Architecture and Programming Model Support For Reconfigurable Accelerators in Multi-Core Embedded Systems

Satyajit Das

To cite this version:

Satyajit Das. Architecture and Programming Model Support For Reconfigurable Accelerators in Multi-Core Embedded Systems. Hardware Architecture [cs.AR]. Université Bretagne Sud, 2018. English. NNT : . tel-01826653

HAL Id: tel-01826653
https://theses.hal.science/tel-01826653

Submitted on 29 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE DE DOCTORAT DE

L'UNIVERSITÉ BRETAGNE SUD
COMUE UNIVERSITE BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies
de l'Information et de la Communication
Spécialité : Électronique

Par
« Satyajit DAS »

« Architecture and Programming Model Support For Reconfigurable Accelerators in Multi-Core Embedded Systems »

Thèse présentée et soutenue à Lorient, le 4 juin 2018
Unité de recherche : Lab-STICC
Thèse N° : 491

Rapporteurs avant soutenance :

Michael HÜBNER	Professeur, Ruhr-Universität Bochum
Jari NURMI	Professeur, Tampere University of Technology

Composition du Jury :

Président	
François PÊCHEUX	Professeur, Sorbonne Université
Angeliki KRITIKAKOU	Maître de conférences, Université Rennes 1
Davide ROSSI	Assistant professor, Université de Bologna
Kevin MARTIN	Maître de conférences, Université Bretagne Sud
Directeur de thèse	
Philippe COUSSY	Professeur, Université Bretagne Sud
Co-directeur de thèse	
Luca BENINI	Professeur, Université de Bologna

In memory of my father
To my mother
With love and eternal appreciation...

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 43,500 words including bibliography, footnotes, tables and equations and has fewer than 50 figures.

Satyajit Das
June 2018

Acknowledgements

Writing the acknowledgements was a hard exercise, as it is tough to escape from the repetitive and ready to use expressions, commonly used in such occasions. I hope, I have found the words to appropriately thank all the people who influenced me to achieve the goals of this thesis.

I owe my deepest gratitude to my thesis advisor Professor Philippe Coussy, who suggested this topic to me and continued backing me up throughout. Without his continuous optimism concerning this work, enthusiasm, encouragement and support this journey would hardly have been completed. I also express my warmest gratitude to my other advisor Professor Luca Benini for his continuous support, feedback, and the criticisms for my work which helped for the betterment of the thesis.

I am specially grateful to my supervisors Dr. Kevin Martin and Dr. Davide Rossi for the encouragement, mentorship and guidance throughout my Ph.D. studies. Everything I learned during my PhD studies was because of you guys. It's been a great pleasure and honour working with you as a student. I wish to continue working with you in future as well.

I would like to thank all of my friends, current and former colleagues in Prof. Benini's group in Bologna and in LabSTICC. Particularly, I would like to thank (in no particular order) Francesco Paci, Alessandro Capotondi, Giuseppe Tagliavini, Daniele Cesarini, Igor Loi, Francesco Conti, Andrea Marongiu, Andrea Bartolini, Francesco Beneventi, Marco Balboni, Thomas Bridi, Simone Benatti and Manuele Rusci from the Energy-Efficient Embedded Systems laboratory in Bologna, Robin Danilo, Huges Wouafo, Mohamed Mourad Hafidhi, Maria Mendez and Erwan Moreac from LabSTICC and Abhishek Sachan from LIMATB lab in Lorient. I also would like to thank Florence Palin from LabSTICC and Angela Cavazzini from University of Bologna for their invaluable help in the most difficult of research tasks - the bureaucracy!

I want to express my gratitude to the reviewers of the manuscript Professor Jari Nurmi and Professor Michael Hubner, whose valuable suggestions helped to enhance the quality of the thesis.

Finally, I would like to thank my family to whom I owe a great deal. To my late father Shyam Baran Das, thank you for teaching me to believe in myself. To my brother Subhajit

and sister Susmita, thank you for all your love and support. To my girlfriend Devaroopa, thank you for continued and unfailing love and support. Finally, to my mother Rekha Das, who has been a constant source of support and encouragement and has made an untold number of sacrifices for the entire family. She's been a great inspiration to me.
I thank you all.

ABSTRACT

Emerging trends in embedded systems and applications need high throughput and low power consumption. Due to the increasing demand for low power computing, and diminishing returns from technology scaling, industry and academia are turning with renewed interest toward energy efficient hardware accelerators. The main drawback of hardware accelerators is that they are not programmable. Therefore, their utilization can be low as they perform one specific function and increasing the number of the accelerators in a system on chip (SoC) causes scalability issues. Programmable accelerators provide flexibility and solve the scalability issues.

Coarse-Grained Reconfigurable Array (CGRA) architecture consisting several processing elements with word level granularity is a promising choice for programmable accelerator. Inspired by the promising characteristics of programmable accelerators, potentials of CGRAs in near threshold computing platforms are studied and an end-to-end CGRA research framework is developed in this thesis.

The major contributions of this framework are: CGRA design, implementation, integration in a computing system, and compilation for CGRA. First, the design and implementation of a CGRA named Integrated Programmable Array (IPA) is presented. Next, the problem of mapping applications with control and data flow onto CGRA is formulated. From this formulation, several efficient algorithms are developed using internal resources of a CGRA, with a vision for low power acceleration. The algorithms are integrated into an automated compilation flow. Finally, the IPA accelerator is augmented in PULP - a Parallel Ultra-Low-Power Processing-Platform to explore heterogeneous computing.

Table of contents

List of figures **xv**

List of tables **xvii**

Introduction **5**

1 Background and Related Work **11**
- 1.1 Design Space 13
  - 1.1.1 Computational Resources 13
  - 1.1.2 Interconnection Network 13
  - 1.1.3 Reconfigurability 14
  - 1.1.4 Register Files 16
  - 1.1.5 Memory Management 16
- 1.2 Compiler Support 19
  - 1.2.1 Data Level Parallelism (DLP) 19
  - 1.2.2 Instruction Level Parallelism (ILP) 19
  - 1.2.3 Thread Level Parallelism 20
- 1.3 Mapping 22
- 1.4 Representative CGRAs 23
  - 1.4.1 MorphoSys 23
  - 1.4.2 ADRES 23
  - 1.4.3 RAW 25
  - 1.4.4 TCPA 25
  - 1.4.5 PACT XPP 25
- 1.5 Conclusion 26

2 Design of The Reconfigurable Accelerator **27**
- 2.1 Design Choices 31
- 2.2 Integrated Programmable Array Architecture 35

2.2.1 IPA components 35
2.2.2 Computation Model 37
2.3 Conclusion 42

3 Compilation flow for the Integrated Programmable Array Architecture 45
3.1 Background 45
3.1.1 Architecture model 45
3.1.2 Application model 46
3.1.3 Homomorphism 49
3.1.4 Supporting Control Flow 49
3.2 Compilation flow 53
3.2.1 DFG mapping 55
3.2.2 CDFG mapping 62
3.2.3 Assembler 67
3.3 Conclusion 68

4 IPA performance evaluation 69
4.1 Implementation of the IPA 69
4.1.1 Area Results 70
4.1.2 Memory Access Optimization 70
4.1.3 Comparison with low-power CGRA architectures 75
4.2 Compilation 78
4.2.1 Performance evaluation of the compilation flow 78
4.2.2 Comparison of the register allocation approach with state of the art predication techniques 78
4.2.3 Compiling smart visual trigger application 79
4.3 Conclusion 81

5 The Heterogeneous Parallel Ultra-Low-Power Processing-Platform (PULP) Cluster 83
5.1 PULP heterogeneous architecture 84
5.1.1 PULP SoC overview 84
5.1.2 Heterogeneous Cluster 85
5.2 Software infrastructure 86
5.3 Implementation and Benchmarking 88
5.3.1 Implementation Results 89
5.3.2 Performance and Energy Consumption Results 89

5.4 Conclusion 93

Summary and Future work **95**

References **99**

List of figures

1 Block diagram of CGRA 7
2 Wide spectrum of Accelerators 7
3 Energy efficiency vs Flexibility and Performance of CGRA vs CPU, DSP, MC, GPU, FPGA and ASIC 8
4 Thesis overview 10

1.1 Different interconnect topologies 14
1.2 Reconfigurability 15
1.3 Morphosys Architecture 24
1.4 ADRES Architecture 24
1.5 RAW Architecture 25
1.6 TCPA Architecture 26
1.7 PAE architecture of PACT XPP 26

2.1 Latency comparison for 3×3 CGRA 32
2.2 Latency comparison for 4×4 CGRA 32
2.3 Latency comparison for 5×5 CGRA 33
2.4 Execution strategies in multi-core cluster and IPA cluster 34
2.5 System level description of the Integrated Programmable Array Architecture 36
2.6 Components of PE 38
2.7 The configuration network for load-context 39
2.8 Segments of the GCM 40
2.9 Format of the data and address bus in the configuration network 40
2.10 Sample execution in CPU and IPA 41

3.1 Architecture and application model used for mapping 47
3.2 CDFG of a sample program 48
3.3 Classification of control flow present in an application 49
3.4 Mapping using full and partial predication 51

3.5 Compilation flow 54
3.6 DFG scheduling and transformation 56
3.7 Graph transformation 57
3.8 Performance comparison for different threshold functions 58
3.9 Mapping latency comparison for 3x3 CGRA 60
3.10 Compilation time comparison for 3x3 CGRA 61
3.11 Architectural coverage between methods 62
3.12 Assignment routing graph transformation 65
3.13 Different location constraints in CDFG mapping 65

4.1 Synthesized area of IPA for different number of TCDM banks 71
4.2 Performance of IPA executing matrix multiplication of different size 73
4.3 Latency performance in different configurations ([#LSUs][#TCDM Banks]) 73
4.4 Average power breakdown in different configurations ([#LSUs][#TCDM Banks]) 74
4.5 Average energy efficiency for different configurations ([#LSUs][#TCDM Banks]) 75
4.6 Energy efficiency/area trade-off between several configurations ([#LSUs][#TCDM Banks]) 76

5.1 PULP SoC. Source [91] 85
5.2 Block diagram of the heterogeneous cluster. 86
5.3 Block diagram of the IPA subsystem. 87
5.4 Synchronous interface for reliable data transfer between the two clock domains. 88
5.5 Power consumption breakdown while executing compute intensive kernel in PULP heterogeneous cluster 90
5.6 Power consumption breakdown while executing control intensive kernel in PULP heterogeneous cluster 90

List of tables

1.1 Qualitative comparison between different architectural approaches 18
1.2 CGRA design space and compiler support: CR - Computational Resources; IN - Interconnect Network; RC - Reconfigurability; MM - Memory management; CS - Compilation Support; MP - Mapping; PR - Parallelism 21

2.1 Characteristics of the benchmarks used for design space exploration 31
2.2 Energy gain in IPA cluster compared to the multi-core execution 35
2.3 Structure of a segment 40
2.4 Instruction format 41
2.5 Summary of the opcodes (R = Result, C = Condition bit) 43

3.1 Comparison between different approaches to manage control flow in CGRA 53
3.2 Comparison of RLC and TLC numbers between different CDFG traversal . 67

4.1 Specifications of memories used in TCDM and each PE of the IPA 70
4.2 Code size and the maximum depth of loop nests for the different kernels in the IPA 71
4.3 Overall instructions executed and energy consumption in IPA vs CPU . . . 72
4.4 Comparison with the state of the art low power targets 77
4.5 Performance (cycles) comparison between the register allocation approach and the state of the art approaches 79
4.6 Energy consumption (μJ) comparison between the register allocation approach and the state of the art approaches 80
4.7 Performance comparison 80
4.8 Performance comparison of smart visual surveillance application [cycles/pixel] 81

5.1 List of APIs for controlling IPA 88
5.2 Cluster Parameters and memories used 89
5.3 Synthesized area information for the PULP heterogeneous cluster 91

5.4 Performance evaluation in execution time (ns) for different configuration in the heterogeneous platform 92
5.5 Performance comparison between iso-frequency and 2× frequency execution in IPA 92
5.6 Energy consumption evaluation in μ J for different configuration in the heterogeneous platform 93
5.7 Comparison between total number of memory operations executed 93

List of Abbreviations

HWAC	Hardware Accelerator
ASIC	Application-Specific Integrated Circuit
GPU	Graphics Processing Unit
GPGPU	General-Purpose Graphics Processing Unit
HPC	High Performance Computing
CLB	Configurable Logic Block
CGRA	Coarse-Grained Reconfgurable Array
PE	Processing Element
FU	Functional Unit
ALU	Arithmetic Logic Unit
LSU	Load-Store Unit
RF	Register File
MC	Multi-Core
DSP	Digital Signal Processor
ASIP	Application Specific Instruction Processor
CMP	Chip Multi Processing
SIMD	Single Instruction Multiple Data
VLIW	Very Long Instruction Word
IPA	Integrated Programmable Array
SoC	System on Chip
PULP	Parallel-processing Ultra Low-power Platform

DLP	Data Level Parallelism
ILP	Instruction Level Parallelism
TLP	Thread Level Parallelism
CR	Computational Resources
IN	Interconnect Network
MM	Memory Management
RC	Reconfigurability
CS	Compilation Support
DFG	Data Flow Graph
CDFG	Control Data Flow Graph
ASAP	As Soon As Possible schedule
TCDM	Tightly Coupled Data Memory
GCM	Global Context Memory
IPAC	IPA controller
PEA	PE Array
PMU	Power Management Unit
IRF	Instruction Register File
RRF	Regular Register File
CRF	Constant Register File
OPR	Output Register
CR	Control Register
JR	Jump Register
NOP	No Operation
BB	Basic Block
TLC	Target Location Constraint
RLC	Reserved Location Constraint
RED	REDundant elimination

SnoB	Stochastic pruning with no Bounds
SLUB	Stochastic pruning with Lower and Upper Bounds
SLoB	Stochastic pruning with Lower only Bound
MOPS	Million Operations Per Second
FIFO	First In First Out
SLS	Systematic Load Store
DMA	Direct Memory Access
JIT	Just In Time

Introduction

With the increasing transistor density, the power dissipation improves very little with each generation of Moore's law. As a result, for fixed chip-level power budgets, the fraction of transistors that can be active at full frequency is decreasing exponentially. The empirical studies in [34] show that the strategy to enhance performance by increasing the number of cores will probably fail since voltage scaling has slowed or almost stopped, and the power consumption of individual cores are not reducing enough to allow the increase in the number of active computing units. Hence, as technology scales, an increasing fraction of the silicon will have to be dark, i.e., be powered off or under-clocked. This study estimated that at 8nm, more than 50% of the chip will have to be dark. The most popular approach to improve energy efficiency is a heterogeneous multi-core which is populated with a collection of specialized or custom hardware accelerators (HWAC), each optimized for a specific task such as graphics processing, signal processing, cryptographic computations etc.

Depending on the specific application domain, the trend is to have few general-purpose processors accelerated by highly optimized application-specific hardware accelerators (ASIC-HWACCs) or General-Purpose Graphics Processing Units (GPGPUs). Although ASIC-HWACCs provide the best performance/power/area figures, the lack of flexibility drastically limits their applicability to few domains (i.e. those where the same device can be used to cover large volumes, or where the cost of silicon is not an issue). Graphics Processing Units (GPUs) are very popular in high performance computing (HPC). Although they are programmable, their energy efficiency and performance advantages are limited to parallel loops [94]. Moreover, GPUs require significant effort to program them using specialized languages (e.g. CUDA). GPUs have rapidly evolved not to be limited only to perform graphics processing but also general purpose computing and referred to as GPGPU. As GPGPUs consist of thousands of cores, they are excellent computing platforms for the workloads that can be partitioned into a large number of threads with minimal interaction between the threads. The effectiveness of GPGPUs decreases significantly as the number of workload partitions decreases or the interaction between them increases [106]. In addition, the memory access contentions across threads should be minimized to diminish the performance penalty [106]. In such acceleration model, programmers are responsible to find an effective way to partition the workload.

In between of the two extremes ASIC-HWACC and GPU, Field Programmable Gate Arrays (FPGAs) provide high degree of reconfigurability. FPGAs offer significant advantages in terms of sharing hardware between distinct isolated tasks, under tight time constraints. Historically, reconfigurable resources available in FPGA fabrics have been used to build high performance accelerators in specific domains. The HPC domain has driven the development of reconfigurable resources, from relatively simple modules to highly parametrizable and configurable subsystems. While FPGAs started out as a matrix of programmable processing elements, called configurable logic blocks (CLBs) connected by programmable interconnect to configurable I/O, they have evolved to also include a variety of processing macros, such as reconfigurable embedded memories and DSP blocks to improve the efficiency of FPGA based accelerators. The flexibility, however, comes at the cost of programming difficulty and high static power consumption [42]. The high energy overhead due to fine-grained reconfigurability and long interconnects limit their use in ultra-low power environments like internet-of-things (IoT), wireless sensor networks etc. In addition, limitations and overhead of reconfiguring FPGAs at run-time impose a significant restriction on using FPGAs extensively in wider set of energy-constrained applications. Kuon et al in [58] shows that FPGAs require an average of 40× area overhead, 12× power overhead and 3× execution time overhead than a comparable ASIC.

A promising alternative to ASIC-HWACC, GPU and FPGA is the Coarse-Grained Reconfigurable Array (CGRA) [104]. As opposed to FPGAs, CGRAs are programmable at instruction level granularity. Due to this feature, compared to FPGAs, a significantly less silicon area is required to implement CGRAs. Besides, static power consumption is much lower in CGRAs compared to FPGAs. CGRAs consist of multi-bit functional units, which are connected through rich interconnect network and have been shown to achieve high energy efficiency [8] while demonstrating the advantages of a programmable accelerator.

This dissertation capitalizes on the promising features of CGRAs as ultra-low-power accelerator in three segments. The first part studies the hardware and compiler for state of the art CGRAs. The second part is devoted to implementing a novel CGRA architecture referred to as Integrated Programmable Array (IPA). The third part is dedicated to compilation problems associated with the acceleration of applications in CGRAs and a novel compilation flow. The final part explores heterogeneous computing by augmenting the IPA in a state of the art multi-core platform.

Motivation for CGRAs

CGRAs (Figure1) are built around an array of processing elements (PEs) with word level granularity. A PE is a single-pipeline stage functional unit (FU) (e.g., ALU, multiplier) with a small local register file and simple control unit, which fetches instruction from a small instruction memory. Additionally, some PEs can perform load-store operations on a shared data memory, which are usually referred to as load-store units (LSUs). Rather than a compiler mapping a C program onto a single core, a CGRA tool flow maps a high level program over multiple processing elements.

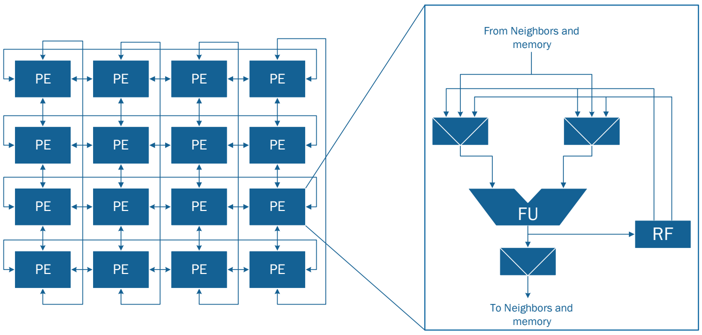

PE
From Neighbors and memory
FU
RF
To Neighbors and memory

Fig. 1 Block diagram of CGRA

In Figure 2, we present wide range of accelerator solutions. Figure 3 compares the CGRA architecture against the instruction processor solutions such as CPU, DSP, multi-core (MC), GPU and FPGA. Energy efficiency is shown against flexibility and performance.

