level data and control mechanisms.

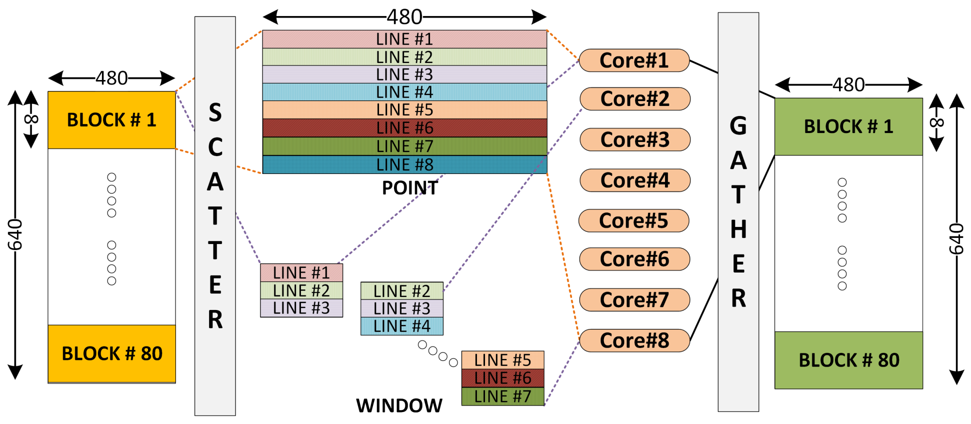


Figure 6.10: Parallel distribution of row-wise cyclic image pixels.

Besides, each parallel data stream can be converted into a form necessary for *point* and *area* processing.

Parallel point and window generation

Image pre-processing functions are composed of point and window/area operations. Figure 6.10 shows the row-wise scatter and gather process of an image with the maximum parallel granularity of eight for both operations. Compared to point operations, overlapping of multiple lines of pixels (two lines in case of a 3x3 window) is required for window operations. Therefore, dedicated software configurable point and area-based data distribution architecture has been proposed in Figure 6.11. The value of programmable P_A_REG register defines whether a stream or window of pixels is feeding to the core. Buffering of three incoming lines of pixels into LINE_BUFFER#1, LINE_BUFFER#2 and LINE_BUFFER#3 allows generation of 3x3 window. The *window controller* iteratively reads the line buffers and generates a stream of window pixels which can be fed to the cores of multicore IPPro through (S8 - S15) input interfaces as shown in Figure 6.12.

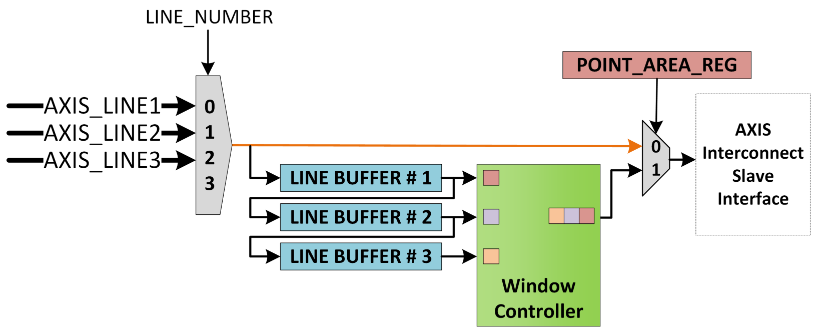


Figure 6.11: Generation and distribution of the point or window pixels.

This approach gives software control on data generation and mapping of point and area operations on IPPro cores. On the contrary, the HLS-based hardware architectures require code rewriting, verification, synthesis, place-and-route.

Configurable scattering and gathering of data streams

Software configurable *scatter* and *gather* hardware blocks have been designed with a FIFO interface to easily integrate with other image processing system [127]:

1. **Direct video streaming** An incoming video stream is stored into an on-chip frame buffer. A controller sequentially reads pixels from the frame buffer and stores into the data FIFO.

2. **Buffered video streaming** An incoming video stream is stored into an off-chip frame buffer. The host processor initiates a *direct-memory-access* (DMA) to read pixels from the frame buffer and stores into the data FIFO.

Both hardware blocks have AXI4-Lite registers to provide controllability on data distribution as shown in Figure 6.12 and listed in Table 6.7. The host processor must configure these registers during platform initialisation process. *Scatter* and *Gather* blocks have five and three programmable registers where

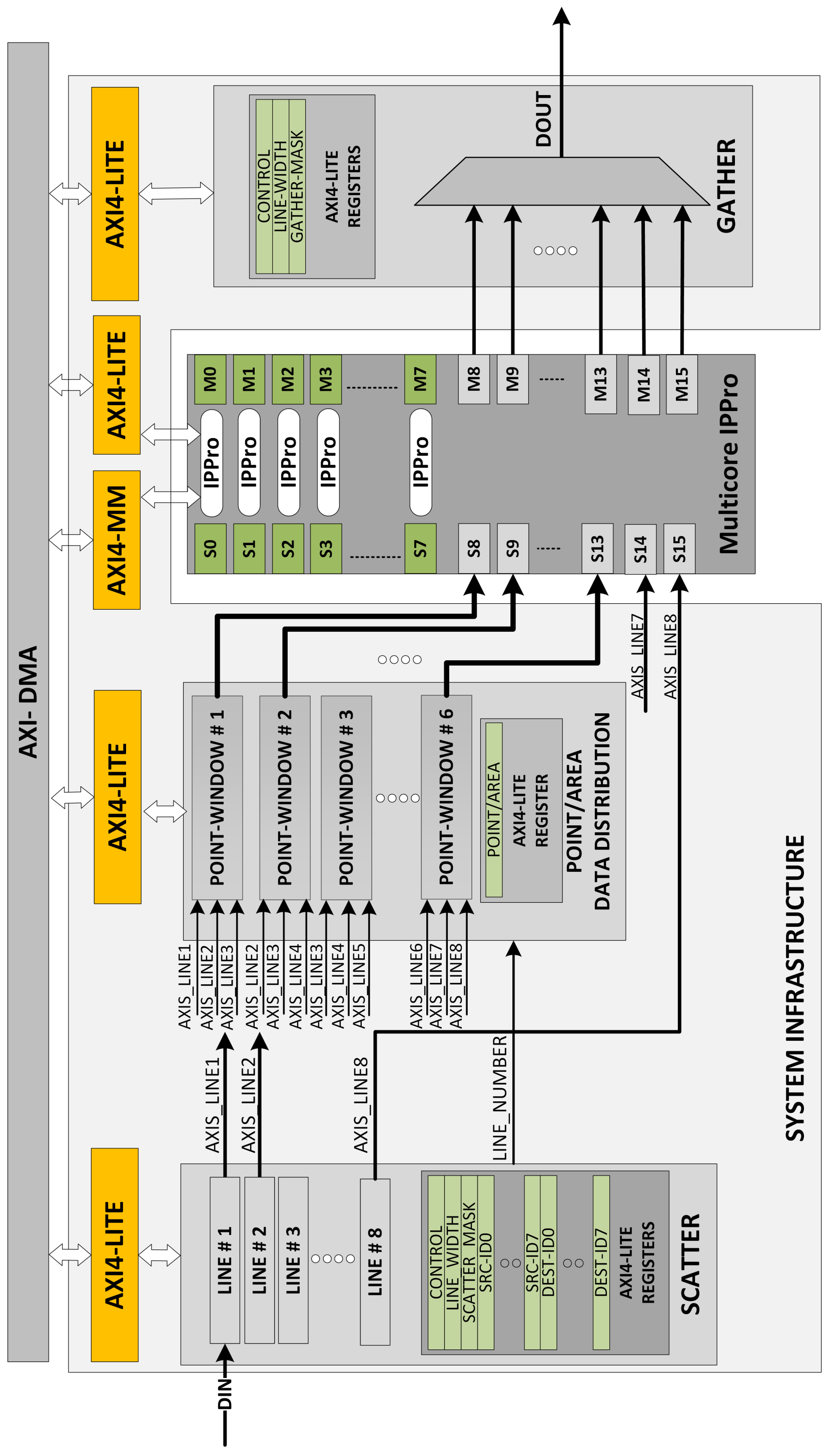


Figure 6.12: Block diagram of programmable hardware acceleration platform. The diagram only shows a single multicore IPPro due to space limitations. Cascading of multiple multicore IPPro cores is possible permited to FPGA area resources.

Table 6.7: The AXI4-Lite (control) register map of platform hardware modules.

AXI4-Lite Registers		**Bits**							
Addr.		31 - 28	27 - 24	23 - 20	19 - 16	15 - 12	11 - 8	7 - 4	3 - 0
Scatter Module									
0x00	CONTROL	xxx						PAREG	RST
0x04	SRC_ID_REG	LINE7	LINE6	LINE5	LINE4	LINE3	LINE2	LINE1	LINE0
0x08	DEST_ID_REG	LINE7	LINE6	LINE5	LINE4	LINE3	LINE2	LINE1	LINE0
0x0C	LINE_REG	xxx						LINE_WIDTH	
0x10	SCA_MASK	xxx						MASK	
Gather Module									
0x00	CONTROL	xxx							RST
0x04	LINE_REG	xxx						LINE_WIDTH	
0x08	GAT_MASK	xxx						MASK	

CONTROL, *LINE_REG* and *MASK* registers are common. *LINE_REG* defines the width of line buffers to support different image/video resolutions up to 2048, e.g. (640x480, 800x600). *MASK* defines data distribution granularity to generate parallel streams to exploit *data* parallelism. *SRC_ID_REG* and *DEST_ID_REG* register stores control tags (line buffer - FIFO queue) and (line buffer - core) mappings respectively. Listing 6.4 and 6.5 presents the C-APIs developed to configure scatter and gather blocks. These C-API hides the underlying implementation details of scatter and gather modules, provides transparent software interface (driver) that shall be used by the compiler framework to deploy different data and task parallel optimisations and hidden from the user.

Listing 6.4: C-APIs to manage scatter and gather blocks.

```
//-----------------------------------------------------------//
//       Split , compute and merge skeleton           //
//-----------------------------------------------------------//
// Scatter functions
int initScatter(Scatter* inst , uint32_t ScatterBase);
int ScatterWrite(Scatter* inst , ScatterAddr addr , uint32_t command);
// Gather functions
int initGather(Gather* inst , uint32_t GatherBase);
int GatherWrite(Gather* inst , GatherAddr addr , uint32_t command);
//-----------------------------------------------------------//
//                Farm skeleton                        //
//-----------------------------------------------------------//
// Scratchpad read/write functions
int writeSP(Core* inst , uint32_t *data , uint32_t n);
int readSP (Core* inst , uint32_t addr , uint32_t n);
```

Listing 6.5: Software controlled granularity of data distribution and collection functions of parallel streams.

```
// Set video horizontal resolution (640)
ScatterWrite(&Scatter, LINE_WIDTH, 640);
GatherWrite (&Gather , LINE_WIDTH, 640);
// One to one (line buffer - core) mapping
ScatterWrite (&Scatter, SRC_ID_REG, 0x000000);
// LINE#2 to CORE#2 Queue#1
ScatterWrite (&Scatter, SRC_ID_REG, 0x000010);
// Single-core, single active line buffer and no SIMD
ScatterWrite(&Scatter, DEST_ID_REG , 0x00000000);
ScatterWrite(&Scatter, SCATTER_MASK, 0x01);
GatherWrite (&Gather , GATHER_MASK , 0x01);
// Dual-core, 2-way SIMD
ScatterWrite(&Scatter, DEST_ID_REG , 0x00000010);
ScatterWrite(&Scatter, SCATTER_MASK, 0x03);
GatherWrite (&Gather , GATHER_MASK , 0x03);
// 3-way SIMD
ScatterWrite(&Scatter, DEST_ID_REG , 0x00000210);
ScatterWrite(&Scatter, SCATTER_MASK, 0x07);
GatherWrite (&Gather , GATHER_MASK , 0x07);
// 7-way SIMD
ScatterWrite(&Scatter, DEST_ID_REG , 0x06543210);
ScatterWrite(&Scatter, SCATTER_MASK, 0x7F);
GatherWrite (&Gather , GATHER_MASK , 0x07F);
// 8-way SIMD
ScatterWrite(&Scatter, DEST_ID_REG , 0x76543210);
ScatterWrite(&Scatter, SCATTER_MASK, 0xFF);
GatherWrite (&Gather , GATHER_MASK , 0xFF);
```

During execution, *scatter* block sequentially reads data stream from the input data FIFO, divides it into equal blocks (defined by $LINE\_WIDTH$), and consecutively stores into the line buffers (LINE_BUFFER#0 - LINE_BUFFER#7) depending on the SCA_MASK value. Each bit of SCA_MASK corresponds to the individual line buffer. The asserted bits specify that line buffer shall fill during *scatter*. Once the data is available into the line buffers, it is ready for consumption for *point* or *window* operation based on the value of $PAREG$ as discussed in section 6.5.1. The cores concurrently process data while *scatter* refills line buffers as soon as there is a space into the line buffers. The *gather* block reads a stream of processed pixels from output line buffers defined by the $LINE\_WIDTH$. GAT_MASK specifies how many output line buffers shall be read consecutively to reconstruct the output video stream. The C-APIs that pro-

Table 6.8: Area utilisation results of the system infrastructure.

Module	FFs	LUT	BRAM
Multicore Interconnect	9085	9965	0
AXI-Lite scatter control	576	1163	0
Scatter point only	594	980	8
Scatter point and window	2665	2300	20
AXI-Lite gather control	169	208	0
Gather	559	717	8
AXI-Interconnect	221	221	0
Reset processing system 1	48	30	0
Reset processing system 2	48	31	0

grammer shall use in the host application to adjust the underlying architecture depending on the requirements of the application. The user does not have to deal with underlying hardware mechanisms.

6.5.2 Implementation results

Table 6.8 presents the area results of *system infrastructure* implemented on Avnet Zedboard using Xilinx Vivado v2016.4. The multicore interconnect uses 10.27 and 13.19 times more FFs and LUTs than a single IPPro core and 1.28 and 1.64 times more FFs and LUTs than 8 IPPro cores. The cost of flexible multicore interconnect is close to the programmable pipelined implementation of eight dataflow actors. The AXI-Lite control modules consume 1.53 and 1.54 times fewer FFs and LUTs respectively than a single IPPro core. Thus, the cost of incorporating software-driven control and management is marginal.

The cost of scattering parallel windows (area) resulted in 4.48 and 2.34 times more FFs and LUTs compared to scattering parallel point lines. The impact of triple buffered line buffers to generate pixel windows is evident in the reported BRAM utilisation. A consistent area usage has been observed by point *scatter* and *gather*, as the process of scattering line buffers is similar to the gathering of processed pixels. The AXI-interconnect and reset processing system blocks

Table 6.9: Estimation of number of multicore IPPro on Xilinx Zynq MPSoCs.

Area Resources	**FF**	**LUT**	**BRAM**	**DSP48E1**	**# of multicore IPPro's (cores)**
Multicore IPPro	12279 (1)	10941 (1)	18.5 (1)	8 (1)	1 (8)
XC7Z045	343800 (28)	171900 (16)	545 (30)	900 (112)	16 (128)
XC7Z100	554800 (45)	277400 (25)	755 (41)	2020 (252)	25 (200)

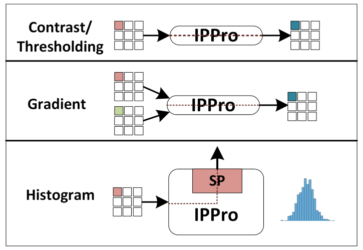
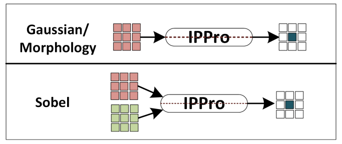
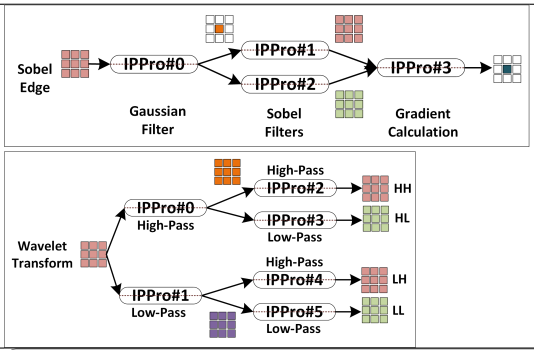
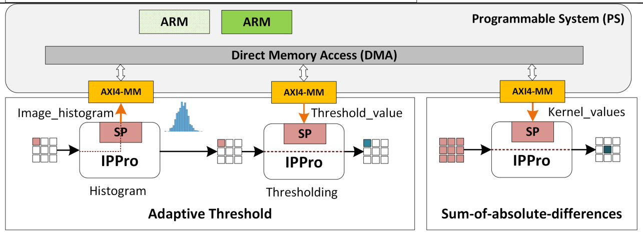
are mandatory system components. They allow to receive data from the host processor and route it to the addressed slave devices. *System infrastructure* has two clock domains (AXI4 bus and IPPro clock) which require two reset processing systems to ensure synchronous reset of the slaves. These are the costs of making *FPGA-based hardware acceleration platform* adaptable which abstracts the FPGA resources and improves design time by avoiding synthesis, place-and-route.

So far, it is considered that the proposed platform is composed of single multicore IPPro. But, Zynq Kintex-7 chips could accommodate more instances of multicore IPPro. Table 6.9 reports the available area resources of Zynq XC7Z045 and XC7Z100 chips, and the numbers normalised to single multicore IPPro are reported in the brackets. These normalised numbers give an estimate that Zynq chips could potentially accommodate up to $\approx$ 16 to 25 instances of multicore IPPro.

6.6 Parallel implementation of image pre-processing functions

Table 6.10 lists the mathematical representation of chosen functions that are fundamental kernels of larger algorithms and often represent the core computation of more extensive practical image processing applications [104], [105],

Table 6.10: Formal mathematical representation of chosen image pre-processing functions.

Cat.	Functions	Mathematical representation	Actor-core mapping
Point	• Contrast • Thresholding • Gradient calc. • Histogram	• $P_{(output)} = P_{input} + Contrast_{val}$ • $P_{(output)} = P_{input} > Threshold_{val} ? 255 : 0$ • $P_{(gradient)} = \|P_x\| + \|P_y\|$ • $Image_{(histogram)} = \sum_{i=0}^{n} Bin(P_i)$	 Contrast/ Thresholding IPPro Gradient IPPro Histogram SP IPPro
Area	• Gaussian • Sobel • Morphology	• $P_{(output)} = \sum_{i=1}^{9}(P_i * K_i)$ • $G_x = \begin{bmatrix} P_1 & P_2 & P_3 \\ P_4 & P_5 & P_6 \\ P_7 & P_8 & P_9 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$ • $P_{(erosion)} = \min_{i=1}^{9}(P_i * K_i)$ • $P_{(dilation)} = \max_{i=1}^{9}(P_i * K_i)$	
Task	• Sobel edge • Wavelet	• $P_{(gradient)} = \|SobelG_x\| + \|SobelG_y\|$ • $P_{(low-pass)} = \sum_{i=1}^{9}(P_i * K_i)$	 Sobel Edge IPPro#0 IPPro#1 IPPro#2 IPPro#3 Gaussian Filter Sobel Filters Gradient Calculation Wavelet Transform IPPro#0 High-Pass IPPro#1 Low-Pass High-Pass IPPro#2 HH IPPro#3 Low-Pass HL High-Pass IPPro#4 LH IPPro#5 Low-Pass LL
Hetero.	• Adaptive Threshold • Sum-of-absolute difference	• $P_{(SAD)} == \sum_{i=1}^{9} \|P_i - R_i\|$	 ARM ARM Programmable System (PS) Direct Memory Access (DMA) AXI4-MM AXI4-MM AXI4-MM Image_histogram SP IPPro Histogram Threshold_value SP IPPro Thresholding Adaptive Threshold Kernel_values SP IPPro Sum-of-absolute-differences

[106], [107], [108], [128]. The adopted actor-core mapping of each onto multi-core IPPro are detailed below:

Data parallel - balanced actor Point and area functions are individually mapped on the cores to realise 2 $\rightarrow$ 8-way data parallel implementations as shown in Table 6.10. This actor-core mapping impersonates *split, compute and merge* parallel skeleton as scatter-gather modules distribute and collect lines of pixels.