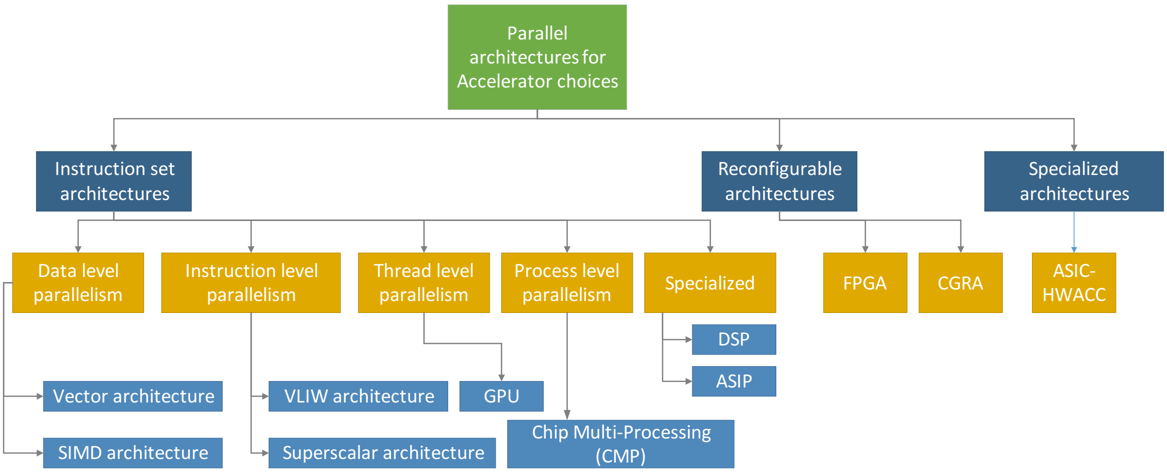

Parallel architectures for Accelerator choices
Instruction set architectures
Reconfigurable architectures
Specialized architectures
Data level parallelism
Instruction level parallelism
Thread level parallelism
Process level parallelism
Specialized
FPGA
CGRA
ASIC-HWACC
DSP
ASIP
Vector architecture
VLIW architecture
GPU
Chip Multi-Processing (CMP)
SIMD architecture
Superscalar architecture

Fig. 2 Wide spectrum of Accelerators

The charts are summarized as follows:

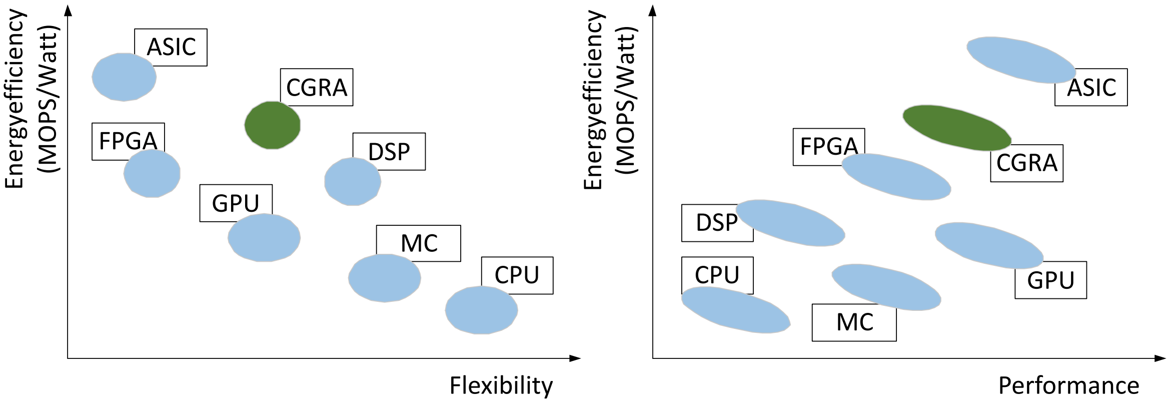


Fig. 3 Energy efficiency vs Flexibility and Performance of CGRA vs CPU, DSP, MC, GPU, FPGA and ASIC

- DSPs have superior energy efficiency to both CPU and GPU but lack scalable performance.
- MCs provide greater performance than CPU and have simpler cores, therefore greater energy efficiency than CPUs but inferior to GPUs.
- GPU energy efficiency has surpassed CPUs and multi-cores over recent years (however, GPUs still require multi-core CPUs to monitor them).
- Modern FPGAs with DSP slices offer superior energy efficiency to CPUs, MSPs, DSPs and GPUs, and provide strong performance.
- Nothing beats an ASIC for performance, but everything beats an ASIC for flexibility.
- CGRAs offer greater flexibility than FPGAs as they can be programmed efficiently in high level languages. They offer greater performance due to the coarse grained nature of the computation. Also, due to the coarser granularity, CGRAs need lesser reconfiguration time compared to FPGAs.

Contribution of the Thesis

The thesis contributes in the following aspects of employing CGRAs as accelerators in computing platforms.

1. CGRA design and implementation: The dissertation presents a novel CGRA architecture design referred to as Integrated Programmable Array (IPA) Architecture. The

design is targeted for ultra-low-power execution of kernels. The design also includes an extensive architectural exploration for best trade-off between cost and performance.

2. Mapping of applications onto the CGRA: For better performance instruction level parallelism is exploited at the time of compilation of the program, which maps operations and data onto the internal resources of CGRA. Efficient resource binding is the most important task of a mapping flow. An efficient utilization of available resources on CGRA plays a crucial role in enhancing performance and reducing the complexity of the control unit of the CGRA. As Register Files (RF) are one of the key components, efficient utilization of registers helps to reduce unwanted traffic between the CGRA and data memory. In this dissertation, we present an energy efficient register allocation approach to satisfy data dependencies throughout the program.

   Also, an accelerator without an automated compilation flow is unproductive. In this regard, we present a full compilation flow integrating the register allocation mapping approach to accelerate control and data flow of applications.

3. Support for control flow: Since control flow in an application limits the performance, it is important to carefully handle the control flow in hardware software co-design for the CGRA accelerator. On the one hand, taking care of the control flow in the compilation flow adds several operations increasing the chance of higher power consumption, on the other hand, implementing bulky centralized controller for the whole CGRA is not an option for energy efficient execution. In this thesis, we implement a lightweight control unit and synchronization mechanism to take care of the control flow in applications.

4. System level integration in a system on chip (SoC): Integration in a computing system is necessary to properly envision the CGRA as an accelerator. The challenge is interfacing with data memory due to scalability and performance issues. In this dissertation, we present our strategy to integrate the accelerator in the PULP [92] multi-core platform and study the best trade-off between the number of load-store units present in the CGRA and the number of banks present in the data memory.

Organization

The dissertation is organized into five chapters. In Figure 4, we show the positioning of the chapters based on the major features of the reconfigurable accelerator and compilation flow. Chapter 1 discusses the background with major emphasis on the state of the art works in architecture and compilation aspects. In chapter 2, the design and implementation of an

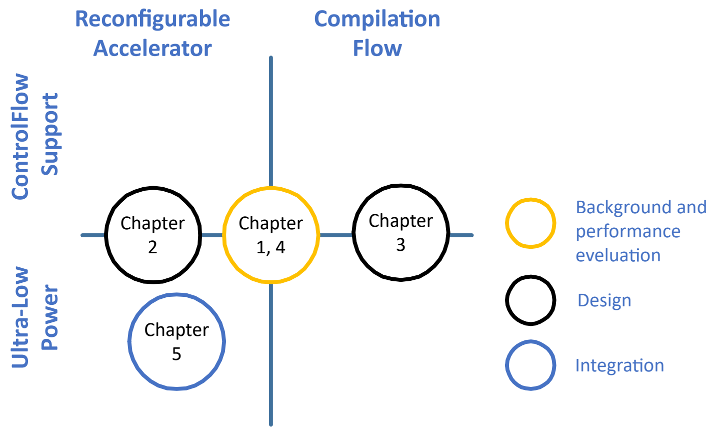

Reconfigurable Accelerator
Compilation Flow
ControlFlow Support
Ultra-Low Power
Chapter 2
Chapter 1, 4
Chapter 3
Chapter 5
Background and performance eveluation
Design
Integration

Fig. 4 Thesis overview

ultra-low power reconfigurable accelerator with the support for control flow management in applications is presented. Chapter 3 covers the compilation frame work for the accelerator. In chapter 4, we evaluate the performance of the accelerator and compilation flow. Chapter 5 addresses the integration of the accelerator in a multi-core SoC along with the software infrastructure.

Finally, the thesis concludes with an overview of the presented work and suggestions for the future research directions.

Chapter 1

Background and Related Work

As a consequence of power-wall, in combination with the ever-increasing number of transistors available due to Moore's Law, it is impossible to use all the available transistors at the same time. This problem is known as the *utilisation-wall* or *dark silicon* [34]. As a result, *energy efficiency* has become the first-order design constraint in all computing systems ranging from portable embedded systems to large-scale data-centers and supercomputers.

The research in the domain of computer architecture to tackle dark silicon, can be categorized into three disciplines.

- *Architectural heterogeneity:* To improve energy efficiency, modern chips tend to bundle different specialized processors or accelerators along with general purpose processors. The result is a system-on-chip (SoC) capable of general purpose computation, which can achieve high performance and energy efficiency occasionally for specialized operations, such as graphics, signal processing etc. The research in this domain identifies most suitable platforms like FPGA, CGRA, ASIC or hybrid, as accelerators. [6] [26] has already shown that accelerators in SoCs are useful for combating utilization wall.

- *Near threshold computing:* Low-voltage is indeed an attractive solution to increase energy efficiency, as the supply voltage has strong influence on both static and dynamic energy. Scaling down the supply voltage close to the threshold voltage (Vth) of the transistor is proven to be highly power efficient. [32] shows that near threshold operation achieves up to 5-10× energy efficiency improvement. Since decreasing supply voltage slows down the transistor, near threshold voltage operation aims to achieve a significant trade-off between performance and energy efficiency.

- *Power Management:* This category of research is concentrated on the architectures and algorithms to optimize the power budget. This involves introducing sleep modes [2] or dynamic voltage and frequency scaling (DVFS) [75], power and clock gating techniques.

The use of application-specific accelerators [36] designed for specific workloads can improve the performance given a fixed power budget. The wide use of such specialized accelerators can be realized in SoCs such as Nvidia's Tegra 4, Samsung's Exynos 5 Octa, Tilera's Gx72. However, due to the long design cycle, increasing design cost and limited re-usability of such application specific approach, reconfigurable accelerators have become attractive solution. FPGAs acting as reconfigurable accelerators [1] [22] [16], are widely used in heterogeneous computing platforms [5] [107] [60]. FPGAs have evolved from employing only matrix of configurable computing elements or configurable logic blocks connected by programmable interconnect to collection of processing macros, such as reconfigurable embedded memories and DSP blocks to improve the efficiency. FPGA accelerators are typically designed at RTL (Register Transfer Level) level of abstraction for best efficiency. The abstraction consumes more time and makes reuse difficult when compared to a similar software design. HLS tools [78] [19] [12] [65] have helped to simplify accelerator design by raising the level of programming abstraction from RTL to high-level languages, such as C or C++. These tools allow the functionality of an accelerator to be described at a higher level to reduce developer effort, enable design portability and enable rapid design space exploration, thereby improving productivity and flexibility.

Even though efforts, such as Xilinx SDSoC, RIFFA, LEAP, ReconOS, have abstracted the communication interfaces and memory management, allowing designers to focus on high level functionality instead of low-level implementation details, the compilation times due to place and route in the back-end flow for generating the FPGA implementation of the accelerator, have largely been ignored. Place and route time is now a major productivity bottleneck that prevents designers from using mainstream design based on rapid compilation. As a result, most of the existing techniques are generally limited to static reconfigurable systems [100]. Apparently, the key features like energy efficiency, ease of programming, fast compilation and reconfiguration motivate the use of CGRAs to address signal processing and high performance computing problems.

In this thesis, we explore CGRA as accelerator in a heterogeneous computing platform and create a research framework to fit the CGRA within an ultra-low power (mW) power envelope. In this chapter, we investigate the design space and compiler support of the state of the art CGRAs. As we move to the next chapters, we will consider the significant architectural features explored in this chapter to design and implement a novel, near-threshold CGRA and a compilation flow with a primary target to achieve high energy efficiency.

1.1 Design Space

CGRAs have been investigated for applications with power consumption profiles ranging from mobile (hundreds of mW) [28] to high performance (hundreds of W) [71]. The design philosophy differs in CGRAs to optimize and satisfy different goals. In the following, we present a set of micro-architectural aspects of designing CGRAs.

1.1.1 Computational Resources

The computational resources (CRs) or Processing Elements (PEs) in CGRAs are typically categorized as Functional Units (FUs) and Arithmetic Logic Units (ALUs) with input bit-widths ranging from 8-32. The FUs are of limited functionality. Specifically, few operations for a specific application domain, as used in ADRES [8] architectural templates. The ALUs feature complex functionality and require larger reconfiguration compared to that of FUs. Due to the versatility of ALU, they make instruction mapping easier. Depending on the application domain a full-custom processing element can be designed, which gives better area and performance, but extensive specialization can also lead to negative effects on energy consumption [105]. The PEs can be homogeneous in nature making instruction placement simple, whereas a heterogeneous PEs may present a more pragmatic choice.

1.1.2 Interconnection Network

There are several options for the interconnection network, such as programmable interconnect network, point-to-point, bus or a Network-on-Chip (NoC).

The programmable interconnection network consists of switches which need to be programmed to extract the desired behaviour of the network. This comes with a cost of configuration overhead. CREMA [38] CGRA consists of such programmable interconnect network.

A point-to-point (P2P) network directly connects the PEs and allows data to travel from one PE to only its immediate neighbours. To perform a multi-hop communication, one needs "move" instruction at each of the intermediate PE. These instructions are a part of the configuration. In contrast with the programmable interconnects, the P2P network does not impose any additional cost on programming the interconnect network. The most popular topologies in P2P network are mesh 2-D and bus based. Designs like MorphoSys [98], ADRES [8], Silicon Hive [10] feature a densely connected mesh-network. Other designs like RaPiD [20] feature a dense network of segmented buses. Typically, the use of crossbars is limited to very small instances because large ones are too power-hungry. SmartCell [64]

CGRA uses crossbar to connect the PEs in a cell. Since only 4 PEs are employed in each cell, the complexity of the crossbar is paid off by the performance in this design.

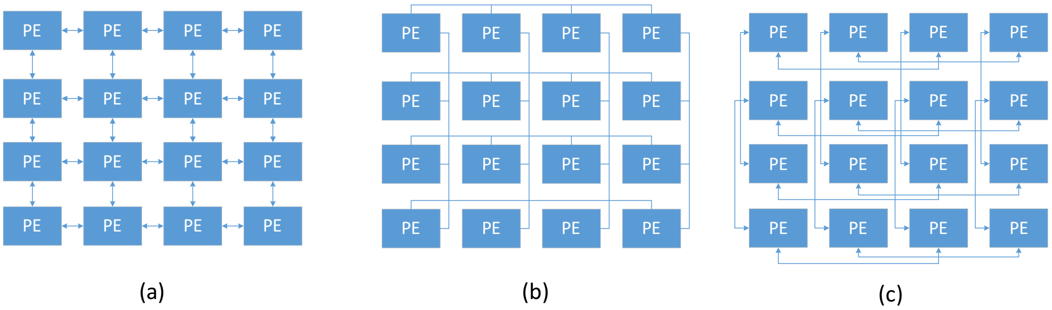


Fig. 1.1 Different interconnect topologies (a) Mesh 2D; (b) Bus based; (c) Next hop

A NoC provides a packet switched network where the router checks the destination field of the packet to be forwarded and determines the appropriate neighbouring router to which it needs to be forwarded. The NoC does not need to program the interconnect network but the hardware must implement a routing algorithm. Generally, cluster based CGRAs like REDEFINE [3], SmartCell use NoC to interact between the clusters. HyCube [52] implements a crossbar switched network. Unlike the NoC, this network implements clock-less repeaters to achieve single clock multi-hop communication.

Apart from these basic types, one may choose a hierarchical interconnection where different network types are used at different levels of the fabric. As an example, MATRIX [76] CGRA consists of three levels of interconnection network which can be dynamically switched. The SYSCORE [82] CGRA uses mesh and *cross interconnect* for low power implementations. In this architecture, cross interconnections are only introduced at odd numbered columns in the array of PEs, to avoid dense interconnect. The cross interconnections are useful to perform non-systolic functions.

1.1.3 Reconfigurability

Two types of reconfigurability can be realized in CGRAs: static and dynamic.

In statically reconfigured CGRAs, each PE performs a single task for the whole duration of the execution. The term "execution" here refers to the total running period between two configurations. In this case, mapping of the applications onto CGRA concerns only space, as illustrated in Figure 1.2. In other words, over times or cycles, the PE performs the same operation. The mapping solution assigns single operation to each PE depending on the data-flow. The most important advantage of static reconfigurability is the lack of reconfiguration overhead, which helps to reduce power consumption. Due to the lack of reconfigurability, the size of the CGRA becomes large to accommodate even small programs or need to break the large program into several smaller ones.

In dynamically reconfigured CGRAs, PEs perform different tasks during whole execution. Usually, in each cycle, the PEs are reconfigured by simple instructions which are referred to as context words. Dynamic reconfigurability can overcome the constraints over resources in static reconfigurability by expanding loop iterations through multiple configurations. Clearly, this comes with the cost of added power consumption due to consecutive instruction fetching. Designs like ADRES and MorphoSys tackle this by not allowing control flow in the loop bodies. Furthermore, if conditionals are present inside the loop body, the control flow is converted in data flow using predicates. This mechanism usually introduces overhead in the code. Liu et al in [67] performs affine transformations on loops based polyhedral model and able to execute up to 2 level deep loops.

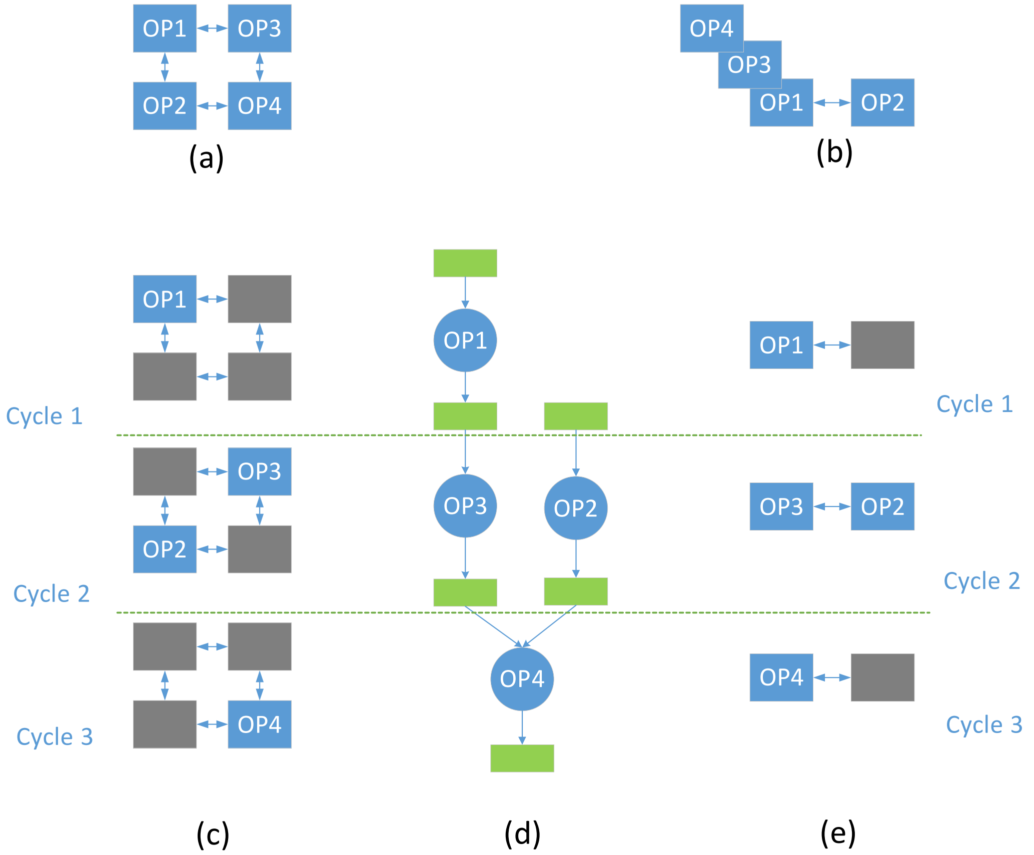


Fig. 1.2 Static and Dynamic Reconfigurability: (a) Mapping of the DFG in (d) for static reconfigurability onto a 2x2 CGRA; (b) Mapping of the DFG in (d) for Dynamic reconfigurability onto a 2x1 CGRA; (c) Execution of the statically reconfigurable CGRA in 3 cycles; (d) Data flow graph; (e) Execution of the dynamically reconfigurable CGRA in 3 cycles.