Task parallel - unbalanced actors They pipeline the point and area functions, where each core maps and executes separate actor as shown in Table 6.10. *Sobel edge* uses CORE#0-2 to perform area and CORE#3 to perform point operations. *Wavelet transform* uses six cores CORE#0-5 for pipelined implementation of area-based Gaussian low and high pass filters. Each core has window generation module as previously presented in Section 6.5.1.

Data parallel - heterogeneous computing The chosen *heterogeneous* functions demonstrate stream and non-stream computing possibilities necessary to realise the *farm* parallel skeleton on the proposed platform. *Adaptive threshold* requires image histogram to compute the new threshold value that involves floating-point calculation which is viable to be implemented on the host processor. The memory-mapped in and out execution paths of IPPro (Figure 6.3) have used to pass image histogram and receive new threshold value from the host processor as shown in Table 6.10. Similarly, for SAD implementation, during platform configuration, the host processor writes the 3x3 kernel value into the scratchpad memory of each IPPro. This kind of decomposition and execution impersonates realisation of *farm* parallel skeleton.

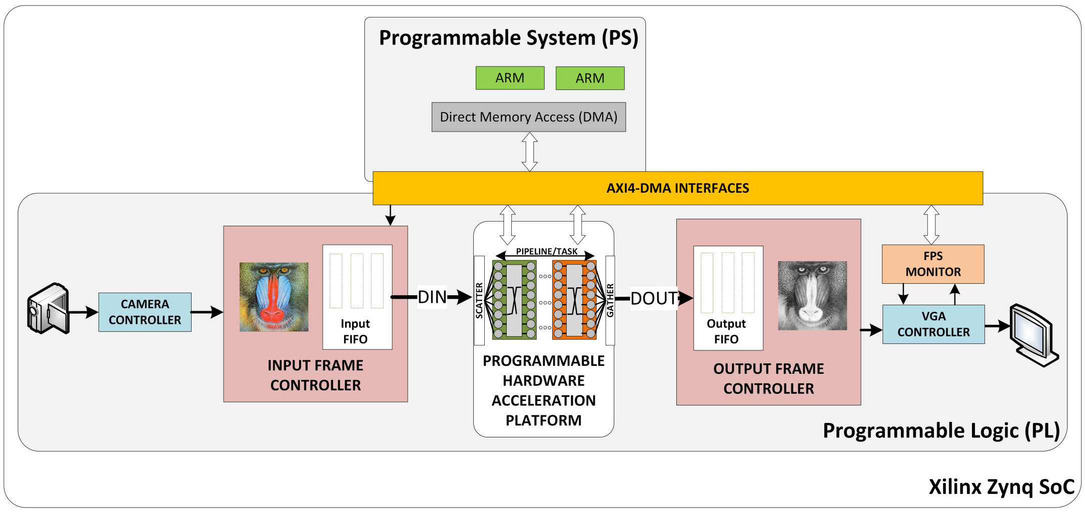


Figure 6.13: Video processing system architecture using FPGA-based programmable hardware acceleration platform.

The discussed functions have implemented on Avnet Zedboard that has Xilinx Zynq SoC (XC7Z020-CLG484-1). Figure 6.13 shows the simplified block diagram of the realised video processing system which is similar to previously presented in Chapter 4, except the middle processing block has replaced with *programmable hardware acceleration platform* as shown in Figure 6.13. A *FPS Monitor* module has been implemented in FPGA logic to measures a time between start and end of frame to calculate the achieved VGA (640x480) frame processing time in frames/second (fps).

6.6.1 Performance analysis

Point functions Table 6.11 reports the acceleration results of point functions using multicore IPPro on Avnet Zedboard. The single-core results affirm a direct relationship between the average cycles/pixel and actor's execution time, which signifies that smaller (decomposed) dataflow delivers better performance. Both *gradient* and *threshold* are data dependent functions and require branch instructions compared to data independent *histogram* and *contrast* functions. This is evident in the reported results as *histogram* and *contrast* have achieved 1.18 and 1.90 times better performance over *threshold* and *gradient* due to lack of branch executions leading to fixed execution time/pixel.

The *point* functions have been implemented with increasing data parallel granularity from 2 $\rightarrow$ 8-way SIMD using 2 - 8 cores are reported in Table 6.11 to analyse the performance improvements. It has achieved a maximum of 7.8 times improvement over single-core implementation because of direct streaming video pipeline which avoided host-to-accelerator data transfer times and achieved a maximum of 75 and 149 fps for *gradient* and *contrast.*

Area functions Table 6.11 reports the acceleration results of area functions on Avnet Zedboard. In contrast to the point, all three functions are data independent, *Morphology* uses *min* and *max* instructions to compute *dilate* and *erode* image operations. As *min* and *max* do not support dataforwarding, they have taken more execution time than *Gaussian* and *Sobel.* Implementation of both *Gaussian* and *Sobel* filter has taken advantage of single-cycle multiply-accumulate, moreover, the zero kernel values has further optimised *Sobel* allowed to save four clock cycles per pixel more than *Gaussian.* Therefore, *Sobel* has achieved 1.12 and 1.20 times better performance over *Morphology* and *Gaussian* filters.

Each function has implemented with increasing data parallel granularity from 2 $\rightarrow$ 6-way using 2 - 6 cores and the results are reported in Table 6.11. It achieved a maximum performance of 5.27 times which is 2.53 times less than *point* due to parallel scattering of windows. The direct streaming video pipeline delivered

Table 6.11: Data parallel performance results of point and area functions using IPPro on Artix-7 (Zedboard).

Functions	**Point**				**Point**		
	Contrast	**Threshold**	**Gradient**	**Histogram**	**Gaussian**	**Sobel**	**Morphology**
Avg. Cycles/Pixel	37	49	56	33	39	35	44
Execution time (ms)	53	61	100	52	61	57	66
Performance	**Frames per second (fps)**						
Single-core	19	16	10	19	16	18	15
2-way	36	30	18	36	28	29	24
3-way	54	45	27	55	41	44	38
4-way	73	61	37	73	58	60	54
5-way	92	76	46	91	74	78	68
6-way	111	92	56	109	91	95	84
7-way	129	108	65	127	-	-	-
8-way	149	123	75	145	-	-	-

Table 6.12: Comparison of data parallel implementation of point functions using IPPro against ARM (-O2,-O3).

Point Functions	**Contrast**			**Threshold**			**Gradient**			**Histogram**		
Architecture	IPPro	ARM		IPPro	ARM		IPPro	ARM		IPPro	ARM	
Optimisation		-O2	-O3		-O2	-O3		-O2	-O3		-O2	-O3
Exec. time (ms)	33.20	45.90	45.71	38.30	45.69	45.67	62.20	49.41	48.47	33.20	53.96	49.44
Performance	**Frames per second (fps)**											
Single-core	30	21.78	21.87	26	21.88	21.89	16	20.23	20.62	30	18.52	20.22
2-way	57	-	-	48	-	-	29	-	-	57	-	-
3-way	85	-	-	73	-	-	43	-	-	87	-	-
4-way	116	-	-	99	-	-	59	-	-	117	-	-
5-way	145	-	-	123	-	-	74	-	-	145	-	-
6-way	175	-	-	149	-	-	90	-	-	174	-	-
7-way	204	-	-	175	-	-	104	-	-	203	-	-
8-way	235	-	-	200	-	-	120	-	-	232	-	-

Table 6.13: Comparison of data parallel implementation of area functions using IPPro against ARM (-O2,-O3).

Area Functions	**Gaussian**			**Sobel**			**Morphology**		
Architecture	IPPro	ARM		IPPro	ARM		IPPro	ARM	
Optimisation		-O2	-O3		-O2	-O3		-O2	-O3
Exec. time (ms)	61.00	71.61	70.69	57.00	66.16	65.03	66.00	66.56	64.74
Performance	**Frames per second (fps)**								
Single-core	16	13.96	14.14	18	15.11	15.37	15	15.02	15.44
2-way	28	-	-	29	-	-	24	-	-
3-way	41	-	-	44	-	-	38	-	-
4-way	58	-	-	60	-	-	54	-	-
5-way	74	-	-	78	-	-	68	-	-
6-way	91	-	-	95	-	-	84	-	-
7-way	-	-	-	-	-	-	-	-	-
8-way	-	-	-	-	-	-	-	-	-

approx. 84 and 95 fps for *Morphology* and *Sobel* respectively using six cores.

By porting the platform to Kintex-7 fabric as reported in Table 6.12 and Table 6.13, further improvements up to ≈ 1.60 times is possible for both *point* and *area* functions due to higher operating frequency of IPPro cores and the multicore interconnect at 300 MHz and 220 MHz respectively (Table 6.3 and Table 6.6).

Performance comparison of IPPro against embedded ARM Cortex-A9 CPU implementation To set the baseline figures and compare the IPPro performance, both *point* and *area* image processing functions has been implemented on embedded ARM Cortex-A9 CPU operating at 667 MHz. Two compiler optimisations -O2 (high) and -O3 (maximum) have been used which are supported by the ARM GCC compiler available in *Xilinx Vivado Software Development Kit* (SDK).

The detailed results are reported in Table 6.12 and Table 6.13 respectively. For *point* and *area* functions, the average performance of 20 and 14 fps have been achieved irrespective of the fact that ARM CPU operates at 2.23 times faster than IPPro core. The performance is limited due to the fact that ARM uses AXI4-DMA to read and write pixels which takes maximum 40 ms data transfer time for a 640x480 video frame configured as maximum burst size of 256x32 bits per DMA transfer. By exploiting ARM compiler optimisations from -O2 to -O3, the maximum performance improvement (excluding the data transfer times) of 1.47 times has been observed. However, this performance improvement become insignificant, as fixed data transfer overhead is ≈ 5.4 times larger than the function's processing time which limits the best achievable theoretical performance

Table 6.14: Implementation results of HLS generated IPs on Kintex-7 fabric. (Normalised area and performance results of multicore IPPro to HLS).