The MorphoSys design reduces the cost of instruction fetching further by limiting the code to Single Instruction Multiple Data (SIMD) mode of operation. All the PEs in a row or a column in this case execute same operation during the whole execution process. The similar approach is realized in SIMD-CGRA [33], where bio-medical applications are executed in an ultra-low-power environment. The RaPiD architecture limits the number of configuration bits to be fetched by making only a small part of the configuration dynamically reconfigurable. Kim et al [55] proposed to reduce the power consumption in the configuration memories by compressing the configurations.

Generally, a limited reconfigurability imposes more constraints on the types and sizes of loops that can be mapped. The compiler also needs to take extra burden to generate mappings satisfying the constraints. The Silicon Hive [10] is one such design which does not impose any restrictions on the code to be executed and allow execution of full control flow in an application. Unfortunately, no numbers on the power consumption are publicly available for this design.

The CGRA design in this thesis adopts the philosophy of unlimited reconfigurability that allows to map any kind of application consisting complex control and data flow in an energy constrained environment.

1.1.4 Register Files

CGRA compilers schedule and place operations in the computational resources (CR) and route the data flow over the interconnect network between the CRs. The data also travel through the Register Files (RF). Hence, the RFs in CGRA is treated as interconnects that can be extended over multiple cycles. As the RFs are treated for routing, compiler must know the location of RFs, their size and topology of interconnection with the CRs. Both power and performance depend on these parameters. Hence, while designing the CGRA, it is important to bear special attention to determine the size, number of ports location of the RFs.

1.1.5 Memory Management

While targeting low power execution, data and context management is of utmost importance. Over past years, several solutions [28] have been proposed to integrate CGRAs as accelerators with the data and instruction memory.

In many low-power targeted CGRAs [8][80][93][53], memory operations are managed by the host processor. Among these architectures, Ultra-Low-Power Samsung Reconfigurable Processor (ULP-SRP) and Cool Mega Array (CMA) operate in ultra-low-power (up to 3 mW) range. In these architectures, PEs can only access the data once prearranged in the shared register file by the processor. For an energy efficient implementation, the main challenge for these designs is to balance the performance of the data distribution managed by the CPU, and the computation in the PE array. However, in several cases, the computational performance of the PE array is compromised by the CPU, due to large synchronization overheads. For example, in ADRES [8] the power overhead of the VLIW processor used to handle the data memory access is up to 20%. In CMA [80] the host CPU feeds the data into the PEs through a shared fetch register (FR) file. This is very inefficient in terms of flexibility. The key feature of this architecture is the possibility to apply independent DVFS [101] or body biasing [73] to balance array and controlling processor parameters to adjust performance and bandwidth requirements of the applications. The highest reported energy efficiency for CMA is 743 MOPS/mW on 8-bit kernels, not considering the overhead of the controlling processor, which is not reported. With respect to this work, which only deals with DFG described with a customized language, we target 32-bit data and application kernels described in C language, which are mapped onto the array using an end-to-end C-to-CGRA compilation flow.

In a few works [98] [54] load-store operations are managed explicitly by the PEs. Data elements in these architectures are stored in a shared memory with one memory port per PE row. The main disadvantages of such data access architecture are: (a) lots of contention between the PEs on the same row to access the memory banks, (b) expensive data exchange between rows through complex interconnect networks within the array. With respect to these architectures, our approach minimizes contention by exploiting a multi-banked shared memory with word-level interleaving. In this way data-exchange among tiles can be performed either through the much simpler point-to-point communication infrastructure or fully flexible shared TCDM.

Solutions targeting high programmability and performance executing full control and data flows are reported for the weakly programmable processor array (WPPA) [56], Asynchronous Array of Simple Processors (AsAP) [108], RAW [103], ReMAP [21] and XPP [11]. The WPPA array consists of VLIW processors. For low power target the instruction set of a single PE is minimized according to domain-specific computational needs. In AsAP, each processor contains local data and instruction memory, FIFOs for tile-to-tile communication and local oscillator for local clock generation. Both the ReMAP and XPP consist of PE array each with DSP extension. These architectures are mainly intended for exploitation of task-level parallelism. Hence, each processor of the array must be programmed independently, which is much closer to many-core architectures. RAW PEs consist of 96 KB instruction cache and 32 KB data cache, router-based communication. These large-scale "array of processors" CGRAs are out of scope for ultra-low power, mW-level acceleration (a single tile would take more than the full power budget).

NASA's Reconfigurable Data-Path Processor (RDPP) [30] and Field Programmable Processor Array (FPPA) [31] are targeted for low-power stream data processing for spacecrafts. These architectures rely on control switching [30] of data streams and synchronous data flow computational model avoiding investment on memories and control. On the contrary, the IPA is tailored to achieve energy-efficient near sensor processing of data with the workloads very different from the stream data processing.

Table 1.1 summarizes an overview of the jobs managed by CGRA and the host processor for different architectural approaches. Acceleration of the kernels involves memory operations, innermost loop computation, outer loop computation, offload and synchronization with the CPU. As shown in the table, IPA manages to execute both the innermost and outer loops and the memory operations of a kernel imposing least communication and memory operation overhead while synchronizing with the CPU execution.

With respect to these state of the art reconfigurable arrays and array of processors, this thesis introduces a highly energy efficient, general-purpose IPA accelerator where PEs have random access to the local memory and execute full control and data flow of kernels on the array starting from a generic ANSI C representation of applications [24]. This work also focuses on the architectural exploration of the proposed IPA accelerator [25], with the goal to determine the best configuration of number of LSUs and number of banks for the shared L1 memory. Moreover, we employ a fine-grained power management architecture to eliminate dynamic power consumption of idle tiles during kernels execution which provides 2× improvement of energy efficiency, on average. The globally synchronized execution model, low cost but full-flexible programmability, tightly coupled data memory organization and fine-grained power management architecture define the suitability of the proposed architecture as an accelerator for ultra-low power embedded computing platforms.

Table 1.1 Qualitative comparison between different architectural approaches

Architectures	ADRES [8], CMA [73], MUCCRA [93], FPPA [31], RDPP [30], MATRIX [76], CHESS [72]	MorphoSys [98], RSPA [54], PipeRench [41], CHARM [17]	Liu et al [67]	IPA [25]
Memory operations	CPU	CGRA	CPU	CGRA
Innermost loop	CGRA	CGRA	CGRA	CGRA
Outer loop	CPU	CPU	CGRA	CGRA
Offload + Synchronization	CPU	CPU	CPU	CPU
Communication overhead	████	███	██	▌

1.2 Compiler Support

The compiler produces an executable for the reconfigurable fabric. As opposed to the compiler for general-purpose processors where only instruction set is visible to the compiler, the micro-architecture details of a CGRA are exposed to a CGRA compiler. This enables compilers to optimize applications for the underlying CGRA and take advantage of interconnection network, Register Files to maximize performance. Like the other compilers, the CGRA compilers generate an intermediate representation from the high level language. The intermediate representation is usually convenient for parallelism extraction, as most CGRAs have many parallel function units. Different architectures exploit different levels of parallelism through their compilers.

1.2.1 Data Level Parallelism (DLP)

The computational resources in this approach operate on regular data structures such as one and two-dimensional arrays, where the computational resources operate on each element of the data structure in parallel. The compilers target accelerating DLP loops, vector processing or SIMD mode of operation. CGRAs like Morphosys, Remarc, PADDI leverage SIMD architecture. The compilers for these architectures target to exploit DLP in the applications. However, DLP-only accelerators face performance issues while the accelerating region does not have any DLP, i.e. there are inter iteration data dependency.

1.2.2 Instruction Level Parallelism (ILP)

As for the compute intensive applications, nested loops perform computations on arrays of data, that can provide a lot of ILP. For this reason, most of the compilers tend to exploit ILP for the underlying CGRA architecture.

State of the art compilers which tend to exploit the ILP, like RegiMap [45], DRESC [74], Edge Centric Modulo Scheduling (EMS) [81] mostly rely on software pipelining. This approach can manage to map the innermost loop body in a pipelined manner. On the other hand, for the outer loops, CPU must initiate each iteration in the CGRA, which causes significant overhead in the synchronization between the CGRA and CPU execution. Liu et al in [67] pinpointed this issue and proposed to map maximum of two levels of loops using polyhedral transformation on the loops. However, this approach is not generic as it cannot scale to an arbitrary number of loops. Some approaches [66] [62] use loop unrolling for the kernels. The basic assumption for these implementations is that the innermost loop's trip count is not large. Hence, the solutions end up doing partial unroll of the innermost loops. The outer loops remain to be executed by the host processor. As most of the proposed compilers handle innermost loop of the kernels, they mostly bank upon the partial predication [48] [13] and full predication [4] techniques to map the conditionals inside the loop body.

Partial predication maps instructions of both if-part and else-part on different PEs. If both the if-part and the else-part update the same variable, the result is computed by selecting the output from the path that must have been executed based on the evaluation of the branch condition. This technique increases the utilization of the PEs, at the cost of higher energy consumption due to execution of both paths in a conditional. Unlike partial predication, in full predication all instructions are predicated. Instructions on each path of a control flow, which are sequentially configured onto PEs, will be executed if the predicate value of the instruction is similar with the flag in the PEs. Hence, the instructions in the false path do not get executed. The sequential arrangement of the paths degrades the latency and energy efficiency of this technique.

Full predication is upgraded in state based full predication [47]. This scheme prevents the wasted instruction issues from false conditional path by introducing sleep and awake mechanisms but fails to improve performance. Dual issue scheme [46] targets energy efficiency by issuing two instructions to a PE simultaneously, one from the if-path, another from the else-path. In this mechanism, the latency remains similar to that of the partial predication with improved energy efficiency. However, this approach is too restrictive, as far as imbalanced and nested conditionals are concerned. To map nested, imbalanced conditionals and single loop onto CGRA, the triggered long instruction set architecture (TLIA) is presented in [68]. This approach merges all the conditions present in kernels into triggered instructions and creates instruction pool for each triggered instruction. As the depth of the nested conditionals increases the performance of this approach decreases. As far as the loop nests are concerned, the TLIA approach reaches bottleneck to accommodate the large set of triggered instructions into the limited set of PEs.

1.2.3 Thread Level Parallelism

To exploit TLP, compilers partition the program into multiple parallel threads, each of which is then mapped onto a set of PEs. Compilers for RAW, PACT, KressArray leverage on TLP. To support parallel execution modes, the controller must be extended for supporting the call stack and synchronizing the threads. As a result, power consumption is increased.

The TRIPS controller supports four operation modes of operation to support all the types of parallelism [96]. The first mode is configured to execute single thread in all the PEs, exploiting ILP. In the second mode, the four rows execute four independent threads exploiting TLP. In the third mode, fine-grained multi-threading is supported by time-multiplexing all

PEs over multiple threads. In the fourth mode each PE of a row executes the same operation, thus implementing SIMD, exploiting DLP. Thus, the TRIPS compiler can exploit the most suited form of parallelism.

The compiler for REDEFINE exploits TLP and DLP to accelerate a set of HPC applications. Table 1.2 presents an overview of several architectural and compilation aspects of the state of the art CGRA designs.

Table 1.2 CGRA design space and compiler support: CR - Computational Resources; IN - Interconnect Network; RC - Reconfigurability; MM - Memory management; CS - Compilation Support; MP - Mapping; PR - Parallelism

<table>
<tr><th colspan="7">High performance targets</th></tr>
<tr><th rowspan="2">Architecture</th><th rowspan="2">CR</th><th rowspan="2">IN</th><th rowspan="2">RC</th><th rowspan="2">MM</th><th colspan="2">CS</th></tr>
<tr><th>MP</th><th>PR</th></tr>
<tr><td>RAW</td><td>RISC core</td><td>Hybrid</td><td>Static and dynamic</td><td>FU</td><td>CDFG</td><td>TLP</td></tr>
<tr><td>TRIPS</td><td>ALU</td><td>NoC</td><td>Dynamic</td><td>FU</td><td>CDFG</td><td>ILP, DLP, TLP</td></tr>
<tr><td>REDEFINE</td><td>ALU</td><td>Hybrid</td><td>Dynamic</td><td>FU</td><td>CDFG</td><td>DLP, TLP</td></tr>
<tr><td>ReMAP</td><td>DSP core</td><td>Programmable</td><td>Dynamic</td><td>FU</td><td>CDFG</td><td>TLP</td></tr>
<tr><td>MorphoSys</td><td>ALU</td><td>Hybrid P2P</td><td>Dynamic</td><td>FU</td><td>DFG</td><td>DLP, ILP</td></tr>
<tr><th colspan="7">Low-power targets</th></tr>
<tr><td>ADRES</td><td>FU</td><td>Hybrid P2P</td><td>Dynamic</td><td>VLIW host</td><td>DFG</td><td>ILP</td></tr>
<tr><td>Smartcell</td><td>ALU</td><td>Hybrid</td><td>Dynamic</td><td>PE</td><td>CDFG</td><td>TLP, DLP</td></tr>
<tr><td>PACT XPP</td><td>ALU</td><td>Hybrid</td><td>Dynamic</td><td>PE</td><td>CDFG</td><td>TLP</td></tr>
<tr><td>TCPA</td><td>ALU</td><td>Hybrid</td><td>Dynamic</td><td>PE</td><td>CDFG</td><td>TLP</td></tr>
<tr><td>AsAP</td><td>ALU</td><td>Mesh 2D</td><td>Dynamic</td><td>PE</td><td>CDFG</td><td>TLP</td></tr>
<tr><td>MUCCRA-3</td><td>FU</td><td>Hybrid P2P</td><td>Dynamic</td><td>VLIW host</td><td>DFG</td><td>ILP</td></tr>
<tr><td>RaPiD</td><td>ALU</td><td>Bus based</td><td>Static and dynamic</td><td>PE</td><td>DFG</td><td>DLP</td></tr>
<tr><th colspan="7">Ultra-low-power targets</th></tr>
<tr><td>CMA</td><td>ALU</td><td>Hybrid P2P</td><td>Dynamic</td><td>Host micro controller</td><td>Data flow</td><td>ILP</td></tr>
<tr><td>ULP-SRP</td><td>FU</td><td>Mesh-x</td><td>Dynamic</td><td>VLIW host</td><td>Data flow</td><td>ILP</td></tr>
<tr><td>SYSCORE</td><td>ALU</td><td>Hybrid</td><td>Dynamic</td><td>DSP host</td><td>Data flow</td><td>DLP</td></tr>
</table>

1.3 Mapping

Mapping process assigns data and operations onto the CGRA resources. The process comprises of scheduling and binding of the operations and data on the functional units and registers respectively. Depending upon how these two steps are realized, mapping process can be divided into two classes.

The first one solves scheduling and binding sequentially using heuristics and/or meta-heuristics [61] [81] [37] or exact methods [44] [45]. [81] and [37] implement heuristic based iterative modulo scheduling [87] approach. In [81] an edge-based binding heuristic is used (instead of classical node-based approaches) to reduce the number of fails. In [37], binding problem is addressed by combining a routing heuristic from FPGA synthesis and a Simulated Annealing (SA) algorithm for placement. Schedulable operation nodes are moved randomly according to a decreasing temperature and a cost function. To efficiently explore the solution space, the temperature needs to be high. Slow decrease in the probability of accepting worse solutions effects the computation time. [61] proposes to improve the computation time, by finding a solution on a simplified problem with heuristic-based methods for both scheduling and binding at first and then trying to improve the initial solution with a genetic algorithm. However, the use of only one seed limits the ability to explore the whole solution space. Mapping proposed in EPIMap [44] and REGIMap [45] solve the scheduling and the binding problems sequentially by using a heuristic and an exact method respectively. Scheduling is made implicit by integrating both architectural constraints (i.e. the number of operations simultaneously executable on the CGRA and the maximum out-degree of each operation due to the communication network) and timing aspect into the DFG by statically transforming it. Binding is addressed by finding the common sub-graph between the transformed DFGs and a time extended CGRA with Levi's algorithm [63]. However, since the graph transformations are done statically, it becomes difficult to know which transformation is relevant at a given time. This reduces the ability of the method to efficiently explore the solution space since the problem is over-constrained. Mapping proposed using graph minor approach in [15] also uses graph transformation and sub-graph matching to find the placement.

The second category solves the scheduling and binding problem concurrently. The mappings proposed in [61], [9], [84] use exact methods, e.g. ILP-based algorithms, to find the optimal results. Due to the exactness of the approaches, these methods suffer from scalability issues. DRESC [74] and its extension [27] that can cope with RFs, leverage on metaheuristics. These are based on a Simulated Annealing (SA) framework that includes a guided stochastic algorithm. The classical placement and routing problem which can be solved with SA, is extended in three dimensions to include scheduling. Thus, schedulable operation nodes are moved randomly through time and space. Therefore, the convergence is even slower than for other SA based methods as it includes scheduling.

The key idea of the mapping approach in this work is to combine the advantages of exact, heuristic and meta-heuristic methods while offsetting their respective drawbacks as much as possible. Hence, as detailed in chapter 4, scheduling and binding problems are solved simultaneously using a heuristic-based algorithm and a randomly degenerated exact method respectively and transforming the formal model of the application dynamically when necessary.

1.4 Representative CGRAs

In this section, we present five well-known and representative CGRA architectures chosen due to their unique network model, functional units and memory hierarchy.

1.4.1 MorphoSys

The MorphoSys reconfigurable array (Figure 1.3) consists of an 8×8 grid of reconfigurable cells (RCs), each of which contains an ALU, RF and multiplier. The interconnect network consists of four quadrants that are connected in columns and rows. Inside each quadrant a dense mesh network ((Figure 1.3 (a)) is implemented. At the global level, there are buses that support inter-quadrant connectivity ((Figure 1.3 (b)). The context memory stores multiple contexts which are broadcast to row or column-wise providing SIMD functionality. Each unit is configured by a 32-bit configuration word. A compiler is developed based on extended C language, but partitioning is performed manually.

1.4.2 ADRES

The ADRES architecture (Figure 1.4) comprises of an array of PEs tightly coupled with a VLIW processor. The reconfigurable cells (RC) consist of ALU and register file and tiny instruction memory. RCs are connected through a mesh interconnection network. A predication network is implemented to execute conditionals. The register file in VLIW processor is shared with the RC array. This reduces the communication between reconfigurable matrix and memory subsystem. ADRES features a C compiler for both VLIW and CGRA.

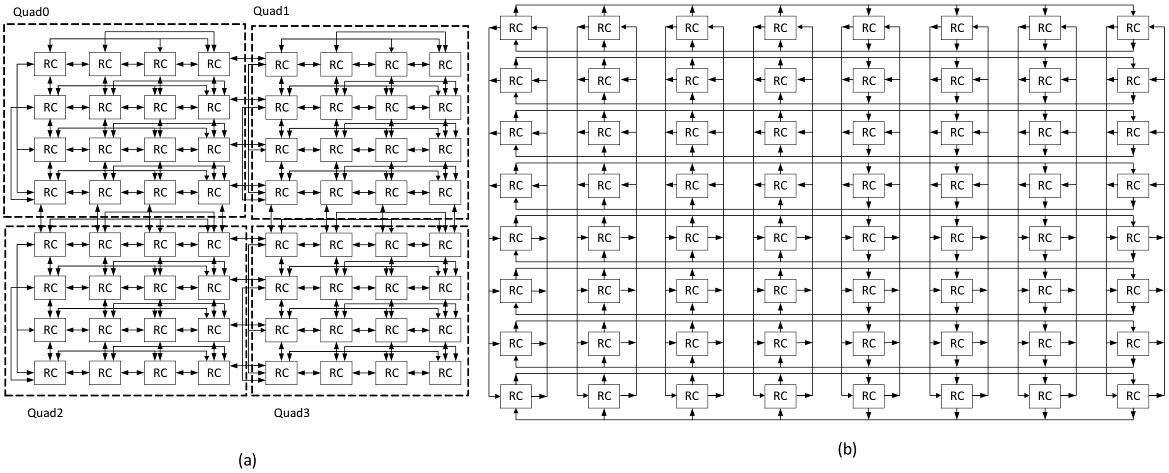

Quad0
Quad1
Quad2
Quad3
RC
(a)
(b)

Fig. 1.3 Morphosys Architecture

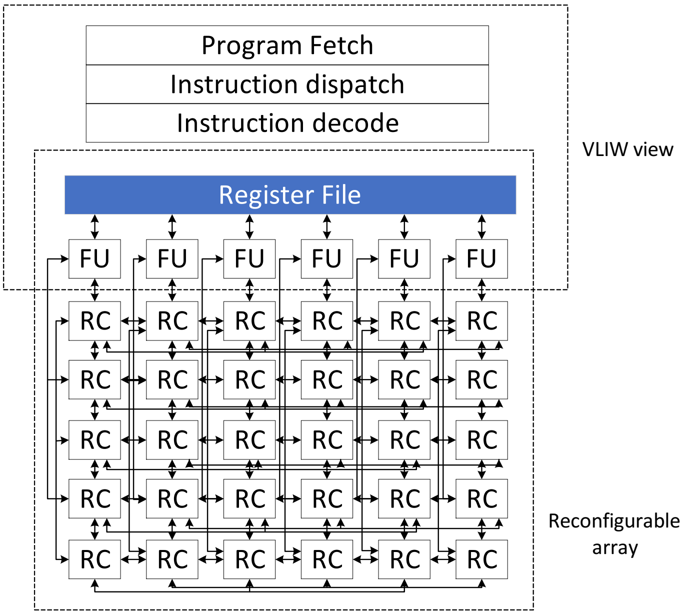

Program Fetch
Instruction dispatch
Instruction decode
VLIW view
Register File
FU
RC
Reconfigurable array

Fig. 1.4 ADRES Architecture

1.4.3 RAW

The RAW architecture (Figure 1.5) is an array of RISC-based pipelined FUs. Each FU consists of instruction and data memory. The FUs communicate via a programmable interconnect network. Each FU is connected to switch that controls the destination addresses used by the network interface, hence, the routing can be statically scheduled. When no data transfer is scheduled on the network, the instruction scheduled dynamically by RAW control unit can utilize the network as a dynamic one. A compiler based on a high-level language implementing TIERS based [97] place and route is available.

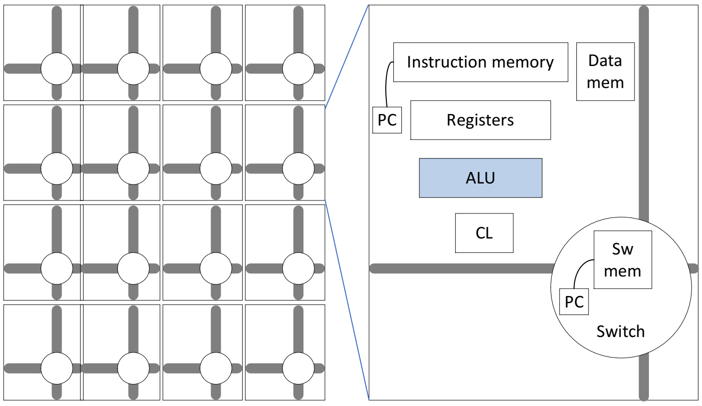


Fig. 1.5 RAW Architecture

1.4.4 TCPA

TCPA (Figure 1.6) consists of an array of heterogeneous, VLIW-style FUs, connected via a programmable network. The heterogeneity of the FUs is a design-time decision. The interconnect between the FUs is statically configured and forms direct connections between the FUs. Each FU has a (horizontal and vertical) mask that allows individual reconfiguration of FUs. In this way, SIMD type behaviour can also be implemented. Unlike conventional VLIW processors, the register files in these FUs are explicitly controlled. Compilation for the architecture is introduced in [102] where algorithms are described in PAULA language [49], designed for multi-dimensional data intensive applications.

1.4.5 PACT XPP

PACT XPP defines two types of processing array elements (PAE). One for computation and another with local RAM. The PAEs are connected with a packet-based network and computation is event driven. Figure 1.7 presents the architecture of a PAE. The typical

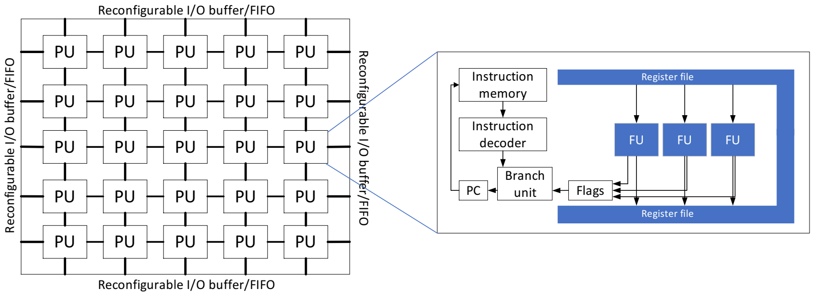

Reconfigurable I/O buffer/FIFO
Reconfigurable I/O buffer/FIFO
Reconfigurable I/O buffer/FIFO
Reconfigurable I/O buffer/FIFO
PU
Instruction memory
Instruction decoder
Branch unit
PC
Flags
Register file
FU
FU
FU
Register file

Fig. 1.6 TCPA Architecture

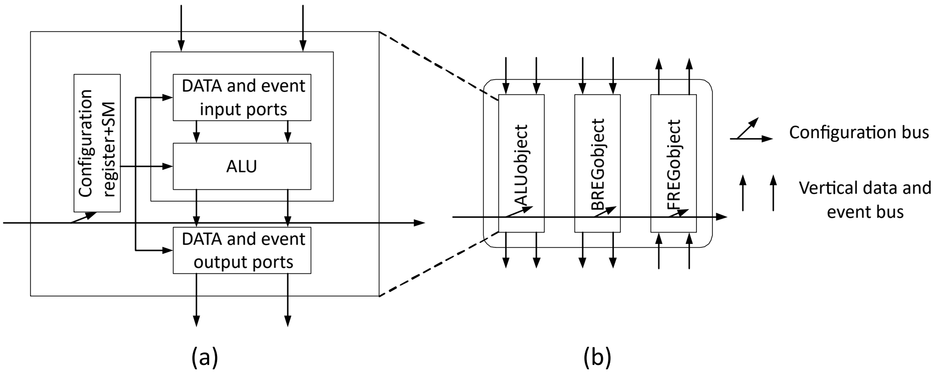

Configuration register+SM
DATA and event input ports
ALU
DATA and event output ports
ALUobject
BREGobject
FREGobject
Configuration bus
Vertical data and event bus
(a)
(b)

Fig. 1.7 PAE architecture of PACT XPP

PAE contains a back registers (BREG) object and forward register (FREG) object which are used for vertical routing, as well as an ALU object which performs the actual computations. Both the operation of the PAEs and the communication are reconfigurable resulting in a large number of configuration bits. The event driven compute model means the control flow is handled in a decentralized fashion such that a configuration can be kept static as long as possible. To support irregular computations that do require to update the configuration, PACT XPP uses two techniques. Firstly, configurations are cached locally to enable fast configuration and secondly, partial configurations are supported. Partial configurations only update selected bits, which can keep them small in many cases, optimizing the use of the local configuration cache.

1.5 Conclusion

This chapter presented an overview of different CGRAs and their execution models. Different architectural and compilation approaches have been presented for a comprehensive view of wide spectrum of the design and compilation. In the next chapter, we make design choices and focus on implementing CGRA operating in ultra-low power domain.

Chapter 2

Design of The Reconfigurable Accelerator

Due to the increasing complexity of near-sensor data analytics algorithms, low power embedded applications such as Wireless Sensor Networks (WSN), Internet of Things (IoT) and wearable sensors combine the requirement of high performance and extreme energy efficiency in a *mW* power envelope [7]. While traditional ultra-low power sensor processing circuits rely on hardwired Application Specific Integrated Circuit (ASIC) architectures [29], near-threshold parallel computing is emerging as a promising solution to exploit the energy boost given by low-voltage operation while recovering the related performance degradation through execution over multiple programmable processors [91].

Even though exploitation of parallel ultra-low power computing provides maximum flexibility, a dominating majority of the power consumed during processing is linked to the typical overheads of instruction processors [39], such as complex fetching and decoding of instructions, control and data-path pipeline overheads (up to 40%), and the load and store overhead needed for processors to work with their L1 memory (up to 30%).

In this chapter, we make significant step forward in parallel near-threshold computing toward the goal of achieving the energy efficiency of application-specific data-paths, by exploiting the Coarse Grain Reconfigurable Array (CGRA) architectural template and revisiting it to fit within an ultra-low power (mW) power envelope. Some of the primary objectives that motivates highly flexible ULP CGRA design are discussed in the following.

- **Flexibility:** Flexibility is the key accomplishment relying on a reconfigurable fabric. However, along the design path, there are several compromises made to satisfy design constraints. As an example, the RaPiD [20] architecture limits the number of configuration bits to be fetched by making a small part of the configuration reconfigurable per cycle. The MorphoSys design reduces the reconfiguration overhead by limiting the supported code to SIMD. To achieve better energy efficiency, other CGRA design which support MIMD, like ADRES [8], CMA [101], ULP-SRP [53] relies on executing the innermost loop only to avoid the control flow hazards. If conditionals exist in the innermost loop, they are tackled in software by flattening them using several predication techniques.

All the restrictions are mostly addressed in the design entry point, where a high level language is used to program the CGRAs. As a result, restrictions in reconfigurability eventually leads to the programmability issues. As previously mentioned, most CGRAs use C language as the entry point, ideally, the designs and their compilers should be able to find a mapping for any valid C program. In practice, this is not the case: only the loops, particularly the innermost loops are mapped onto the CGRAs. In addition, the designs use a subset of C language. In other words, they do not support use of pointer-based access, recursions etc. However, well-structured loops can be written without these structures, but the fact of re-engineering the source code remains.

As flexibility is our primary design philosophy, we prioritize executing any C program, not just the loop kernels. This is achieved by implementing low cost control flow support in the hardware, and an efficient control flow mapping support in the compiler (see chapter 3).

- **Utilization:** One of the most critical design choices is the processing element. Since high performance is achieved by exploiting *parallelism*, choice of the computing unit (FU, ALU) is of paramount importance. FUs are of limited functionality, hence, the reduced area of each unit allows to have a larger number of them in a fabric. However, given an interconnect network, the interaction between the FUs gets limited. For illustration, let us assume a CGRA which comprises FUs interconnected through a mesh-torus topology. Each FU interacts with four of its neighbours. For better utilization, the types of these four FUs must be chosen carefully depending upon the certain instruction sequence in the application domain. If the instruction sequence does not match the type of the neighbouring FUs, it will result multi-hop communication to other FUs. This eventually leads to less utilization of the FUs resulting bottleneck for exploiting parallelism. Use of full fledged ALU supports wide range of functionality increasing the utilization and chance of better exploiting parallelism.

  Another design goal is to achieve high energy efficiency, which is achieved by the *data locality*. In other words, data must stay as close as possible to the computing unit. As, registers are the closest possible storage unit to the processing part, high utilization of register files increases the possibility of achieving high energy efficiency. There are two kinds of data in an application except the regular array inputs and outputs: the *recurring variables* (repeatedly written and read) and the *constants*. In this thesis, we show that register files can be efficiently used to store the recurring variables present not only in the loops but in an application. The existing designs use the shared memory to store the constants. Indeed, storing constants in the memory helps to reduce the instruction width or configuration overhead, but accessing shared/central register file [8] or memory [98] results in higher latency, and increase in the number of load-stores, degrading performance and energy efficiency. In this work, we take care of constants by introducing the concept of constant register file (CRF) which helps local access of constants at the time of execution.

- **Interconnect Network:** Since energy efficiency is first order design constraint in the thesis, the focus is on low power choices for the interconnect networks.

  Both for static and dynamic reconfigurable CGRAs, there are two phases involved in the execution process. The first one is the *configuration phase*, when the fabric is reconfigured partially or fully. The second one is the *compute phase*, when computations are performed on the data. To efficiently support the phases there must be two interconnect networks involved: (a) network to distribute the instruction, (b) network to support the data flow. Depending on the computation model, frequency of performing configuration and compute phase, and ratio between their effective time must be analysed for choosing the ideal interconnect network.

  To give a clear perspective, first, we consider the case for a statically configurable CGRA, where each PE executes single instruction in the whole execution. In this case, the ratio between the computation time and configuration time is usually high. In other words, much time is spent on the computation compared to the configuration. Hence, for computation, low cost interconnect networks (i.e. mesh 2D) provide better energy efficiency. Since configuration or instructions are supplied to the PEs at once, high cost interconnect network can be afforded for better performance. If the size of the CGRA is small, then it may require frequent configuration. In this case, the better choice for instruction delivery network may be a bus-based network [50].

  In the dynamically reconfigurable CGRAs, there may be two types of arrangements. The first one uses a centralized configuration memory. The configurations or instructions are accessed by the PEs from the centralized memory in each cycle. Due to centralized access of instructions, the PEs must access the configurations frequently. Since the configuration phase is frequent, it is convenient to merge the instruction and data distribution networks. In other words, the same interconnect network can be used for both instruction and data distribution. However, if they are performed simultaneously, then loading of instructions affects the data movement due to the use of shared resources. Certainly, some of these conflicts can be avoided through appropriate choice of placement and routing of data and instructions onto the CGRA. Many designs like SmartCell, TRIPS employ NoC as a unified network, which gives a great flexibility in routing data and instructions. In NoC, the destination is specified as a part of the packet and it is then routed based on a hard-wired routing algorithm. However, the flexibility comes with a cost of added power consumption in hardware routing and composing/decomposing of packets.

If the PEs consist of local configuration or instruction memories, the solution can be arranged differently for efficiency. Although the configuration happens in every cycle, no interconnect network is involved to deliver them to the PEs. Instead, the configuration memories are filled prior to the execution starts. Hence, the solution for filling the configuration or instruction memories can be viewed as streaming the instructions before starting the compute stage in statically reconfigurable CGRA. Hence, the arrangements of the network may also be similar.

- **Instruction set architecture:** It is essential to keep each processing element small to maximize the number of processing parts that can fit on a chip. Employing simple instruction set architecture helps to minimize the cost of instruction fetching and decoding.

- **Control flow support:** Acceleration of applications generally depends on efficient computation of the innermost loop kernel. Usually, the host processor takes in charge of initiating the outer loops. This scheme requires regular communication with the host processor, which in turn increases the synchronization overhead. For low power target, it becomes essential for the accelerator to have support for control flow, in order to minimize the communication with the host and synchronization overhead.

- **Compilation:** Automated compilation tools are required alongside the hardware designs, which map applications to the target architectures. A good compilation tool must exploit data locality references (see Chapter 4) for better energy efficiency.

2.1 Design Choices

Based on the discussions presented above, our reconfigurable fabric is designed as an interconnection of typically 4×4 processing elements consisting ALUs. We employ a mesh-torus based network for the data flow and a bus-based interconnection network for configuration. As, the size of the CGRA, interconnection topology and RF size are important dimensions of the architecture, we performed experiments to support the design choices.

Nine applications from signal processing domain have been used for our experiments (Table 2.1). We have used fully unrolled version of these applications. The increased code size of the applications after full unrolling helps to understand how the limit in the size of the CGRA, local RFs and interconnect network impacts on the performance. As, the data flow graphs (DFG) of the fully unrolled applications are mapped onto the CGRA, we consider ASAP (As-soon-as-possible schedule) length of the DFGs as the best performance metric. The cycles taken by the CGRA to compute the DFG will be similar to the ASAP length of the DFG if the particular configuration can exploit all the parallelism (maximum number of operations present in a cycle) available in the application.

Table 2.1 Characteristics of the benchmarks used for design space exploration

Benchmark	nodes	ASAP	Parallelism
2D Discrete Cosine Transform (DCT-2D)	711	81	32
matrix product	504	98	32
Fast Fourier Transform (FFT)	1348	37	64
Trapizoidal (Trapez) filter	332	59	32
Exponential Moving Average Filter (EMA)	412	99	38
Moving Window De-convolution (MWD)	440	112	32
Unsharp Mask	91	27	16
Elliptic Filter	130	31	16
DC Filter	507	96	32

In the experiments, we consider CGRAs with different dimensions (3×3, 4×4, 5×5), with different RF sizes (6, 8, 16, 24), and with different P2P topology (mesh torus, mesh-x, fully connected). Figure 2.1 represents performance of a 3×3 CGRA with different RF sizes and topologies, normalized to the ASAP length of the application DFGs. Similarly, Figure 2.2 and 2.3 presents the performance analysis in 4×4 and 5×5. Latencies closer to the ASAP value represents better performance. A normalized latency of value 0.5 means that the specific configuration is unable to find a mapping solution.

The performance trend is similar to other dimensions of the CGRA, except the fact that graph with higher dimensions consist less number of bars with 0.5 normalized latency, which implies that finding mapping solutions is highly probable in CGRAs with higher dimensions, as expected. However, for this extensive set of experiments the 4×4 CGRA with RF size of 8 is able to find solutions for all the applications with the minimum overhead possible among all the combinations of CGRA configurations. Fig. 2.2 shows that increasing the RF size does not result revolutionary performance gain. After a certain RF size (which depends on the application), the performance does not increase. With the increased interconnection complexity, performance is enhanced, but the small performance gains are not encouraging enough to go for a more complex solution.