IP	Operations	Freq. (MHz)	Area Utilisation				Exec. (ms)	fps
			FF	LUT	BRAM	DSP48E1		
Point	*Add, Subtract* *And, Or, Xor* *Mul, Min, Max*	250	1526 (8.04)	1266 (8.64)	0 (18.5)	2 (4)	2.36	423 (0.55)
Area	*Convolution* *Morphology*	222	5444 (2.08)	3350 (2.94)	2.5 (13)	18 (0.33)	2.76	361 (0.43)

up to 25 fps.

By comparing the obtained ARM CPU results against IPPro, the single-core IPPro implementation achieved maximum of 1.48 times better performance over ARM which is operating at 2.23 times lower operating frequency. Because, IPPro exploits stream processing and avoids reading and writing data transfer overheads. In addition, the data parallel IPPro implementations achieved further performance improvements of 11.47 and 6.43 times using 8 and 6 cores for *point* and *area* functions respectively.

Cost analysis of proposed adaptable approach against HLS The *point* and *area* IPs have been developed by Deng (a research student) using Xilinx Vivado HLS v2016.4. Both IPs have AXI4-Lite interface to select operations as listed in Table 6.14 and fully pipelined. The *area* is 1.12 times slower than point due to necessary line buffering that reduced performance from 423 and 361 fps. The *area* used 3.56 and 2.64 more FFs and LUTs compared to the *point*.

Table 6.14 also presents the normalised performance and area numbers. The IPPro implementation is 1.8 and 2.36 times slower than HLS developed IPs, at the cost of 8.04 and 8.64 times more FFs and LUTs respectively. This increase come at the cost of flexible and programmable architecture that not only allows software programmer to map and execute multiple dataflow actors. But also, provide software controlled granularity to exploit desired data and task parallel implementations using skeletons. This area cost is narrow down to ≈ 2.08 and 2.94 times for *area* IP and the memory utilisation gap is reduced approx. by 1.43 times. This performance gap can be reduced using multiple multicore IPPro as estimated and reported in Table 6.9.

Pipelining multiple tasks Table 6.15 reports the performance results of pipelining multiple dataflow actors exploiting task parallelism. The *Wavelet transform* consists of pipelined execution of balanced area actors, i.e. (high and low-pass filter) as illustrated in Table 6.10 while, *Sobel edge* represents pipelined execution of unbalanced gradient actor.

During execution of *Wavelet Transform*, the first-stage cores pass processed pixels to the second-stage cores. As the actors are balanced, no ripple-effect has been observed as balanced execution hides the data transfer times to second stage cores. Therefore, the average cycles/pixel is close to the *Gaussian*. Though the computation requirement of the *Wavelet transform* is six times more than the *Gaussian*, similar performance has achieved exploiting task parallelism which can further be improved by exploiting data parallelism.

In case of *Sobel Edge*, the *gradient* takes average of 56 cycles/pixel due to data dependent operations which is 1.43 and 1.60 times higher than *Gaussian* and the

Table 6.15: Performance results of task parallel implementations of multiple dataflow actors on multicore IPPro.

Pipelined Tasks	**Artix-7 (Zedboard)**		**Kintex-7 (ZC706)**	
	Wavelet	**Sobel Edge**	**Wavelet**	**Sobel Edge**
No. of cores	6	4	6	4
Avg. Cycles/Pixel	38	64	38	64
Execution Time (ms)	62	133	38	82
Performance (fps)	16	8	25	12

Table 6.16: Performance results of heterogeneous decomposed compute functions using multicore IPPro.

Heterogeneous Operations	Artix-7 (Zedboard)		Kintex-7 (ZC706)	
	Adaptive threshold	SAD	Adaptive threshold	SAD
Avg. Cycles/Pixel	49	260	49	260
Execution time	61	515	38	319
Performance	**Frame per second (fps)**			
Single-core	16	2	26	3
2-way	30	3	48	4
3-way	45	5	73	6
4-way	61	7	99	8
5-way	76	9	123	10
6-way	92	11	149	12
7-way	108	-	175	-
8-way	123	-	200	-

Sobel as reported in Table 6.11. Therefore, during execution the *gradient* stage forces a backward ripple effect which propagates to *Sobel* and *Gaussian* limiting overall performance to 8 fps. The pipelined implementation of unbalanced actors delivered 2.03 times improvement over non-pipelined implementation which suggests that decomposition of dataflow graph into balanced actors is vital to gain maximum advantage of task parallelism.

Heterogeneous computing tasks Table 6.16 presents the results of heterogeneously decomposed *Adaptive Threshold* and *Sum of absolute difference* (SAD) functions illustrated in Table 6.10. The SAD takes 8.44 times more time than *adaptive threshold* due to a nested execution of data dependent branch instructions necessary to compute absolute values of G_x and G_y produced by *Sobel filter*. The data parallel implementation of SAD has achieved maximum improvement of 5.50 times using six cores of multicore IPPro.

In *Adaptive threshold*, the host processor takes 10μs and 25μs to read the image histogram bins from the scratchpad memory and compute the new threshold value respectively. Since the execution of the host processor and the multicore IPPro are concurrent, a maximum performance improvement of 7.6 times has achieved using eight cores which can be improvement by 1.60 times using Kintex-7.

6.7 Summary

This chapter presented an *FPGA-based programmable hardware acceleration platform* that supports a software-controlled implementation of parallel skeletons. The platform provides three layers of software programming abstractions to the software and algorithm developers. Each layer complements the adaptable features supported by the following layer. These layers allow to explore, optimise, map and implement parallel dataflow applications onto the FPGA using IPPro core, multicore IPPro and system infrastructure. The platform enables deploying software-centric *edit-compile-run* flow that improves design time.

The IPPro core sits at the bottom layer implementing a programmable dataflow actor, the multicore IPPro lies in the middle implementing programmable multiple dataflow actors. Middle layer supports producer-consumer data exchange patterns to explore and exploit parallelism. The top layer provides software controlled data distribution and collection mechanisms necessary to support the functional requirements of bottom layers.

The implementation results show that platform's adaptability and flexibility come at the cost of area where, significant amount is consumed by the interconnect followed by software-controlled distribution of window of pixels. The interconnect used 10.27 and 13.19 times more FFs and LUTs than a single IPPro core, and 1.28 and 1.64 times more FFs and LUTs than eight IPPro cores. Similarly, the scattering of pixels consumed ≈ 3 times FFs and LUTs than a single IPPro core.

The platform operates in two separate clock domains and the maximum f_{IPPro} and $f_{Interconnect}$ are 300 MHz and 220 MHz respectively.

A set of *point* and *area* image pre-processing functions are implemented on the platform using Avnet Zedboard (Artix-7), to evaluate and analyse the flexibility and performance. The decomposition and mapping possibilities cover acceleration of balanced and unbalanced actors exploiting both data and task parallelism. The implementation results show that data independent functions deliver better performance over data dependent functions because of the non-linearity introduced by branches. The *point* functions maps better on the platform and provides $\approx$ 2.53 times better acceleration than *area* functions, due to the absence of line buffering mandatory to obtain window of pixels. It can further improve by realising data parallel implementation which can deliver a maximum of 7.80 and 5.27 times for *point* and *area* functions. Comparison of results with embedded ARM Cortex-A9 CPU shows that single-core IPPro has achieved maximum of 10 times better performance while operating at 2.23 times less frequency by avoiding data transfer overheads. In addition, by exploiting data parallelism maximum performance improvements of 11.47 and 6.44 times using 8 and 6 cores for point and area functions respectively over ARM CPU.

The results of pipelined execution show that balanced actors implementation had achieved maximum performance, as they hide the data transfer and processing time of the following stages. In case of unbalanced actors, the maximum achievable performance is limited by the slowest actor due to the ripple effect. These results suggest that it is essential to decompose the dataflow graph into balanced actors to achieve maximum benefit of task parallelism and avoid the ripple effect.

Chapter 7

Conclusion and Future Work

7.1 Summary

FPGAs have not accepted as mainstream computing platform due to longer design times and need of specialist programming tools which can be challenging for use by algorithm and software developers. As existing FPGA-based design approaches struggle to approach the discussed challenges while providing a balance between adaptability and performance, this thesis has proposed an *FPGA-based programmable hardware acceleration platform* architecture implementing different image pre-processing applications. It is maintained that the approach offers a balance between performance and efficient resource utilisation by reducing design time. The platform can be programmed using conventional software development approaches. It enables software and algorithm developers to accelerate applications on an FPGA using edit-compile-run flow rather than time-consuming synthesis, place-and-route design flow, thus reducing design time.

The major architectural challenge has been to find a balance between the supported hardware and software abstraction while maintaining the concurrency and pipelining benefits of the FPGA technology. A hierarchical hardware and software abstraction layers have been used to achieve flexibility where each layer provides unique features that allow hardware platform to implement different high-level application descriptions down to low-level FPGA resources. This allows fine-and coarse-grained mapping and exploitation of data and task parallel realisations on the platform.

7.2 Thesis Contributions

This work has presented an approach to make FPGA-based hardware acceleration easier for software and algorithm developers using software-centric edit-compile-run flow with reduced design time.

1. Design and development of novel FPGA-based *Image Processing Processor* (IPPro) soft-core architecture tailored for acceleration of image processing applications. The architecture has been carefully designed to allow the functional computing requirements to be supported and FPGA compute and memory resources to be efficiently utilised. It comprises a 16-bit signed, 5-stage pipelined RISC processor that supports basic arithmetic, logical and branch instructions with dataforwarding that implements data dependent *point* and *area* image processing operations. It is then used as a basic programmable processing element of the proposed *FPGA-based hardware acceleration platform.* The IPPro operates maximum at 300 MHz and delivers up to three times better raw-computation considering the operating frequency over other soft-core processor architectures by exploiting dedicated DSP block and minimises use of FPGA resources. Results show that the IPPro has achieved up to 5.8 times better performance by util-

ising approximately same amount of FPGA resources compared to other FPGA-based programmable architecture. In addition, comparison of chosen micro-benchmarks shows that IPPro has achieved up to 8.94 times better performance over well established MicroBlaze soft-core processor and consumes fewer resources.

2. The processing capabilities of the IPPro datapath has been extended beyond supported purely by the dedicated DSP48E1 block. Specialised min, max and coprocessor instructions are included in the datapath where coprocessor extension allows complex arithmetic operations to be off-loaded to the coprocessor. The coprocessor executes in parallel and does not stall the execution of IPPro datapath to maximise performance. This optimisation has increased the length of the critical path which reduced the maximum operating frequency of the datapath by 11% and consumed 89 LUTs, 34 FFs.