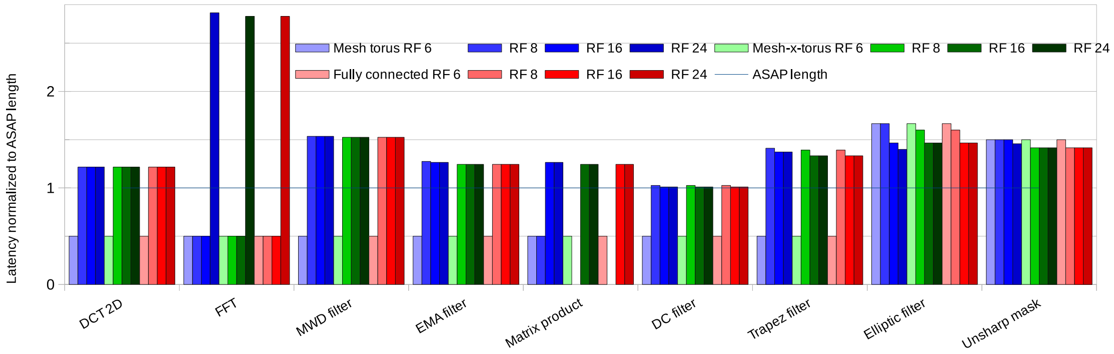


Fig. 2.1 Latency comparison for 3×3 CGRA

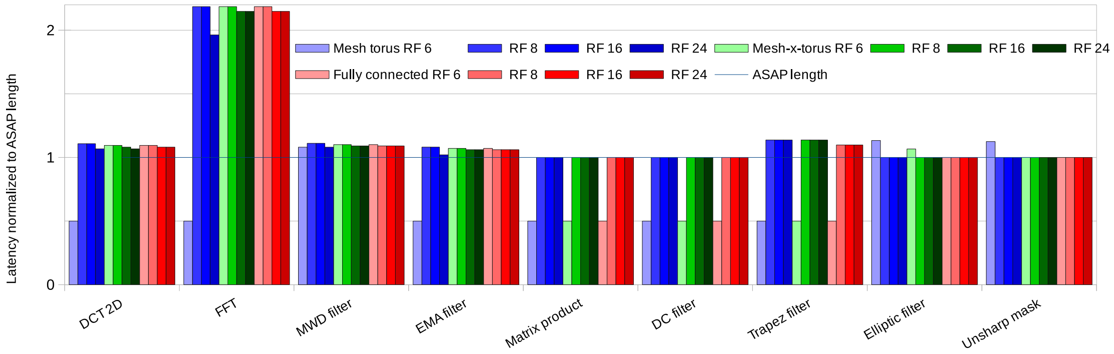


Fig. 2.2 Latency comparison for 4×4 CGRA

With the choice of a 4×4 CGRA with RF size of 8 and mesh-torus topology we move forward to design the novel CGRA architecture referred to as Integrated Programmable Array (IPA) [25].

To cope with the ultra-low power profile and memory sharing challenges, IPA involves a multi-bank Tightly Coupled Data Memory (TCDM) coupled with a flexible and configurable memory hierarchy for data storage. As shown in Figure 2.4, from an architectural viewpoint,

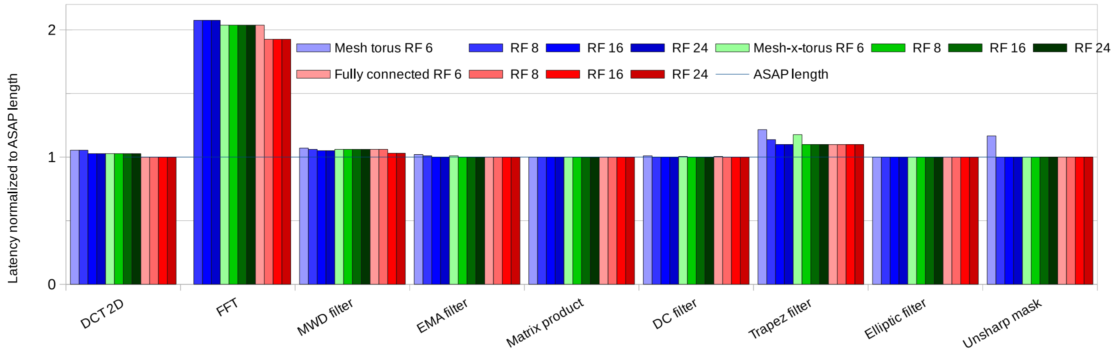


Fig. 2.3 Latency comparison for 5×5 CGRA

point-to-point data communication between processing elements (PEs) (Figure 2.4(b)) during kernel execution, represents a key advantage over energy-hungry data sharing over shared memory that is required when using a traditional processor-cluster architecture (Figure 2.4(a)) for parallel processing. Table 2.2 shows that the IPA cluster performs a lower number of memory operations on the sample program presented in the Listing 2.1, which in turn gives energy improvement of 1.3× over the clustered multi-core architecture, which performs data sharing through the TCDM. In this comparison, we even ignore the barrier synchronization overheads in the many-core cluster for the sake of simplicity.

The IPA approach allows to significantly reduce the pressure on L1 memory. Hence, it requires a smaller number of banks to achieve low contention [91]. As opposed to clustered multi-core architectures, where data-exchange among cores is managed through shared data structures and OpenMP parallel processing constructs, in CGRAs the compiler must take care of data-exchange among PEs by exploiting point-to-point connections among the PEs as much as possible to minimize shared memory accesses.

```
for(i=0; i<1; i++)
{
A[i] = B[i] * C[i]
}
for(i=0; i<1; i++)
{
sum = sum + A[i];
}
```

Listing 2.1 Sample program to execute in the multi-core and IPA cluster

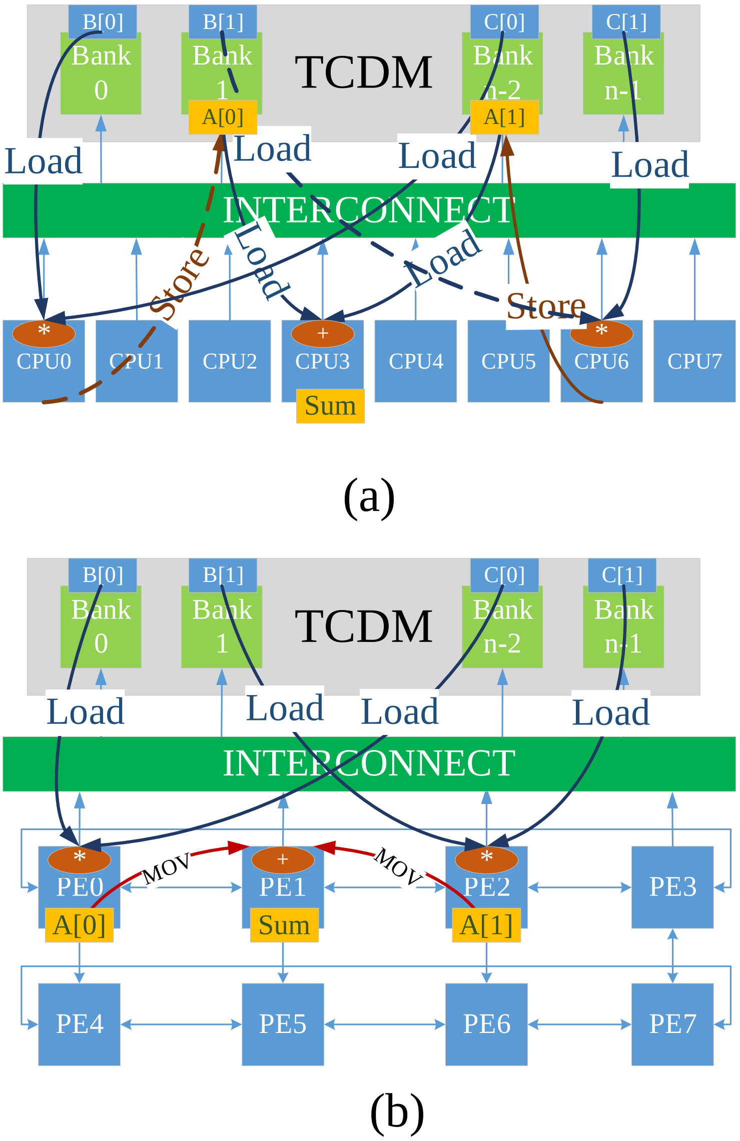

B[0]
B[1]
C[0]
C[1]
Bank 0
Bank 1
Bank n-2
Bank n-1
TCDM
A[0]
A[1]
Load
Store
INTERCONNECT
*
+
CPU0
CPU1
CPU2
CPU3
CPU4
CPU5
CPU6
CPU7
Sum
(a)
MOV
PE0
PE1
PE2
PE3
PE4
PE5
PE6
PE7
(b)

Fig. 2.4 (a) multi-core Cluster and (b) IPA cluster executing the sample program in Listing 2.1.

Table 2.2 Energy consumption comparison between multi-core and IPA while executing the sample program in Figure 2.4(a)

Average energy consumption in pJ/operation					
	Load-Store operations	**Arithmetic operations**	**MOV operation**		
	4.2	3.4	3.1		
Total energy consumption					
	Total #Load-Store	**Total #Arithmetic**	**Total #MOV**	**Energy (pJ)**	**Gain**
Multi-Core	8	3	0	43.8	-
IPA	4	3	2	33.2	1.3x

2.2 Integrated Programmable Array Architecture

The architecture comprises a PE array, a global context memory, a controller, a tightly coupled data memory (TCDM) with multiple banks and a logarithmic interconnect. Figure 2.5 shows the organization of the IPA. In the following we discuss the components of the IPA fabric.

2.2.1 IPA components

Global Context Memory (GCM)

The configurations for the PEs are stored in the Global Context Memory. Prior to the computation starts in the PE array, the configurations are loaded into the PEs through the bus-based network. The configurations contain instructions and the non-recurring variables which are stored into the instruction register file and constant register file of each PE respectively.

IPA Controller (IPAC)

The IPA controller identifies configuration data for the corresponding PE and transfers it in the *load context* stage. It also initiates the execution phase after loading all the contexts. The IPAC handles the important task of synchronizing with the host processor which will be discussed in the following chapter where we integrate the IPA in a multi-core platform.

PE Array (PEA)

The PE Array follows the multiple instruction, multiple data (MIMD) model of computation. All PEs operate on different set of instructions. A bus based interconnect network is implemented to load instructions and constants (i.e. context) from the GCM into the PEs, whereas

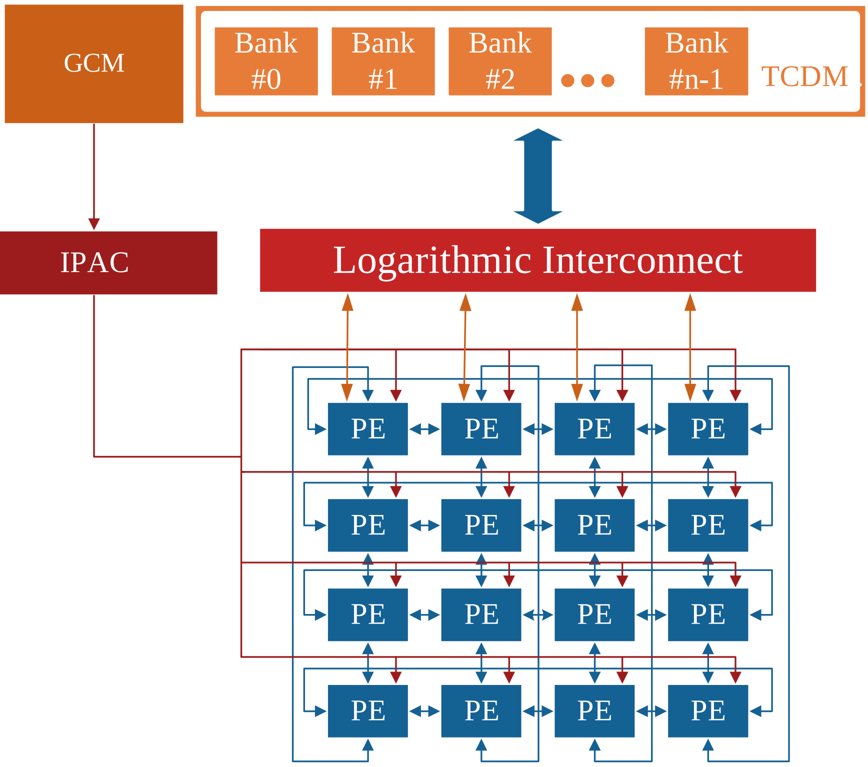


Fig. 2.5 System level description of the Integrated Programmable Array Architecture

the torus network is used during execution phase for low power data communication between the PEs. The details of the load context protocol are discussed later in this chapter. To achieve low power execution, the instruction set architecture was designed from the scratch resulting 20-bit long instruction. We took the advantage of the visibility of the micro-architecture to the compiler and shifted the immediate data to constant register file in the PEs (discussed later) which eases the compression of the instruction, imposing low pressure on the decoder. The details of components of the PEs are discussed in the following.

The PE array consists of a parametric number of PEs (the optimal number of PEs is studied in section 2.1), connected with mesh torus network for the data flow and a bus-based network for instruction distribution. Figure 2.6 describes the components of a PE. Two Muxes (IN0 and IN1) selects the inputs of each PE. The input sources are the neighbouring PEs and the register file. A 32-bits ALU and a 16-bits x 16-bits = 32-bits multiplier are employed in this block. The Load Store Unit (LSU) is optional for the PEs (the optimal number of LSU is a parameter studied later in this chapter). The Control unit is responsible for fetching the instruction from the corresponding address of the instruction memory and managing program flow. The Constant Register File (CRF) stores the non-recurring values or constants, while the Regular Register File (RRF) and Output Register (OPR) store the recurring variables.

Control flow support: In order to minimize the synchronization overhead with the host processor, the PEs support branch instructions for executing loops and conditionals. The Controller in the PE fetches the instructions from the Instruction Register File (IRF). If the decoded instruction is a *jump*, the target address of the *jump* is stored in the Jump Register (JR). The *cjump* (conditional jump) instruction contains two target addresses. The true path is evaluated in the JR by the Boolean OR of the Condition Register (CR) bits of the PEs.

Power Management Unit (PMU)

To reduce dynamic power consumption in idle mode, each PE contains a tiny Power Management Unit (PMU) which clock gates the PEs when idle. An idle condition for a PE arises from three situations: (i) Unused PE: when a PE is not used during mapping; (ii) Load Store stall: In case of TCDM banking conflict the PMU generates a *global stall*, which is broadcast to all the PEs. Until the global stall is resolved, all the PEs are clock gated by their corresponding PMUs. LSUs are placed in the global clock region (Figure 2.6) to avoid deadlocks; (iii) Multiple NOP operations: a NOP instruction contains the number of successive NOPs. When a NOP instruction is fetched, the decoder loads this number into a counter within the PMU. The *clockgate_en* remains low until the count reaches zero. The counter gets halted when it encounters a global stall and resumes the count after the stall is resolved, synchronizing the execution flow among PEs.

Due to the fine-grained nature of the power management, more aggressive power gating is not implemented, since it imposes large area penalty without significant benefits. Since the leakage power of each tile is so small that does not change significantly the energy efficiency when the rest of the system is active.

TCDM and logarithmic interconnect

The TCDM acts as L1 memory for the IPA. Featuring a number of ports equal to the number of memory banks, it provides concurrent access to different memory locations. The TCDM is interfaced with the LSUs of the PE array through a low latency, logarithmic interconnect [85], implementing a word level interleaving scheme to minimize access contention. To optimize the performance and energy efficiency, we explore the IPA architecture with special focus on shared memory access in the next section.

2.2.2 Computation Model

After compiling a kernel (see the Chapter 5), the compiler generates the assembly and the addresses for the input and output data in the local shared memory. The assembler takes the

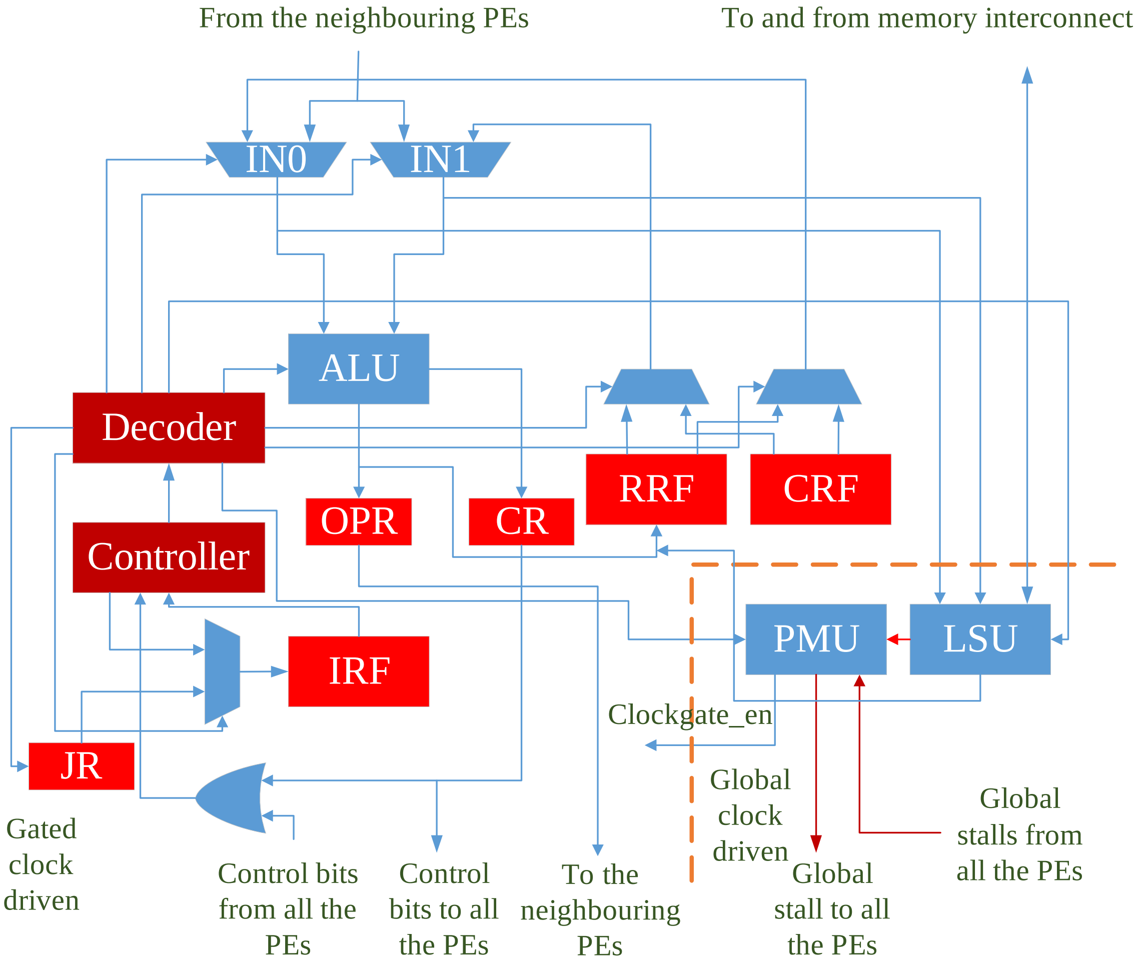

From the neighbouring PEs
To and from memory interconnect
IN0
IN1
ALU
Decoder
Controller
OPR
CR
RRF
CRF
IRF
JR
PMU
LSU
Clockgate_en
Gated clock driven
Control bits from all the PEs
Control bits to all the PEs
To the neighbouring PEs
Global clock driven
Global stall to all the PEs
Global stalls from all the PEs

Fig. 2.6 Components of PE

assembly and the Instruction Set Architecture (ISA) of the IPA, to generate the context (i.e. the program to be stored into the IRF) for each PE, which is pre-loaded in the GCM. The context contains instructions and constants for each PE in the array. Prior to the execution start, the context is loaded into the corresponding IRF and CRF of the PEs. We assume that the code fits in the local memory. Larger execution contexts can be handled using the IPA controller and overlays.