3. Creation of the IPPro as an independent, self-managed, programmable dataflow accelerator that receives tokens from multiple producers and sends the processed token to multiple consumers by executing stream instructions. The architecture supports fine-and coarse-grained mapping and execution of dataflow nodes using producer-consumer computing model. The actor firing rule is software programmable as the IPPro code consists of both actor's functional description and control (firing rule). It avoids the need for an external controller, token re-ordering and synchronisation mechanisms and which are necessary for *high-level synthesis* (HLS) and HDL-based design approaches. In addition, stream instructions based on data and control

mechanisms avoid data-transfer overheads and simplify multicore synchronisation problems by avoiding the intervention of both the host processor and communication controller.

4. The IPPro datapath supports both message-passing and shared memory data models which allows for processing of both uniform and non-uniform distributed data or combinations thereof. These data processing paths allow the IPPro to implement split, compute, merge, pipeline and farm parallel computing skeletons on the FPGA. It facilitates better programming abstraction which can be used to explore, profile, optimise and evaluate different mapping possibilities and deploy them on the underlying architecture using software-centric edit-compile-run flow to find a suitable solution to the problem.

5. Creation of a multicore IPPro architecture that allows mapping and execution of multiple dataflow actors using dynamic routing of data streams among IPPro cores. The architecture facilitates both data and control mechanisms supported by the underlying IPPro cores. It uses the stream routing information issued by the producer core to forward data tokens to the consumer cores. The supported data passing patterns are one-to-many, many-to-one, many-to-many which are essential to map tree reduction and expansion structures effectively. This flexible connectivity among cores enables the adaptable realisation of a pipelined dataflow graph exploiting task parallelism, vertical scaling of a dataflow actor to exploit data parallelism or combinations thereof in order to maximise resource re-use. It enables a wide range of application profiling, exploration and optimisation options

for the user. It comes at the cost of 10.27 and 13.19 times more FFs and LUTs than a single IPPro core. The multicore IPPro has two separate clock domains, where maximum frequencies of the IPPro and interconnection are 300 MHz and 220 MHz respectively.

6. Implementation of *k*-means algorithm using multiple IPPro cores on Avnet Zedboard has been achieved which allows exploration of actor-core mapping possibilities and their evaluation on area, performance, power and resource efficiency. Four IPPro-based hardware accelerator designs composed of single, dual, 8-way-SIMD and dual 8-way-SIMD cores have been realised. The results have been compared against equivalent HLS and GPU implementations. The results shows that up to 7.3 times performance improvements over single-core is possible by exploiting both data and task parallelism at the cost of increased area. Comparing against other technologies, FPGA achieved 27 times better performance over the embedded CPU by exploiting parallelism and consumes 4.9 times less power than the embedded GPU. Moreover, the power efficiency (fps/W) numbers shows that FPGA implementation is 57 and 24 times more power efficient than embedded CPU and GPU respectively.

7. *Point* and *area* image pre-processing functions are implemented on Avnet Zedboard to evaluate performance and analyse flexibility of the platform. The selected decomposition and mapping possibilities cover acceleration of both balanced and unbalanced, data independent and dependent dataflow actors exploiting data and task parallel implementations. They exhibit implementation of a split, compute, merge, pipeline and farm parallel skeletons

on FPGA technology. The results show that data independent functions deliver better performance over data dependent functions because of the non-linearity introduced by branches. Comparison of results with embedded ARM Cortex-A9 CPU shows that single-core IPPro has achieved maximum of 10 times better performance while operating at 2.23 times less frequency by avoiding data transfer overheads. In addition, by exploiting data parallelism maximum performance improvements of 11.47 and 6.44 times using 8 and 6 cores for point and area functions respectively over ARM CPU.

7.3 Suggestions for further work

The presented work was intended to propose a novel *FPGA-based programmable hardware acceleration platform soft processor* that is adaptable, and facilitates fast-prototyping and exploration possibilities for software and algorithm developers using software-centric edit-compile-run flow. Some suggested future directions to extend this work:

1. **Extension IPPro datapath to support execution of dynamic dataflow graphs** where an actor can produce and consume the different number of tokens in each firing. One possible solution is to extend the data payload (ACTION_TRIGGER, SRC_ID, DEST_ID, DATA) and include additional instruction similar to TEST to decode control information generated by preceding actor nodes.

2. **Syntactic extension of high-level programming language** to effectively exploit the underlying supported parallel skeletons. It can be an extension of well-established high-level languages such as OpenCL and OpenMP.

3. **Software-based profiling framework** that uses static analysis techniques to profile the execution and interaction among actors of the dataflow application. This profiling information can be used to optimise different processing stages. It shall also provide data dependent analysis capability to profile and analyse the impact of control and data dependent dataflow nodes on the performance. This can be achieved by determining their computational load, data transfers and storage load. The computational load can be determined by recording the execution of control statements. Data-transfer and storage load can be determined by the rate of token production/consumption and inter-stage buffer utilisation.

4. **Extending the IPPro platform to modern FPGA architecture** such as Xilinx Znyq UltraScale+ MPSoC. The *programmable hardware acceleration platform* uses IPPro softcore processor to implement different applications that exploits hardened DSP block. In terms of performance, the modern DSP48E2 block in the Zynq UltraScale+ delivers $\approx 16\%$ better timing (f_{Max}) compared to DSP48E1 block and provides $\approx 20\%$ more blocks. This would allow the IPPro cores not only to operate at higher operating frequency (improving raw-computation capacity) but also the possibility to accommodate more IPPro cores within the FPGA fabric. A high density UltraRAM has been introduced in the Zynq UltraScale+ memory hierarchy to extend the on-chip memory capabilities. It enables up to 500Mb of total on-chip storage which is equivalent to a 6 times increase in on-chip memory resource compared to Zynq-7000. It is a dual-port synchronous memory block similar to the dual-port true Block RAM with higher memory density.

The scratchpad memory in the IPPro architecture can be realised using UltraRAM that would allow to store/buffer large images and implement global image processing functions within the FPGA fabric. In terms of power, the Zynq UltraScale+ provides 3.5 times better performance/Watt compared to Zynq-7000 MPSoC. It supports clock gating, frequency scaling and ability to assign different computational units into multiple power domains, i.e. (Full power, Low power, Battery power). These features gives better power optimisation opportunities to the designer to realise power optimised domain specific applications.

Appendix A

Author's Publications

1. F. Siddiqui, M. Russell, B. Bardak, R. Woods & K. Rafferty, *"IPPro: FPGA based Image Processing Processor"*, in *Proceedings of IEEE International Workshop on Signal Processing Systems (SiPS)*, Belfast, United Kingdom, 21-22 Oct,2014. (Published)

2. C. Kelly, F. Siddiqui, B. Bardak & R. Woods, *"Histogram of Oriented Gradients front end processing: an FPGA Based Processor Approach"*, in *Proceedings of IEEE International Workshop on Signal Processing Systems (SiPS)*, Belfast, United Kingdom, 21-22 Oct, 2014. (Published)

3. B. Bardak, F. Siddiqui, C. Kelly & R. Woods, *"Dataflow toolset for soft-core processors on FPGA for image processing applications"*, in *Proceedings of 28th IEEE Asilomar conference on Signal, Systems and Computers*, Asilomar, USA, 2-5 Nov, 2014. (Published)

4. C. Kelly, F. Siddiqui, B. Bardak, Yun Wu & R. Woods, *"FPGA Soft-core Processors, Compiler and Hardware Optimisations validated using HOG"*, in *Proceedings of 12th International Symposium on Applied Reconfigurable Computing (ARC)*, Rio de Janeiro, Brazil, 22-24 Mar, 2016. (Published)

5. M. Amiri, F. Siddiqui, C. Kelly, R. Rafferty, R. Woods & B. Bardak, *"FPGA-based soft-core processors for image processing applications"*, in *Journal of VLSI Signal Processing (JVSP)*, 2016, vol 87, no. 1, pp. 139-156. (Published)

6. T. Deng, F. Siddiqui, D. Crookes & R. Woods, *"Accelerating Image Algorithm Development using Soft Co-Processors on FPGAs"*, in *Proceedings of 29th IEEE Irish Signals and Systems Conference (ISSC)*, Belfast, United Kingdom, 21-22 Jun, 2018. (Published)

Appendix B

IPPro: Technical details

Implementation of DSP48E1-based ALU

The IPPro datapath uses dedicated DSP48E1 block to implement arithmetic and logical instructions. The DSP48E1 can be dynamically configured using OPMODE, INMODE and ALUMODE control and CEA2, CEB2, CEC, CEM and CEP pipelined registers. Table B.1 shows the detailed configuration use by the IPPro.

Table B.1: IPPro supported instruction set and their corresponding DSP48E1 control signals.

Instruction	INMODE	OPMODE	ALUMODE	CEA2	CEB2	CEC	CEM	CEP
Add	00000	110011	0000	1	1	1	0	1
Sub, Min, Max	00000	110011	0011	1	1	1	0	1
Mul	10001	000101	0000	0	0	1	1	1
Muladd	10001	110101	0000	0	0	1	1	1
Mulsub	10001	110101	0011	0	0	1	1	1
Mulacc	10001	100101	0000	0	0	1	1	1
land	00000	110011	1100	1	1	1	0	1
Lxor	00000	110011	0100	1	1	1	0	1
Lxnr	00000	110011	0101	1	1	1	0	1
Lor	00000	110011	1100	1	1	1	0	1
Lnor	00000	110011	1110	1	1	1	0	1
Lnot	00000	110011	1111	1	1	1	0	1
Lnand	00000	110011	1110	1	1	1	0	1
Lsl, Lsr	10001	000101	0000	0	0	1	1	1

Instruction Set

IPPro supports a 32-bit instruction set architecture (ISA) to process stream and non-stream data. Table B.2 lists the supported instruction set.

Table B.2: IPPro instruction set.