Load context

Figure 2.7 shows the configuration network to load the context in each PE. In each cycle of this stage the *IPAC* receives the context word from the GCM and broadcasts to the PEs. For the PEs with same instruction, *broadcast mode* is used to distribute instruction to a set of PEs. To load PEs with non-identical set of instructions and constants *normal addressing mode* is used. The organization of context word in these two modes are described in the following.

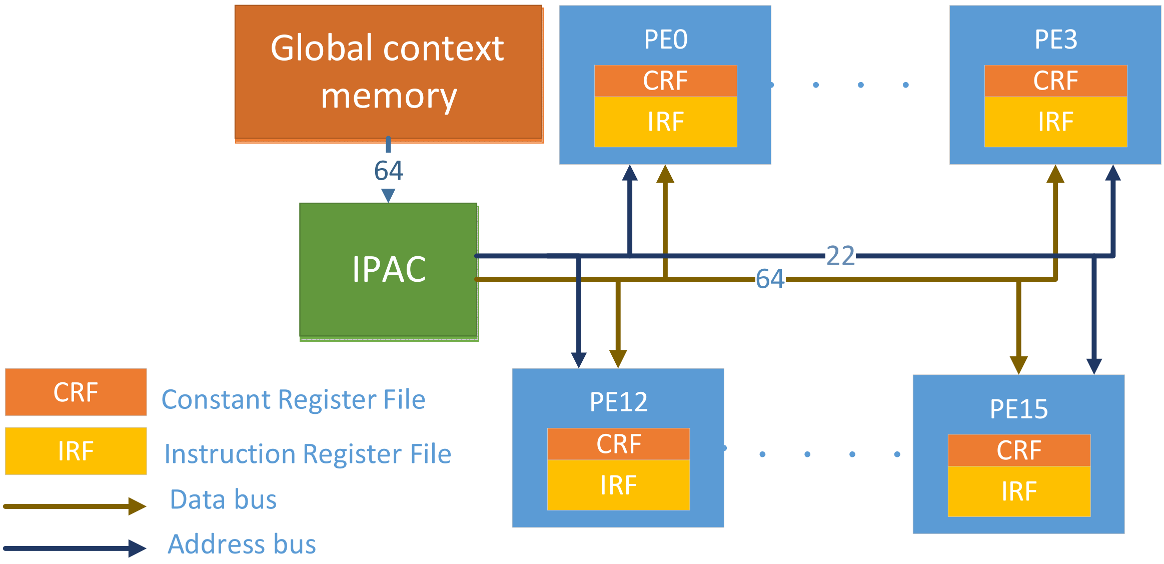


Fig. 2.7 The configuration network for load-context

The GCM (Figure 2.8) contains the context of the PEs. Each address in the GCM contains a 64-bits context word. To distinguish between several sets of the instructions and constants, the GCM is divided into several *segments* (table 2.3), where each *segment* contains a set of instructions and constants to be broadcast or normally addressed. The first bit in each segment represents whether the next set of instructions is addressed to broadcast (0) or normal addressing mode (1). In broadcast mode following 16 bits represent the mask, where the position of the high bits represents the addresses of the PEs to be broadcast. For normal addressing mode, only 4 bits are used to address the target PE.

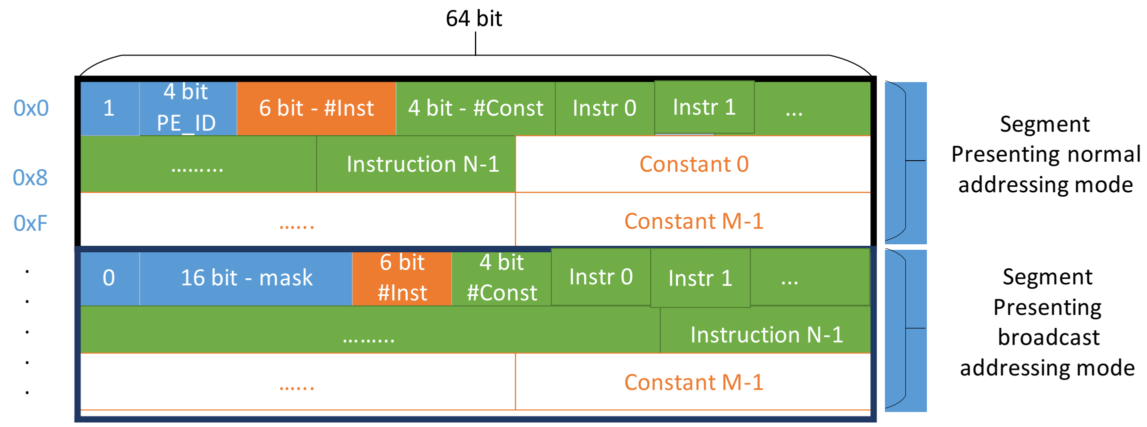

64 bit
0x0
0x8
0xF
1
4 bit PE_ID
6 bit - #Inst
4 bit - #Const
Instr 0
Instr 1
...
....
Instruction N-1
Constant 0
......
Constant M-1
0
16 bit - mask
6 bit #Inst
4 bit #Const
Instr 0
Instr 1
...
....
Instruction N-1
......
Constant M-1
Segment Presenting normal addressing mode
Segment Presenting broadcast addressing mode

Fig. 2.8 Segments of the GCM

Table 2.3 Structure of a segment

Number of bits	**Encoded information**
1	Addressing modes
4/16	Normal Address/Mask
6	Total number of instructions (N)
4	Total number of constants (M)
20xN	Instructions
32xM	Constants

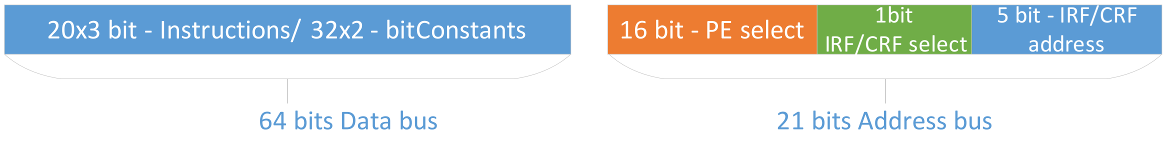

20x3 bit - Instructions/ 32x2 - bitConstants
16 bit - PE select
1bit IRF/CRF select
5 bit - IRF/CRF address
64 bits Data bus
21 bits Address bus

Fig. 2.9 Format of the data and address bus in the configuration network

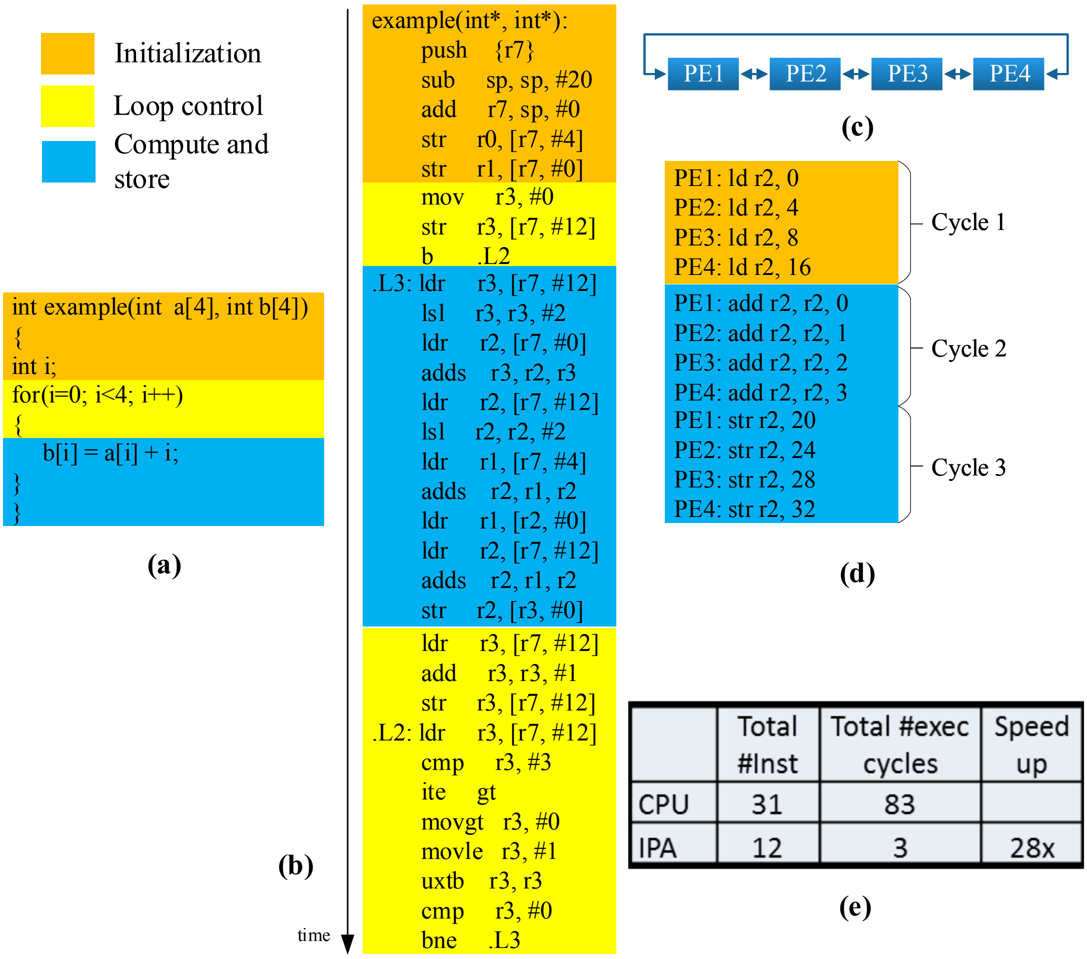
The format of the *address* and *data* bus in the configuration network is presented in Figure 2.9. The *address bus* encodes 22 bits of information containing the 16 bits mask or address of the target PE, 1 bit to select IRF or CRF followed by 5 bits address. The 64 bits data bus consists of 20×3 bits instruction or 32×2 bits constant.

Execution

In every cycle, each PE fetches 20-bits instruction from the local IRF. Table 2.4 describes the instruction format. The first field in the instruction is used to present the *opcode*, which is of 5 bits width supporting a maximum of 32 different operations. Details of supported operations are in Table 2.5.

Table 2.4 Instruction format

5 bits	**2 bits**	**3 bits**	**1 bit**	**4 bits**	**1 bit**	**4 bits**
Opcode	Output Reg type	Dest Reg Addr	IN0 Type	IN0 Addr	IN1 Type	IN1 Addr
Jmp	Address		unused			
Cjmp	Address of the true path		Address of the false path		unused	
NOP	Number of consecutive NOPs		unused			



```
int example(int  a[4], int b[4])
{
int i;
for(i=0; i<4; i++)
{
    b[i] = a[i] + i;
}
}
```

```
example(int*, int*):
      push    {r7}
      sub     sp, sp, #20
      add     r7, sp, #0
      str     r0, [r7, #4]
      str     r1, [r7, #0]
      mov     r3, #0
      str     r3, [r7, #12]
      b       .L2
.L3:  ldr     r3, [r7, #12]
      lsl     r3, r3, #2
      ldr     r2, [r7, #0]
      adds    r3, r2, r3
      ldr     r2, [r7, #12]
      lsl     r2, r2, #2
      ldr     r1, [r7, #4]
      adds    r2, r1, r2
      ldr     r1, [r2, #0]
      ldr     r2, [r7, #12]
      adds    r2, r1, r2
      str     r2, [r3, #0]
      ldr     r3, [r7, #12]
      add     r3, r3, #1
      str     r3, [r7, #12]
.L2:  ldr     r3, [r7, #12]
      cmp     r3, #3
      ite     gt
      movgt   r3, #0
      movle   r3, #1
      uxtb    r3, r3
      cmp     r3, #0
      bne     .L3
```

```
PE1: ld r2, 0
PE2: ld r2, 4
PE3: ld r2, 8
PE4: ld r2, 16
PE1: add r2, r2, 0
PE2: add r2, r2, 1
PE3: add r2, r2, 2
PE4: add r2, r2, 3
PE1: str r2, 20
PE2: str r2, 24
PE3: str r2, 28
PE4: str r2, 32
```

	Total #Inst	Total #exec cycles	Speed up
CPU	31	83	
IPA	12	3	28x

Fig. 2.10 (a) Sample program (b) Execution in CPU (c) Example PEA (d) Execution in IPA (e) Execution metrics in CPU and IPA

Figure 2.10 shows the execution of a sample program in a traditional CPU and the IPA. The total number of instructions for the sample program in the CPU and the IPA are 31 and 12 respectively. Also, the IPA achieves 28× performance gain compared to that of the CPU while executing the sample program. The decrease in the number of instructions in the IPA in this specific example is mainly due to the much lower number of memory operations and the fact that the small loop can be completely unrolled without code size blown-up.

2.3 Conclusion

In this chapter, we presented the design of a CGRA targeting ultra-low power computing. The proposed *Integrated Programmable-Array* (IPA) is a 2-D array of NxN processing elements involving two layers of interconnect network. The context distribution network uses a bus-based solution for better performance, while the data distribution network uses a mesh-torus based solution for better energy efficiency. The proposed design leverages a multi-banked tightly coupled data memory for data storage to ease the integration in clustered multi-core architectures. The compilation flow for the IPA is presented in the next chapter. The succeeding chapter evaluates the performance and energy efficiency along with the implementation of the IPA.

Table 2.5 Summary of the opcodes (R = Result, C = Condition bit)

Mnemonic	**Opcode**	**Instruction**	**Operation**
NOP	0x00	No operation	-
UADD	0x01	Unsigned addition	$R = (U)\ Op1+Op2$ $C = 0$
SADD	0x02	Signed addition	$R = Op1+Op2$ $C = 0$
SSUB	0x03	Signed subtraction	$R = Op1-Op2$ $C = 0$
SMUL	0x04	Signed multiplication	$R = Op1*Op2$ $C = 0$
LS	0x06	Shift left	$R = Op1 \ll Op2$ $C = 0$
RS	0x06	Shift right	$R = Op1 \gg Op2$ $C = 0$
LD	0x07	Load	-
STR	0x09	Store	-
AND	0x0b	Bit-wise AND	$R = Op1\&Op2$ $C = 0$
OR	0x0c	Bit-wise OR	$R = Op1 \vert Op2$ $C = 0$
NOT	0x0d	Bit-wise NOT	$R = \overline{Op1}$ $C = 0$
XOR	0x0e	Bit-wise XOR	$R = Op1 \oplus Op2$ $C = 0$
MOV	0x0f	Copy input to output	$R = Op1$ $C = 0$
LTE	0x10	Conditional less than equal	$if\,(Op1<=Op2)$ $C = 1$ $else$ $C = 0$
GTE	0x11	Conditional greater than equal	$if\,(Op1>=Op2)$ $C = 1$ $else$ $C = 0$
NE	0x12	Conditional not equal	$if\,(Op1!=Op2)$ $C = 1$ $else$ $C = 0$
EOC	0x1f	End of computation	-

Chapter 3

Compilation flow for the Integrated Programmable Array Architecture

Over the last twenty five years, CGRAs have been an active field of research. However, the lack of efficient and automated compiler prevents widespread use of the CGRAs. As opposed to the general purpose computing platforms, the micro-architecture of a CGRA must be visible to the compiler to be able to improve performance by extracting the advantages of the underlying interconnect network and distribution of register files.

In this chapter, we discuss about the design of a compiler to map programs onto a CGRA specifically for the IPA. First, we present the background and the problems for mapping applications onto CGRAs. Then, we study the design of the compiler based on a CGRA model, which can be varied to accommodate a wide range of CGRA designs.

3.1 Background

As discussed earlier the compiler must know the underlying architecture of the CGRA, it takes two inputs. The first is the architecture model (PE array (PEA) of the IPA), and the second is the application described by a high level language, in our case it is ANSI-C code of the application.

3.1.1 Architecture model

The PEA is modelled by a bipartite directed graph with two types of nodes: operators and registers. Timing is implicitly represented by connections between registers and operators, which is referred to as the *time extended model* of the PEA [45]. Two types of operator nodes are defined for the PEAs. The first type is the computing operator (functional unit (FU) nodes in Figure 3.1(a)) that represents the physical implementation of an arithmetic and logical operation (+, ×, -, OR, AND) and/or memory access (e.g. load/store). The second type of operator is the memorization operator (circular nodes in Figure 3.1(b)). It is associated with the output register and represents the operation of keeping a value in a local register explicitly.

Figure 3.1 (a) shows a sample PEA with two PEs connected by a torus network. Each PE has 3 registers in the distributed register file, and a single output register. Figure 3.1 (b) represents the *time extended model* of the PEA shown in Figure 3.1 (a).

In this model, one can vary the interconnect network, the distribution and size of the register file, and the type of the PE, to explore different PEA designs.

3.1.2 Application model

The application is modelled as a control and data flow graph (CDFG). Supporting control flow gives the opportunity to accelerate a kernel without any intervention of the host processor. A CDFG is depicted as $G = (V,E)$ where V is the set of basic blocks and $E \subseteq V \times V$ is the set of directed edges representing control flow. A Basic Block (BB) is represented as a data flow graph (DFG) or $BB = (D,O,A)$ where D is the set of data nodes, O is the set of operation nodes and A is the set of arcs representing dependencies. The control flow from one basic block to another is supported with jump (*jmp*) and conditional jump (*cjmp*) instructions.

Figure 3.2 shows the CDFG representation of the sample program presented in Listing 3.1. In the figure, basic blocks are represented as blue rectangles. The flow from one basic block to another basic block is represented by black arrows and managed by simple branch (*jmp*) operation. The true and false paths of a conditional managed by *cjmp*, are shown by solid and dashed arrows respectively. The execution flow of the CDFG is presented as: $BB\_1 \rightarrow BB\_2$ $\rightarrow$ (either BB_3 or BB_8) if $BB_3 \rightarrow BB\_4 \rightarrow$ (either BB_5 or BB_6) $\rightarrow BB\_7 \rightarrow BB\_2 \cdots$.

In order to maintain the execution flow, it is necessary to synchronize all the PEs in the array, to the execution of the same basic block. When the execution flow jumps from one basic block to another, all the PEs in the PEA must be synchronized to the current basic block execution. This allows to use all the PEs concurrently or sequentially, while executing a single basic block, since only one basic block is executed at a time. Dually, several basic blocks can use the same PE. The synchronized execution allows the compiler to map several operations and data onto the same PE. Next, we present the homomorphism of the CDFG model with the application model, to support different stages in the compilation flow.