Addressing mode	Instruction	Description
	NOP	No Operation
Register File	ADD	RD = RB + RC
	SUB	RD = RC - RB
	MUL	RD = RA * RB
	MULADD	RD = RC + (RA * RB)
	MULSUB	RD = RC - (RA * RB)
	MULACC	RD = (RA * RB) + RD-1
	LAND	RD = RB & RC
	LXOR	RD = RB ˆRC
	LXNR	RD = ∼ (RB ˆRC)
	LOR	RD = RB — RC
	LNOR	RD = ∼(RB — RC)
	LNOT	RD = ∼RB
	LNAND	RD = ∼(RB & RC)
	MIN	RD = MIN(RB, RC)
	MAX	RD = MAX(RB, RC)
Data Handling	LDWM	RD = WMn(ADDRESS)
	STWM	WMn(ADDRESS) = RC
	LDWMI	RD = WM(R31)
	STWMI	WM(R31) = RC
	PUSH	FIFO(output) = RA
	GET	RD = FIFO (input)
	TEST	Checks no. of input tokens available in FIFO
	STR	RD = IMM (16-bit signed value)
BRANCH	JMP	16-bit code memory address
	BNEQ*	Branch if equal flag is clear
	BEQ*	Branch if equal flag is set
	BZ*	Branch if zero flag is set
	BNZ*	Branch if zero flag is clear
	BS*	Branch if Sign flag is set
	BNS*	Branch if Sign flag is clear

* The branch instructions have been added at no extra cost and included in the IPPro instruction set, as the IPPro flags (zero, sign and equal) have been generated using a pattern-detect and a sign-bit produced by the embedded DSP48E1 block.

AXI4 Control Registers

IPPro has AMB-AXI4-Lite interface that allows configuration of actor firing, source-ID and destination-ID encoder modules necessary to implement multi-rate dataflow actors. The datapath has nine registers listed in Table B.3 that stores the configurations.

Table B.3: The AXI4-Lite control register map.

AXI4-Lite Registers		**Bits**							
Addr.	**IPPro Core**	31 - 28	27 - 24	23 - 20	19 - 16	15 - 12	11 - 8	7 - 4	3 - 0
0x00	CONTROL	xxx							RST
0x04	FIRING MASK	xxx						FIRING_MASK	
0x08	Tk_CONSUMPTION	Q7	Q6	Q5	Q4	Q3	Q2	Q1	Q0
0x0C	Tk_PRODUCTION	xxx							Tk
0x10	IM_ADDRESS	WR_EN	xxx	IM_ADDR					
0x14	IM_DATA	IM_DATA							
0x18	SP_ADDRESS	WR_EN	xxx	SP_ADDR					
0x1C	SP_DATA_IN	xxx				SP_DATA_IN (16-bits)			
0x20	SP_DATA_OUT	xxx				SP_DATA_OUT			

Software-based control interface

C-APIs has been developed to ease programming and control of supported features. The IPPro datapath has nine AXI4-Lite control registers. Listing A.1 shows these functions:

Listing B.1: C-APIs to control and manage IPPro core.

```c
// Core Register read/write functions
int IPProWrite(IPPro* inst, IPProRegAddr addr, uint32_t command);
uint32_t IPProRead(IPPro* inst, IPProRegAddr addr);
int IPProSetTokenConsumption(IPPro* inst, IPProRegAddr addr, uint32_t *qCount);
int IPProSetTokenProduction(IPPro* inst, IPProRegAddr addr, uint32_t command);
int IPProSetFiringMask(IPPro* inst, IPProRegAddr addr, uint32_t command);

// Instruction memory functions
int IPProIMWrite(IPPro* inst, uint32_t addr, uint32_t program);
int IPProIMInit(IPPro* inst, uint32_t *code, uint32_t n);

// Scratchpad memory functions
int IPProSPWrite(IPPro* inst, uint32_t addr, uint32_t data);
uint32_t IPProSPRead(IPPro* inst, uint32_t addr);
int IPProSPInit(IPPro* inst, uint32_t *data, uint32_t n);
```

Bibliography

[1] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge Analytics in the Internet of Things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24 – 31, 2015.

[2] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management (IJIM)*, vol. 35, no. 2, pp. 137 – 144, 2015.

[3] (2016) Video Analytics Hardware, Software, and Services Revenue to Reach $3 Billion by 2022. [Online]. Available: https://www.embedded-vision.com/industry-analysis/market-analysis/video-analytics-hardware-software-and-services-revenue-reach-3-bil

[4] I. L. Markov, "Limits on Fundamental Limits to Computation," *Coputing Research Laboratory (CoRR)*, vol. abs/1408.3821, 2014. [Online]. Available: http://arxiv.org/abs/1408.3821

[5] "Programmable embedded platforms for remote and compute intensive image processing applications (EP/K009583/1)." [Online]. Available: http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/K009583/1

[6] R. Stewart, K. Duncan, G. Michaelson, P. Garcia, D. Bhowmik, and A. Wallace, “RIPL: A Parallel Image Processing Language for FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS*, vol. 11, no. 1, Mar. 2018.

[7] E. Bezati, “High-level synthesis of dataflow programs for heterogeneous platforms,” Ph.D. dissertation, EPFL, Lausanne, Switzerland, 2015. [Online]. Available: https://infoscience.epfl.ch/record/207992/files/EPFL_TH6653.pdf

[8] D. F. Bacon, R. Rabbah, and S. Shukla, “FPGA Programming for the Masses,” *ACM Queue Magazine*, vol. 11, no. 2, pp. 40 – 52, Feb. 2013.

[9] M. Gort and J. Anderson, “Design re-use for compile time reduction in FPGA high-level synthesis flows,” in *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, Shanghai, China, Dec. 2014, pp. 4 – 11.

[10] G. Stitt and J. Coole, “Intermediate Fabrics: Virtual Architectures for Near-Instant FPGA Compilation,” *IEEE Embedded Systems Letters*, vol. 3, no. 3, pp. 81 – 84, 2011.

[11] R. Baxter, S. Booth, M. Bull, G. Cawood, K. D’Mellow, X. Guo, M. Parsons, J. Perry, A. Simpson, and A. Trew, “High-Performance Reconfigurable Computing - the View from Edinburgh,” in *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, Washington, USA, 2007, pp. 373 – 279.

[12] H. Chenini, J. P. Dérutin, R. Aufrère, and R. Chapuis, "Parallel embedded processor architecture for FPGA-based image processing using parallel software skeletons," *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, p. 153, 2013.

[13] L. Natvig, A. Iordan, M. Eleyat, M. Jahre, and J. Amundsen, *Multi- and Many-Cores, Architectural Overview for Programmers*, 2017, ch. 1, pp. 1 – 27.

[14] A. Vajda, *Multi-core and Many-core Processor Architectures*, Boston, USA, 2011, ch. 2, pp. 9 – 43.

[15] K. Andryc, T. Thomas, and R. Tessier, "Soft GPGPUs for Embedded FPGAs: An Architectural Evaluation," *CoRR*, vol. abs/1606.06454, 2016. [Online]. Available: http://arxiv.org/abs/1606.06454

[16] N. Kapre, "Custom FPGA-based soft-processors for sparse graph acceleration," in *Proc. 26th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Toronto, Canada, Jul. 2015, pp. 9 – 16.

[17] T. Lieske, M. Reichenbach, B.Ringlein, and D. Fey, "Dataflow optimization for programmable embedded image preprocessing accelerators," in *Proc. IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Nov. 2016, pp. 1 – 8.

[18] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch,

P. Puschner, A. Rocha, C. Silva, J. Spars, and A. Tocchi, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture (JSA)*, vol. 61, no. 9, pp. 449 – 471, 2015.

[19] J. Heisswolf, A. Zaib, A. Weichslgartner, M. Karle, M. Singh, T. Wild, J. Teich, A. Herkersdorf, and J. Becker, "The invasive network on chip - a multi-objective many-core communication infrastructure," in *Proc. IEEE International Workshop on Architecture of Computing Systems (ARCS)*, Luebeck, Germany, Feb. 2014, pp. 1 – 8.

[20] D. She, Y. He, L. Waeijen, and H. Corporaal, "A Co-Design Framework with OpenCL Support for Low-Energy Wide SIMD Processor," *Journal of Signal Processing Systems (JSPS)*, vol. 80, no. 1, pp. 87 – 101, Jul. 2015.

[21] S. Fernando, F. Siyoum, Y. He, A. Kumar, and H. Corporaal, "MAMPSx: A design framework for rapid synthesis of predictable heterogeneous MP-SoCs," in *Proc. IEEE International Symposium on Rapid System Prototyping (RSP)*, Montreal, Canada, Oct. 2013, pp. 136 – 142.

[22] J. Sérot, F. Berry, and C. Bourrasset, "High-level dataflow programming for real-time image processing on smart cameras," *Journal of Real-Time Image Processing (JRTIP*, vol. 12, no. 4, pp. 635 – 647, 2016.

[23] C. Liu, H. C. Ng, and H. K. H. So, "QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay," in *Proc. IEEE International Conference on Field Programmable Technology (FPT)*, Queenstown, New Zealand, Dec. 2015, pp. 56 – 63.

[24] R. Stewart, D. Bhowmik, A. Wallace, and G. Michaelson, “Profile Guided Dataflow Transformation for FPGAs and CPUs,” *Journal of Signal Processing Systems (JSPS)*, vol. 87, no. 1, pp. 3 – 20, 2017.

[25] C. Sau, P. Meloni, L. Raffo, F. Palumbo, E. Bezati, S. Casale-Brunet, and M. Mattavelli, “Automated Design Flow for Multi-Functional Dataflow-Based Platforms,” *Journal of Signal Processing Systems (JSPS)*, vol. 85, no. 1, pp. 143 – 165, Oct. 2016.

[26] P. R. Schaumont, *Data Flow Modeling and Transformation*, Boston, USA, 2013, pp. 31 – 59.

[27] J. Su, F. Yang, X. Zeng, D. Zhou, and J. Chen, “Efficient Memory Partitioning for Parallel Data Access in FPGA via Data Reuse,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 10, pp. 1674 – 1687, 2017.

[28] M. Sadri, C. Weis, N. Wehn, and L. Benini, “Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ,” in *Proc. 10th FPGAworld Conference (FPGAworld ’13)*, Stockholm, Sweden, 2013, pp. 5:1–5:8.

[29] K. Vipin and S. A. Fahmy, “ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq,” *IEEE Embedded Systems Letters*, vol. 6, no. 3, pp. 41 – 44, Sep. 2014.

[30] M. I. Gordon, W. Thies, and S. Amarasinghe, “Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs,” *Proc. 12th International conference on Architectural support for programming languages*

and operating systems (ASPLOS XII), vol. 41, no. 11, pp. 151 – 162, Oct. 2006.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proc. in International Conference on Compiler Construction*, Berlin, Heidelberg, Mar. 2002, pp. 179 – 196.

[32] M. Schmid, N. Apelt, F. Hannig, and J. Teich, "An image processing library for C-based high-level synthesis," in *Proc. 24th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, Gernoble, France, Sep. 2014, pp. 1 – 4.

[33] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient Realization of Streaming Applications on FPGAs," in *Proc. 25th ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08)*, New York, NY, USA, Oct. 2008, pp. 41 – 50.

[34] O. Reiche, K. Häublein, M. Reichenbach, F. Hannig, J. Teich, and D. Fey, "Automatic Optimization of Hardware Accelerators for Image Processing," *CoRR*, vol. abs/1502.07448, 2015. [Online]. Available: http://arxiv.org/abs/1502.07448

[35] H. K.-H. So and C. Liu, *FPGA Overlays*, 2016, pp. 285 – 305.

[36] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *Proc. IEEE International Conference on Field-Programmable Technology (FPL)*, Kyoto, Japan, Dec. 2013, pp. 230 – 237.

[37] G. Stitt, "Are Field-Programmable Gate Arrays Ready for the Mainstream?" *IEEE Micro*, vol. 31, no. 6, pp. 58 – 63, 2011.

[38] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1591 – 1604, Oct. 2016.

[39] F. Hannig, *A Quick Tour of High-Level Synthesis Solutions for FPGAs*, 2016, pp. 49 – 59.

[40] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 144:1–144:11, Jul. 2014.

[41] S. C. Brunet, M. Mattavelli, and J. W. Janneck, "Buffer optimization based on critical path analysis of a dataflow program design," in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, Beijing, China, May 2013, pp. 1384 – 1387.

[42] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level Synthesis: Productivity, Performance, and Software Constraints," *Journal of Electrical and Computer Engineering (JECE)*, vol. 2012, Jan. 2012.

[43] E. Bezati, S. C. Brunet, M. Mattavelli, and J. W. Janneck, "High-level system synthesis and optimization of dataflow programs for MPSoCs," in *Proc.*

50th IEEE International Conference on Signals, Systems and Computers, California, USA, Nov. 2016, pp. 417 – 421.

[44] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?" in *Proc. 14th IEEE International Conference on Dependable, Autonomic and Secure Computing(DASC)*, Auckland, New Zealand, Aug. 2016, pp. 586–593.

[45] R. Rigamonti, B. Delporte, A. Convers, and A. Dassatti, "Transparent Live Code Offloading on FPGA," in *Proc. 3rd International Workshop on FPGAs for Software Programmers (FSP)*, Aug. 2016, pp. 1 – 10.

[46] F. de Dinechin and B. Pasca, *Reconfigurable Arithmetic for High-Performance Computing*, 2013, pp. 631 – 663.

[47] X. Chu and J. McAllister, "FPGA based soft-core SIMD processing: A MIMO-OFDM fixed-complexity sphere decoder case study," in *Proc. IEEE International Conference on Field-Programmable Technology (FPL)*, Beijing, China, Dec. 2010, pp. 479 – 484.

[48] C. E. LaForest and J. G. Steffan, "OCTAVO: an FPGA-centric processor family," in *Proc. ACM/SIGDA International Symposium on Field programmable gate arrays (FPGA)*, Monterey, USA, Feb. 2012, pp. 219 – 228.

[49] B. Ronak and S. A. Fahmy, "Evaluating the efficiency of DSP block synthesis inference from flow graphs," in *Proc. IEEE International Conference on Field Programmable Logic and Applications (FPL)*, Oslo, Norway, 2012, pp. 727 – 730.

[50] Xilinx, "All Programmable 7 Series: Product Selection Guide," Xilinx Inc., Tech. Rep., 2018.

[51] ——, "UltraScale+ FPGAs: Product Tables and Product Selection Guide," Xilinx Inc., Tech. Rep., 2018.

[52] ——, "7 Series DSP48E1 Slice," Xilinx Inc., User Guide: UG479(v1.10), 2018.

[53] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773 – 801, May 1995.

[54] S. Casale-Brunet, "Analysis and optimization of dynamic dataflow programs," Ph.D. dissertation, EPFL, Lausanne, Switzerland, 2015. [Online]. Available: https://infoscience.epfl.ch/record/207992/files/EPFL_TH6653.pdf

[55] A. Azarian and J. M. Cardoso, "Pipelining data-dependent tasks in FPGA-based multicore architectures," *Microprocessors and Microsystems*, vol. 42, pp. 165 – 179, 2016.

[56] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical Modeling of Pipeline Parallelism," in *Proc. 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Raleigh, USA, Sep. 2009, pp. 281 – 290.

[57] A. Turjan, B. Kienhuis, and E. Deprettere, "Solving Out-of-Order Communication in Kahn Process Networks," *Journal of VLSI signal processing*

systems for signal, image and video technology, vol. 40, no. 1, pp. 7 – 18, 2005.

[58] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A Catalog of Stream Processing Optimizations," *ACM Jounral on Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 46:1 – 46:34, Mar. 2014.

[59] F. Otto, V. Pankratius, and W. F. Tichy, "XJava: Exploiting Parallelism with Object-Oriented Stream Programming," in *Proc. 15th European Conference on Parallel Processing (Euro-Par)*, Delft, Netherlands, Aug. 2009, pp. 875 – 886.

[60] C. Kelly, "Design exploration of image processing algorithms using FPGA based soft-core processors," Ph.D. dissertation, Queens University Belfast, May 2016.

[61] X. Chu, J. McAllister, and R. Woods, "A Pipeline Interleaved Heterogeneous SIMD Soft Processor Array Architecture for MIMO-OFDM Detection," in *Proc. 7th IEEE International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC)*, Belfast, UK, 2011, pp. 133 – 144.

[62] X. Wang and S. Ziavras, "Exploiting mixed-mode parallelism for matrix operations on the HERA architecture through reconfiguration," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, pp. 249 – 260(11), Jul. 2006.

[63] P. P. Jonker, "Why linear arrays are better image processors," in *Proc. 12th IEEE International Conference on Pattern Recognition.*

[64] D. G. Bailey and C. T. Johnston, "Algorithm Transformation for FPGA Implementation," in *Proc. 5th IEEE International Symposium on Electronic Design, Test and Applications (DELTA)*, Ho Chin Minh City, Vietnam, Jan. 2010, pp. 77 – 81.

[65] C. Nugteren, H. Corporaal, and B. Mesman, "Skeleton-based automatic parallelization of image processing algorithms for GPUs," in *Proc. IEEE International Conference on Systems, Architectures, Modeling, and Simulation (SAMOS)*, Samos, Greece, Jul. 2011, pp. 25 – 32.

[66] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid, "High Level Programming for FPGA Based Image and Video Processing Using Hardware Skeletons," in *Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Mar. 2001, pp. 219 – 226.

[67] P. A. Nelson, "Parallel Programming Paradigms," Ph.D. dissertation, 1987.

[68] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, *Fastflow: High-level and Efficient Streaming on Multi-core*, 2017, ch. 13.

[69] M. Vanneschi, "The programming model of assist, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709 – 1732, 2002.

[70] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107 – 113, 2008.

[71] H.-W. Loidl, "Parallel Program Design & Algorithmic Skeletons," Heriot-Watt University, Edinburgh, Tech. Rep., 2017. [Online]. Available: https://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/l08_handout.pdf

[72] M. Poldner and H. Kuchen, "On implementing the Farm skeleton," *Parallel Processing Letters*, vol. 18, no. 1, pp. 117 – 131, 2008.

[73] F. Plavec, B. Fort, Z. Vranesic, and S. Brown, "Experiences with Soft-Core Processor Design," in *Proc. 19th IEEE International Proceedings on Parallel and Distributed Processing Symposium*, Denver, USA, Apr. 2005, pp. 167b – 167b.

[74] T. Kranenburg and R. Van Leuken, "MB-LITE: A robust, light-weight soft-core implementation of the MicroBlaze architecture," in *Proc. IEEE International Conference & Exhibition on Design, Automation Test in Europe (DATE)*, Dresden, Germany, Mar. 2010, pp. 997 – 1000.

[75] (2013) Open Cores Main Page. [Online]. Available: http://opencores.org/or1k/Main_Page

[76] (2013) Aeroflex Gaisler AB. Leon3. [Online]. Available: http://gaisler.com/index.php/products/processors/leon3

[77] S. Hive, "Silicon Hive Technology Primer: Reconfigurable accelerators that bring computational efficiency (MOPS/W)and programmability together, to displace ASIC and DSP co-processors in Systems-on-Chips," Silicon Hive, Tech. Rep., 2003. [Online]. Available: https://www.scribd.com/document/26948963/Silicon-Hive

[78] E. V. Dalen, S. G. Pestana, and A. V. Wel, "An Integrated, Low-Power Processor for Image Signal Processing," in *in Proc. of 8th IEEE International Symposium on Multimedia (ISM'06)*, San Diego, US, Dec. 2006, pp. 501–508.

[79] A. Duller, G. Panesar, and D. Towner, "Parallel Processing-the picoChip way," *Communicating Processing Architectures*, pp. 125–138, 2003.

[80] (2007) BDTI Releases Benchmark Results for Massively Parallel picoChip PC102. [Online]. Available: https://www.bdti.com/InsideDSP/2007/09/26/Pico

[81] R. M. Russell, "The CRAY-1 computer system," *Communications of the ACM - Special issue on Computer Architectures*, vol. 21, no. 1, pp. 63 – 72, Jan. 1978.

[82] C. H. Chou, A. Severance, A. D. Brant, Z. Liu, S. Sant, and G. G. Lemieux, "VEGAS: soft vector processor with scratchpad memory," in *Proc. ACM/SIGDA International Symposium on Field programmable gate arrays (FPGA)*, Monterey, USA, Feb. 2011, pp. 15 – 24.