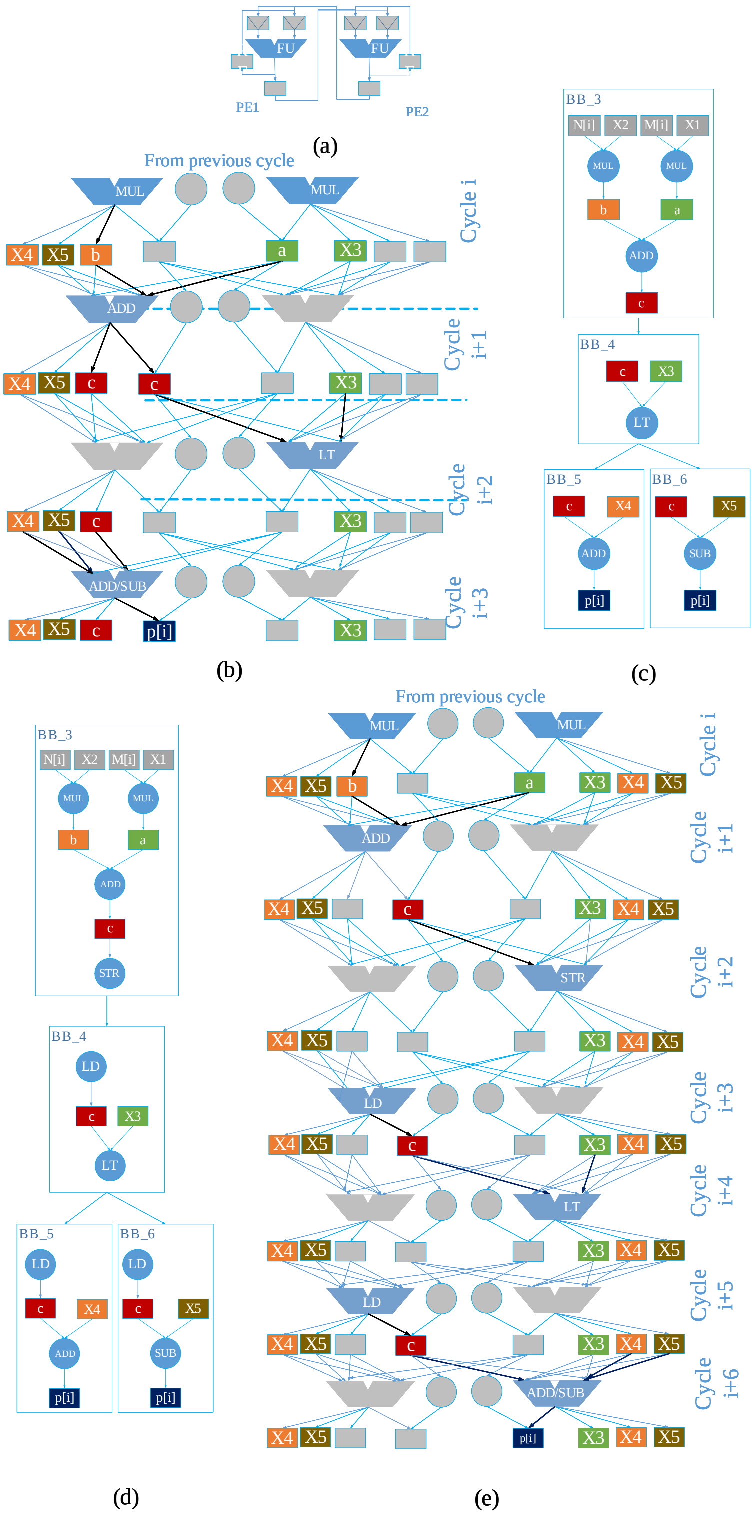

FU
FU
PE1
PE2
(a)
From previous cycle
MUL
MUL
X4
X5
b
a
X3
ADD
c
LT
ADD/SUB
p[i]
Cycle i
Cycle i+1
Cycle i+2
Cycle i+3
(b)
BB_3
N[i]
X2
M[i]
X1
MUL
MUL
b
a
ADD
c
BB_4
c
X3
LT
BB_5
c
X4
ADD
p[i]
BB_6
c
X5
SUB
p[i]
(c)
BB_3
N[i]
X2
M[i]
X1
MUL
MUL
b
a
ADD
c
STR
BB_4
LD
c
X3
LT
BB_5
LD
c
X4
ADD
p[i]
BB_6
LD
c
X5
SUB
p[i]
(d)
From previous cycle
MUL
MUL
ADD
STR
LD
LT
LD
ADD/SUB
p[i]
Cycle i
Cycle i+1
Cycle i+2
Cycle i+3
Cycle i+4
Cycle i+5
Cycle i+6
(e)

Fig. 3.1 (a) A 2×1 PEA with 3 registers in RF and one output register (c) CDFG model (b) A possible mapping of (b) onto the PEA over 4 cycles using register allocation based approach. (d) The transformed CDFG of (b) for systematic load store based approach (e) A possible mapping of (d) onto the PEA over 7 cycles using systematic load store based approach

```
//Sample program to demonstrate CDFG model
X1 = 10;
X2 = 20;
X3 = 500;
X4 = 30;
X5 = 50
for(i = 0; i < q; i++)
{
 a = m[i] * X1;
 b = n[i] * X2;
 c = b + a;
 if(c < X3)
  p[i] = c + X4;
 else
  p[i] = c - X5;
}
```

Listing 3.1 Sample program with control flow

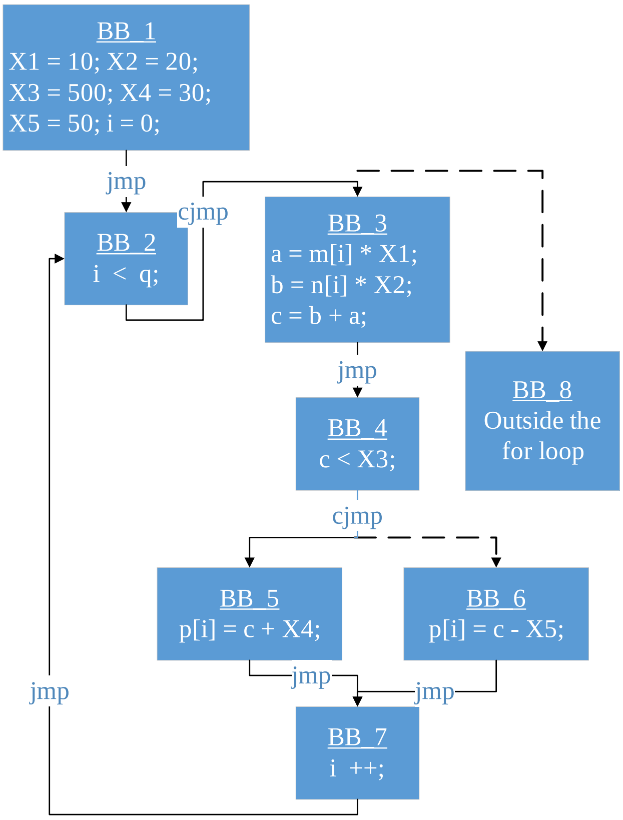

BB_1
X1 = 10; X2 = 20;
X3 = 500; X4 = 30;
X5 = 50; i = 0;
jmp
BB_2
i < q;
cjmp
BB_3
a = m[i] * X1;
b = n[i] * X2;
c = b + a;
jmp
BB_4
c < X3;
BB_8
Outside the for loop
cjmp
BB_5
p[i] = c + X4;
BB_6
p[i] = c - X5;
jmp
jmp
jmp
BB_7
i ++;

Fig. 3.2 CDFG representation of the sample program in 3.1

3.1.3 Homomorphism

The basic blocks in the CDFG, presented in Figure 3.1(c), are composed of data nodes, operation nodes, and data dependencies. Three equivalences between the basic block DFGs and PEA model nodes are defined: (1) data and registers; (2) computation and computing operators; (3) data dependences and connection between the time extended PE components. As the two models are homomorphic, the mapping of a DFG onto the PEA is therefore a problem equivalent to finding a DFG in the PEA graph.

Figure 3.1(b) represents a possible mapping of the sample CDFG in Figure 3.1(c) onto the PEA in Figure 3.1(a) over 4 cycles.

3.1.4 Supporting Control Flow

One of the major challenges associated with all accelerators is to effectively handle control flow in the applications. Since the goal of the compiler presented in this chapter is to execute a complete program efficiently onto a CGRA, by *control flow*, we do not only mean the conditionals which are present inside a loop body, but any conditional or unconditional branch in general. For better understanding, we classify the control flow into three categories as presented in Figure 3.3. The *unconditional branch*es can be optimized by merging basic blocks or *straightening* which is applicable to pairs of basic blocks such that the first has no successors other than the second and the second has no predecessors other than the first. If there exists more than one basic block in a program after optimization, which is often the case, the **underlying accelerator must support unconditional branch** to avoid host interference.

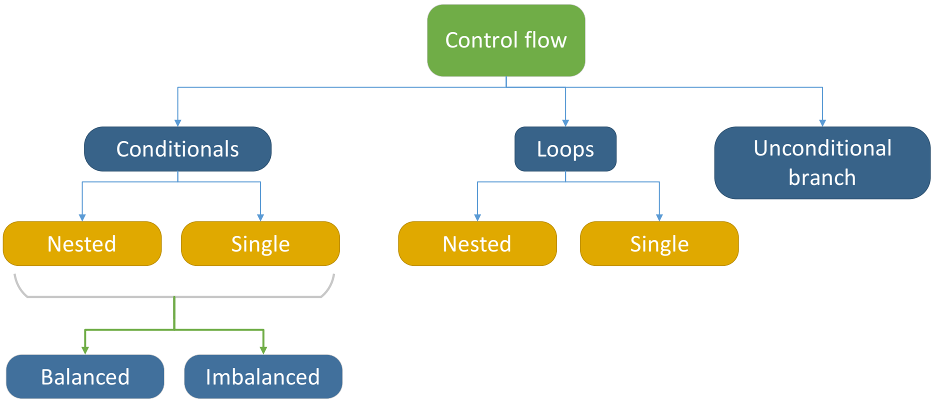


Fig. 3.3 Classification of control flow present in an application

The fundamental problem for the *conditionals* are outcome of the branch at runtime. Hence, effective resource allocation is a problem. Hardware accelerators and FPGAs executes both the paths of a conditional branch in parallel, and then choose the results of the true path. This results in waste of resources and power. GP-GPUs also schedule the instructions and allocated resources for both the paths of the conditionals, but at the runtime, instructions from the false path are not issued. This saves power, but the cycles and resources allocated for the not-taken path are still wasted. In the graphics processing community, this is referred to as the problem of *branch divergence*. CGRAs widely use predication techniques to deal with the conditionals. Fundamentally partial, and full predication are adapted by the compilers, which are now discussed along with some other notable schemes.

Partial predication

Since conditionals are constructed by if-then-else (ITE), in partial predication, the operations of both the if-part and the else-part are mapped on different PEs. If the same variable needs to be updated in both the if-part and the else-part, the final result is computed by selecting the output from the true path, which is decided at runtime. This is achieved through a special operation, named *select*, which takes in the result of the branch condition from predicates[1], and two updated values of the variable to select the correct one. If a variable is to be updated in only one path, a select operation is still necessary to maintain the validity of the variable for the upcoming cycles.

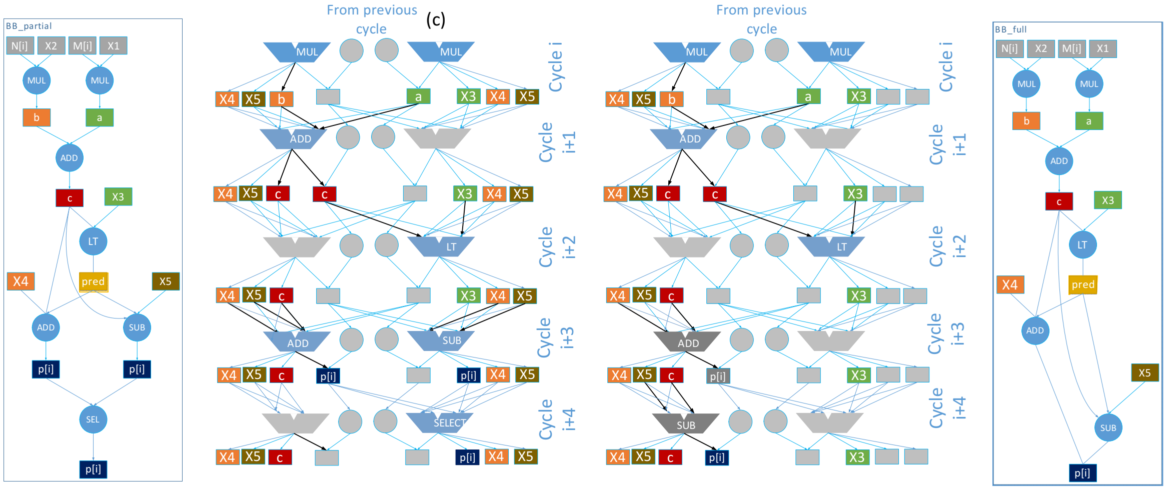
Figure 3.4 (a) shows the partial predication transformation of the CDFG presented in Figure 3.1(c), and mapping of the transformed DFG onto the CGRA (Figure 3.1(a)) in Figure 3.4(b). To map a conditional that has n operations on each path, the number of operations for partial predication transformation is, in the worst-case, $3n$. This is because all the operations from both the paths must be mapped (2n), as well as the select operations (n), assuming the worst-case produces outputs in each operation, which are used outside of the conditionals.

Full predication

Full predication executes the two paths sequentially. Unlike partial predication the full predication does not need the *select* operation, instead, the operations that update the same variable are mapped to the same PE but in different cycles. Since only one of the operations will be executed at runtime (and the other will be squashed), the correct value of the output is present in the register file of that PE by the end of that iteration. If the paths have different variables to update, then they can be mapped in different PEs. This is done so that after

---

[1] A predicated network in hardware is necessary to support the execution



(a) (b) (c) (d)

Fig. 3.4 (a) Transformed DFG of the conditional presented in figure 3.1 by partial predication; (b) Mapping of the DFG onto the CGRA in figure 3.1(a); (d) Transformed DFG of the conditional presented in figure 3.1 by full predication; (c) Mapping of the DFG onto the CGRA in figure 3.1(a)

executing an ITE, for each variable there is a unique PE, that has its value and therefore no select operation is required.

Figure 3.4 (a) shows the full predication transformation of the CDFG presented in Figure 3.1(c), and mapping of the transformed DFG onto the CGRA (Figure 3.1(a)) in Figure 3.4(b). Since both the PEs update the same variable in this case, they are mapped onto the same PE, and the output is validated at the end of the ITE execution. A conditional that possesses n operations in each path, full predication DFG transformation in the worst-case costs $2n$. Since the execution of one path gets squashed, there is performance penalty in this technique.

Others

Dual issue scheme [46] targets energy efficiency by issuing two instructions to a PE simultaneously, one from the if-path, another from the else-path. In this mechanism, the latency remains similar to that of the partial predication with improved energy efficiency. However, this approach is too restrictive, as far as imbalanced and nested conditionals are concerned. To map nested, imbalanced conditionals and single loop onto CGRA, the triggered long instruction set architecture (TLIA) is presented in [68]. This approach merges all the conditionals present in kernels into triggered instructions, and creates instruction pool for each triggered instruction. As the depth of the nested conditionals increases the performance of this approach decreases. As far as the loop nests are concerned, the TLIA approach reaches bottleneck to accommodate the large set of triggered instructions into the limited set of PEs.

In this chapter, we address this problem by introducing a *register allocation* mapping approach where both the true and false path can reuse the resources preventing the waste of additional resource and power. This allows to map both loops and conditionals of any depth. In our case, the only limitation in the mapping of kernels onto the CGRA is given by the size of instruction memory of the PEs, and not by the structure of the application (i.e. number of loops, and branches). Also, one can increase the size of code segment to be executed in the CGRA as much as possible, minimizing the control and synchronization overheads with the core, which is not negligible in the other approaches.

Traditional CGRAs manage to execute only the *innermost loop*, since they lack the support for branches. The traditional software pipelining is an excellent choice for accelerating the innermost loop only. Compilation flow proposed in [81], [74], [45], [43] [35] [14] use modulo scheduling [87] for innermost loop pipelining. For the outer loops, the CPU or the host initiates each iteration. As, the loop nests increases, the communication overhead goes high both in terms of performance and power penalty. However, software pipelining faces several limitations such as, in-loop function calls[2], multiple exits inside the loop. Loops with uncertain exits (example loop in the following code to compute greatest common divisor (Listing 3.2)) are not qualified for software-pipelining either.

```
//greatest common divisor (gcd)

void gcd (n1, n2)
{
    while(n1!=n2)
    {
        if(n1 > n2)
            n1 -= n2;
        else
            n2 -= n1;
    }
    res = n1;
}
```

Listing 3.2 Loop with uncertain exits

On the other hand, loop unrolling has its own limits for increasing code size immensely, preventing optimizing all the loop levels of a nested loop structure. Hence, for a flexible application acceleration, the need to support branches in CGRA accelerators is unavoidable.

---

[2] this can be sorted out using intrinsics

The compilation flow discussed in the latter sections uses partial unrolling of the innermost loop.

Table 3.1 presents a comprehensive comparison between several techniques to manage control flow in the kernels. The table clearly shows that the register allocation approach can deal with any kind of conditionals and loops.

Table 3.1 Comparison between different approaches to manage control flow in CGRA

Techniques	**Conditionals**		**Loops**	
	Balanced	**Imbalanced**	**Single**	**Nested**
Partial predication [13]	√	√	×	×
Full predication [4]	√	√	×	×
State based full predication [47]	√	√	×	×
Dual issue single execution [46]	√	×	×	×
TLIA [68]	√	√	√	×
Software pipelining [74]	×	×	√	×
Loop unrolling [62]	×	×	√	NA
Register allocation [24]	√	√	√	√

Next, in this chapter, we discuss the problem for supporting branches in CGRAs, and formulate a *register allocation* approach for supporting control flow efficiently. The compilation flow, described later in this chapter, is developed using this approach. Results at the final part of this chapter demonstrates the flexibility and efficiency of the compilation approach both in terms of performance and energy gain.

3.2 Compilation flow

Figure 3.5 shows a schematic representation of the compilation flow for mapping CDFGs onto the PEA. A CDFG mapping is a set of DFG mappings that are compatible with each other. To be compatible, the DFGs must access the data that remain in the PEs (see symbol variables (see definition 3.2.1)) in the same location. This is ensured by the register allocation approach.