[83] A. Severance and G. Lemieux, "VENICE: A compact vector processor for FPGA applications," in *Proc. IEEE International Conference on Field-Programmable Technology (FPT)*, Seoul, Korea, Dec. 2012, pp. 261 – 268.

[84] A. Severance and G. G. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *Proc. IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Montreal, Canada, Oct. 2013, pp. 1 – 10.

[85] M. Milford and J. McAllister, “An ultra-fine processor for FPGA DSP chip multiprocessors,” in *Proc. 43rd Asilomar Conference on Signals, Systems and Computers*, Belfast, UK, Nov. 2009, pp. 226 – 230.

[86] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, “The iDEA DSP Block-Based Soft Processor for FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS*, vol. 7, no. 3, pp. 19:1–19:23, 2014.

[87] J. Eker and J. Janneck, “CAL language report: Specification of the CAL actor language,” University of California, Berkelry, Tech. Rep., 2003. [Online]. Available: https://ptolemy.berkeley.edu/papers/03/Cal/

[88] B. Bardak, R. Stewart, and D. Bhowmik, “Data-flow modeling to capture the processing and data organisation of image processing algorithms in language representation,” Project Rathlin, Tech. Rep., Feb. 2014. [Online]. Available: http://rathlin.hw.ac.uk/images/documents/report/D1_report.pdf

[89] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, “Orcc: Multimedia Development Made Easy,” in *Proceedings of the 21st ACM International Conference on Multimedia*, ser. MM ’13. ACM, 2013, pp. 863–866.

[90] A. Azarian and J. M. P. Cardoso, “Coarse/Fine-grained Approaches for Pipelining Computing Stages in FPGA-Based Multicore Architectures,” in *Proc. European Conference on Parallel Processing Workshops (Euro-Par)*, Porto, Portugal, 2014, pp. 266 – 278.

[91] W. Sadiq and M. E. Orlowska, "Analyzing process models using graph reduction techniques," *Information Systems*, vol. 25, no. 2, pp. 117 – 134, 2000.

[92] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. of the IEEE*, vol. 77, no. 4, pp. 541 – 580, Apr. 1989.

[93] D. Buono, M. Danelutto, S. Lametti, and M. Torquati, "Parallel Patterns for General Purpose Many-Core," in *Proc. 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Belfast, UK, Feb. 2013, pp. 131 – 139.

[94] D. Nguyen, D. Halupka, P. Aarabi, and A. Sheikholeslami, "Real-time face detection and lip feature extraction using field-programmable gate arrays," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 36, no. 4, pp. 902 – 912, Aug. 2006.

[95] R. O. Chavez-Garcia and O. Aycard, "Multiple Sensor Fusion and Classification for Moving Object Detection and Tracking," *IEEE Transactions on Intelligent Transportation Systems (ITS)*, vol. 17, no. 2, pp. 525 – 534, Sep. 2016.

[96] F. Bounini, D. Gingras, V. Lapointe, and H. Pollart, "Autonomous Vehicle and Real Time Road Lanes Detection and Tracking," in *Proc. IEEE Vehicle Power and Propulsion Conference (VPPC)*, Montreal, Canada, Oct. 2015, pp. 1 – 6.

[97] Xilinx, "7 Series FPGAs Memory Resource," Xilinx Inc., User Guide: UG473 (v1.12), 2016.

[98] T. Feist, "Zynq-7000 All Programmable SoC (Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics," Xilinx Inc., DataSheet: DS187 (v1.20), 2017.

[99] S. Gupta, "Comparison of different data flow graph models," University of Stuttgart, Tech. Rep., 2010. [Online]. Available: http://www.iti.uni-stuttgart.de/~radetzki/Seminar06/11_report.pdf

[100] G. Martin and S. Leibson, "Beyond the Valley of the Lost Processors: Problems, Fallacies, and Pitfalls in Processor Design," in *Processor Design.* Springer, 2007, pp. 27 – 67.

[101] F. M. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty, "IP-Pro: FPGA based image processing processor," in *Proc. IEEE International Workshop on Signal Processing Systems (SiPS)*, Oct. 2014, pp. 1 – 6.

[102] C. Kelly, F. M. Siddiqui, B. Bardak, and R. Woods, "Histogram of oriented gradients front end processing: an FPGA based processor approach," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, 2014, pp. 1 – 6.

[103] C. Kelly, F. M. Siddiqui, B. Bardak, Y. Wu, R. Woods, and K. Rafferty, "FPGA Soft-Core Processors, Compiler and Hardware Optimizations Validated Using HOG," in *Proc. International Symposium on Applied Reconfigurable Computing (ARC)*, Rio de Janeiro, Brazil, Mar. 2016, pp. 78 – 90.

[104] S. Jin, J. Cho, X. D. Pham, K. M. Lee, S. K. Park, M. Kim, and J. W. Jeon, "FPGA Design and Implementation of a Real-Time Stereo Vision System,"

IEEE Transactions on Circuits and Systems for Video Technology, vol. 20, no. 1, pp. 15 – 26, Jan. 2010.

[105] W. He and K. Yuan, “An improved Canny edge detector and its realization on FPGA,” in *Proc. 7th World Congress on Intelligent Control and Automation*, Chongqing, China, Jun. 2008, pp. 6561 – 6564.

[106] B. A. Draper, J. R. Beveridge, A. P. W. Bohm, C. Ross, and M. Chawathe, “Accelerated image processing on FPGAs,” *IEEE Transactions on Image Processing*, vol. 12, no. 12, pp. 1543 – 1551, Dec. 2003.

[107] R. Harinarayan, R. Pannerselvam, M. M. Ali, and D. K. Tripathi, “Feature extraction of Digital Aerial Images by FPGA based implementation of edge detection algorithms,” in *Proc. International Conference on Emerging Trends in Electrical and Computer Technology (ICETECT)*, Nagercoil, India, Mar. 2011, pp. 631 – 635.

[108] D. G. Bailey, *Local Filters*, 2011, ch. 8, pp. 233 – 273.

[109] P. Schleuniger, S. A. McKee, and S. Karlsson, “Design Principles for Synthesizable Processor Cores,” in *Proc. 25th International Conference on Architecture of Computing Systems (ARCS)*, Munich, Germany, Mar. 2012, pp. 111 – 122.

[110] M. Reichenbach, T. Lieske, S. Vaas, K. Haublein, and D. Fey, “FAUPU - A design framework for the development of programmable image processing architectures,” in *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Dec 2015, pp. 1–8.

[111] Xilinx, “FIFO Generator v13.0, LogiCORE Product Guide,” Xilinx Inc., Product Guide: PG057, 2015.

[112] D. Capalija and T. S. Abdelrahman, “A high-performance overlay architecture for pipelined execution of data flow graphs,” in *Proc. 23rd IEEE International Conference on Field programmable Logic and Applications (FPL)*, Porto, Portugal, Sep. 2013, pp. 1 – 8.

[113] M. Mustafa Rafique, A. R. Butt, and D. Nikolopoulos, *Programming and Managing Resources on Accelerator-Enabled Clusters*, 2015, ch. 20.

[114] H. Wei, J. Yu, H. Yu, M. Qin, and G. R. Gao, “Software Pipelining for Stream Programs on Resource Constrained Multicore Architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2338 – 2350, Dec. 2012.

[115] L. Li, T. Fanni, T. Viitanen, R. Xie, F. Palumbo, L. Raffo, H. Huttunen, J. Takala, and S. S. Bhattacharyya, “Low power design methodology for signal processing systems using lightweight dataflow techniques,” in *Proc. IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Rennes, France, Oct. 2016, pp. 82 – 89.

[116] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer, *The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems*, 2013, pp. 218 – 236.

[117] G. J. Garca, C. A. Jara, J. Pomares, A. Alabdo, L. M. Poggi, and F. Torres, “A Survey on FPGA-Based Sensor Systems: Towards Intelligent and Re-

configurable Low-Power Sensors for Computer Vision, Control and Signal Processing," *Sensors*, vol. 14, no. 4, pp. 6247 – 6278, 2014.

[118] K. M. A. Ali, R. B. Atitallah, S. Hanafi, and J. L. Dekeyser, "A generic pixel distribution architecture for parallel video processing," in *Proc. IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Dec. 2014, pp. 1 – 8.

[119] Chen, Chang Wen and Luo, Jiebo and Parker, Kevin J, "Image segmentation via adaptive K-mean clustering and knowledge-based morphological operations with biomedical applications," *IEEE Trans. Image Process.*, vol. 7, no. 12, pp. 1673 – 1683, 1998.

[120] F. Winterstein, S. Bayliss, and G. A. Constantinides, "FPGA-based K-means clustering using tree-based data structures," in *Proc. 23rd IEEE International Conference on Field programmable Logic and Applications (FPL)*, Porto, Portugal, Sep. 2013, pp. 1 – 6.

[121] Xillybus, "Getting started with Xillybus on a Linux host," Guide, Version 2.2, 2017. [Online]. Available: http://xillybus.com

[122] M. Danek, J. Kadlec, R. Bartosinski, and L. Kohout, "Increasing the level of abstraction in FPGA-based designs."

[123] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 62 – 80, 2018.

[124] L. Verdoscia and R. Giorgi, "A data-flow soft-core processor for accelerating scientific calculation on FPGAs," *Mathematical Problems in Engineering*, 2016.

[125] R. Kumar, V. Zyuban, and D. M. Tullsen, "Interconnections in Multi-Core Architectures: Understanding Mechanisms, Overheads and Scaling," in *Proc. 32nd International Symposium on Computer Architecture (ISCA)*, Madison, USA, Jun. 2005, pp. 408 – 419.

[126] M. Aldinucci and M. Danelutto, "Skeleton-based parallel programming: Functional and parallel semantics in a single shot," *Computer Languages, Systems & Structures*, vol. 33, pp. 179 – 192, 2007.

[127] S. Neuendorffer, T. Li, and D. Wang, "Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libra ries," Xilinx Inc., Application Note: XAPP1167 (v3.0), 2015.

[128] S. Wong, S. Vassiliadis, and S. Cotofana, "A sum of absolute differences implementation in FPGA hardware," in *Proc. 28th IEEE Euromicro Conference*, Dortmund, Germany, Sep. 2002, pp. 183–188.