First, the flow orders the basic blocks and for each basic block it finds a set of DFG mappings that are compatible with the DFG already mapped by settings the constraints. When no solution for scheduling and binding is found, the flow tries to transform the application

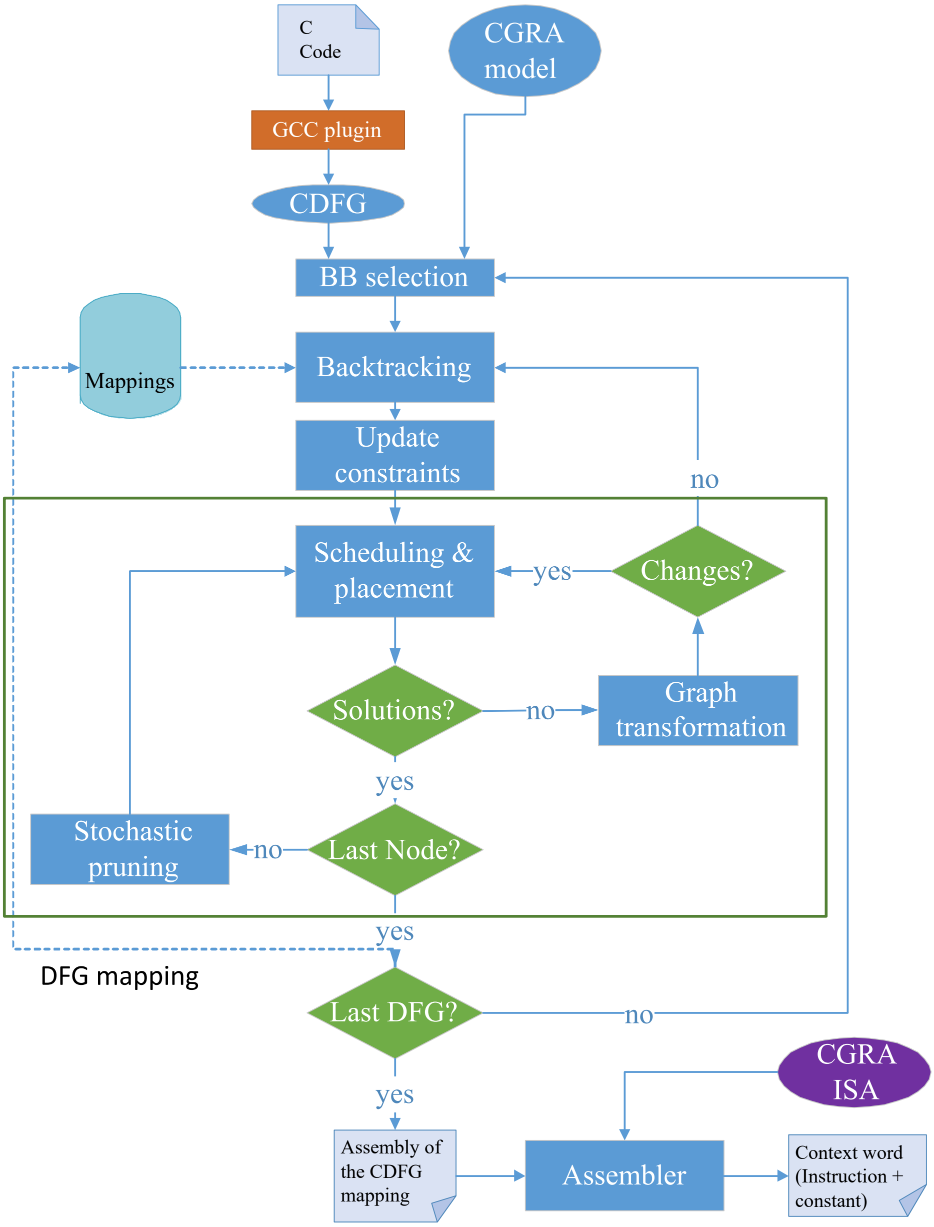

C
Code
CGRA
model
GCC plugin
CDFG
BB selection
Mappings
Backtracking
Update
constraints
no
Scheduling &
placement
yes
Changes?
Solutions?
no
Graph
transformation
yes
Stochastic
pruning
no
Last Node?
yes
DFG mapping
Last DFG?
no
CGRA
ISA
yes
Assembly of
the CDFG
mapping
Assembler
Context word
(Instruction +
constant)

Fig. 3.5 Compilation flow

graph to ease the mapping. When no transformation can be applied, it means that a mapping for the current basic block cannot be found given the constraints of the selected mappings of the other basic blocks. A backtrack mechanism is used to select another consistent set of already mapped DFGs to map the current DFG. The set of valid mappings found for the current basic block is saved into a mapping bank. To map the basic blocks, we rely on the highly scalable and efficient mapping approach for DFGs described in [23]. The compilation flow proposed here, extends the DFG mapping to accommodate the register allocation approach to map a full CDFG onto the PE array. As presented in Figure 3.5, the full compilation flow is composed of six interdependent stages: BB selection, backtracking, update constraints, scheduling and placement, graph transformation and a stochastic pruning. First, we discuss the steps involving the mapping of DFGs, then, we introduce the problems while mapping the control flow graph and discuss the solutions.

3.2.1 DFG mapping

As shown in Figure 3.5, mapping of DFGs involves three steps, scheduling and placement, graph transformation, and stochastic pruning.

Scheduling and placement

The scheduling step uses a backward traversal [83] list scheduling algorithm to schedule nodes of the DFG. It relies on a heuristic in which the schedulable operations are listed by priority order. In backward traversal, a node is schedulable if and only if all its children are already scheduled (e.g. node 2, in Fig. 3.6(b), is not schedulable since node 3 is not yet scheduled. So, it must be routed to keep data dependency resulting in Fig. 3.6(c)). The priority of nodes depends on their mobility and number of successors (fan-outs). It is possible to process memorization nodes and conventional nodes differently. When several nodes have the same mobility, their respective number of successors is used as a second priority criterion. The higher the number of successors, the higher the priority. Indeed, a node with a higher number of successors is more difficult to map due to routing constraint coming from the limited amount of connections between tiles. Thus, scheduling these nodes at first usually allows for reducing the application's latency (e.g. node 2 in Fig. 3.6(d) has a higher priority than node 1).

As soon as nodes are prioritized and ordered, our approach tries to find a binding solution. The first node is then selected from the ordered list and the algorithm searches for a binding solution. If no binding solution exists, the graph is transformed (see Section 3.2.1). The proposed placement uses an incremental version of Levi’s algorithm [63], i.e. fully exhaustive

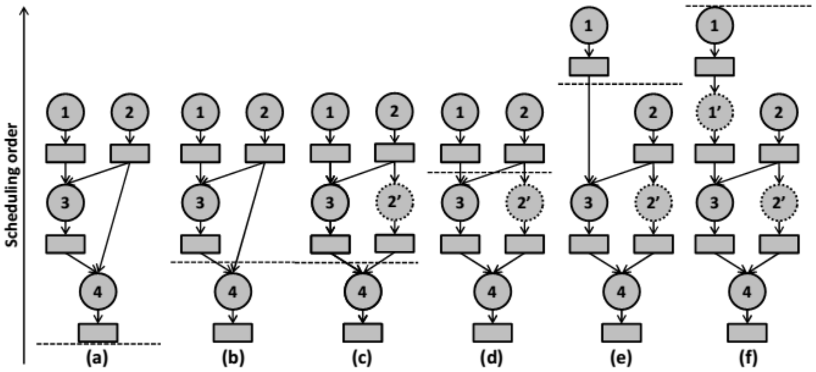


Fig. 3.6 Example of scheduled and transformed DFG on a CGRA with one PE. (a) Initial DFG, (b) after scheduling node 4, (c) after adding node 2', (d) after scheduling node 3 and 2', (e) after scheduling node 2, (f) Scheduled DFG after routing and scheduling node 1. Horizontal line shows the limit between scheduled and non scheduled nodes. Memorization nodes are doted circles.

search of the whole DFG. The algorithm we propose, adds the newly scheduled operation node and its associated data node to the sub-graph composed of already scheduled and bound nodes. Only the previous set of solutions that have been kept are used to find every possibility to add this couple of nodes without considering the non-yet scheduled nodes. If no solution is found, there is absolutely no possibility to bind this couple in all the previous partial solutions because Levi's algorithm provides a complete exploration of the available solution space.

Introducing stochasticity in the scheduling: The scheduling discussed above is a list ordering using a backward traversal. This heuristic approach proposes to schedule the nodes according to a priority function. The priority is derived depending on two criteria: i) the mobility of the nodes, ii) the number of outgoing arcs for the nodes having the same mobility. Despite these two types of criteria, it is possible that several nodes have the same priority (typically, those with same mobility and only one outgoing arc). Nodes with similar mobility and number of successors are ordered randomly. *Stochasticity* is introduced in the scheduling process to get better coverage of the underlying micro-architecture. The ability of the random selection of the similar priority nodes to better architectural exploration, is examined at the end of this section.

Graph transformation

DFG is transformed dynamically when no binding solution is found. Following are the two graph transformations (Figure 3.7) used in our compilation flow.

1. *Operation splitting:* duplicates an operation node by keeping its same inputs and distributing output edges to reduce the number of successors of the original operation node (see Fig. 3.7(b)).
2. *Memorization routing:* adds a memorization node and its associated data node to delay one operation and to keep data dependencies (see Fig. 3.7(c)).

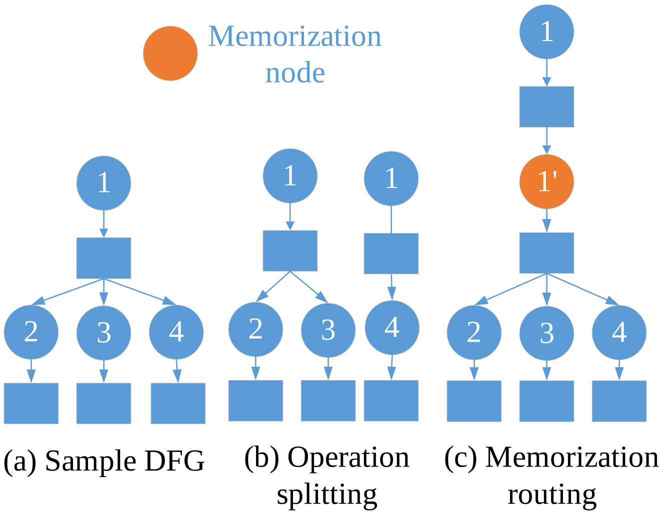


Fig. 3.7 Graph transformation

Stochastic pruning

The exhaustive enumeration of Levi's algorithm usually leads to a very large number (up to tens of thousands) of partial mappings (depending on the data dependencies and the architectural constraints) which prevents its use with large DFG and/or complex CGRA. In [83], the idea to reduce this number was to remove redundant partial mappings. A partial mapping is redundant when it uses the same operators to make the same operations as another partial mapping at the current scheduling cycle. This step allows for keeping only all the different partial solutions and preserving an exhaustive search. However, this pruning technique does not scale well. The problem is so complex that it is difficult to define a smart and efficient pruning function. To keep both computation time and memory usage to a reasonable level in the mapping tool, we propose to use a stochastic selection instead of removing redundant partial mappings. This pruning step is made after the binding step and before scheduling the next node. Let the result of the binding be a list *nbMappings* (*nbM*) of partial solutions. The *stochastic pruning* step selects *nbCurrentMappings* (*nbCM*) number of partial solutions from *nbMappings*.

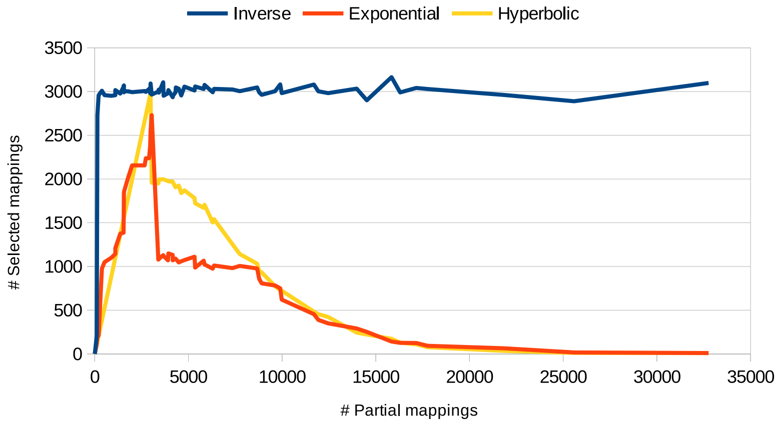


Fig. 3.8 Performance of threshold functions (we choose inverse function for its stable selection capability)

For each partial mapping, a random number between 0 and 1 is generated and compared to a threshold. This threshold must be chosen carefully: it should be low enough to scale up and high enough to allow keeping enough partial solution among which at least one solution can lead to a complete mapping. Thus, the threshold should adapt itself to *nbMappings*. For that purpose, *nbMappings* is normalized by a reference number λ, set by the user. This number is used by the threshold function. Many functions can be considered (e.g. exponential, invert, hyperbolic etc.). To select an optimal threshold function we present a performance graph (Fig. 3.8) which presents the average number of selected partial mappings (*nbCurrentMappings*) for ten runs with average number of original partial mappings (*nbMappings*) for λ value 3000 (the same trend is experienced with several other values of λ).

We experience exponential decay in selected number of mappings for exponential and hyperbolic function as opposed to inverse function. Hence the inverse function has been chosen as the threshold function (see Eq. 3.1) in our approach.

$$Threshold(nbM,\lambda) = \begin{cases} (\lambda/nbM) & if\, nbM > \lambda \\ 1 & if\, nbM \leq \lambda \end{cases} \tag{3.1}$$

After choosing the right threshold function it becomes very important to have control over the number of selected partial mappings as this leads the approach to find a valid solution. We propose to introduce bounds as control mechanisms: *LB (Lower Bound)* and *UB (Upper Bound)*. We propose two variants based on bounds.

LB & UB: This variant sets an upper bound and lower bound on *nbCurrentMappings* as presented in equation 3.2 and 3.3. In this method also, a random number is generated between 0 and 1, which is compared to the threshold value. If the random number is less than or equal to the threshold or the lower bound is not satisfied, then it selects the partial solution from *nbMappings* and stores into *nbCurrentMappings* otherwise the solution is discarded. If *nbCurrentMappings* exceeds the upper bound, then it stops selection of partial mappings.

$$\max nbCM = \lceil nbM/3 \rceil \tag{3.2}$$

$$\min nbCM = \begin{cases} |nbM/\lambda| & \text{if } nbM > \lambda \\ \lceil nbM/3 \rceil & \text{if } nbM \leq \lambda \end{cases} \tag{3.3}$$

LB only: This variant generates a random number between 0 and 1 which is compared to the *threshold*. If the random number is less than or equal to the threshold then it selects the partial solution from *nbMappings* and stores into *nbCurrentMappings* otherwise the solution is discarded. The solution space *nbMappings* is traversed again and again until *nbCurrentMappings* reaches the minimum bound as presented in the equation 3.4.

$$\min nbCM = \lceil nbM/\lambda \rceil \tag{3.4}$$

Efficiency of stochasticity in mapping: To demonstrate the efficiency of stochastic based approach in the mapping flow, we perform experiments involving the kernels presented in Table 3.1 in chapter 3. Since latency performance and compilation time are the most critical parameters that are affected by the introduction of stochasticity in the mapping, we analyse the effect of different combinations of stochastic behaviour and a non-stochastic approach, on both latency and compilation time. For the different combinations of stochastic behaviour in the pruning stage, we consider (a) mapping with Stochastic pruning with no bounds or *SNoB*, (b) mapping with stochastic pruning with lower and upper bounds (LB & UB) or *SLUB*, (c) mapping with stochastic pruning with lower only bound (LB) or *SLoB*. For a non-stochastic based approach in pruning we select a mapping approach based on redundant elimination-based pruning, proposed in [83], which is referred to as *RED* in the comparisons. Since the mappings are based on different stochastic solutions, in the experiments we take the best outcome out of ten runs to ensure the best performance of the corresponding method.

- *Latency and computation time:* In Figure 3.9, we compare the latency obtained by the different mapping approaches normalized to the ASAP length of the corresponding DFGs. As the trends are the same for different sizes of CGRA and RF, we have presented results for only 3x3 CGRA with RF 24. Fig. 3.10 presents compilation time comparison. To realize the gain in compilation time using stochastic based pruning over the state of the art pruning using redundant elimination *RED*, we have normalized the compilation time over *RED*.

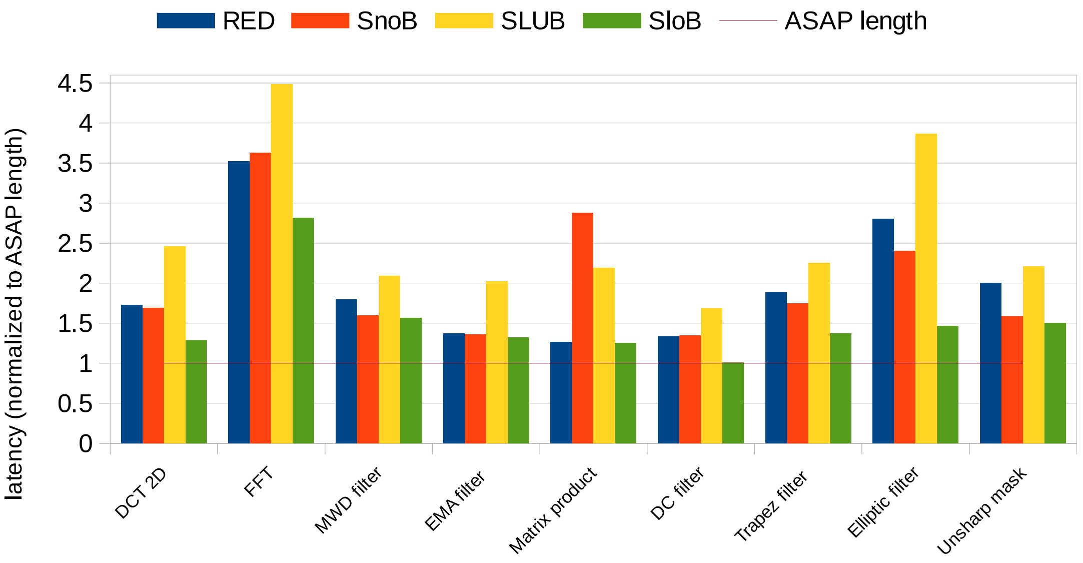


Fig. 3.9 Mapping latency comparison for 3x3 CGRA

Fig. 3.9 shows the ability of different methods to find mappings with best latency. The latency value closer to the ASAP line refers to the capability of finding better mappings. The latency comparison depicts that *SLoB* generates the best of mappings whereas *SLUB* produces the worst latencies. The compilation time comparison in Fig. 3.10 shows that *SLUB* and *SLoB* achieve best scaling. Comparing both the performance metrics, the *SLoB* is the clear winner.

- *Architectural coverage:* After comparing the performance metrics, we analyse the ability of the stochasticity in mapping, to explore the underlying micro-architecture. Since better architectural coverage ensures better resource utilization, we consider the best performed candidate, the *SLoB* from the above set of experiments. As discussed in the previous section, we introduce *SLoBS* which integrates ***stochastic scheduling*** in *SLoB*. For the experiment, we consider FFT kernel, as it possesses the highest number of parallelism (see Table 3.1 in chapter 3).

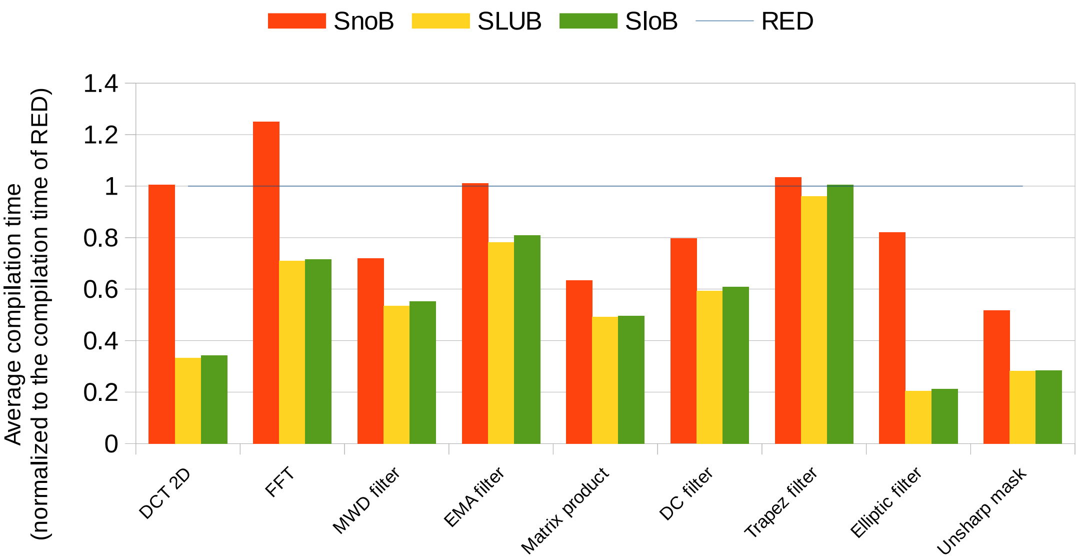


Fig. 3.10 Compilation time comparison for 3x3 CGRA

Fig. 3.11 exhibits the architectural coverage for different mapping approaches. In this figure, we present FFT benchmark running for different CGRA configurations using three different methods, *RED, SLoB and SLoBS*.

Since the trend realized in this figure is similar for other benchmarks, results for only one kernel is presented for clarity and better understanding. The CGRA configurations are presented using dimension and the RF size (i.e. 4×4 RF 16, means the configuration is for a 4×4 CGRA where each PE consists of a RF of size 16). Each point in the Fig. 3.11 corresponds to the outcome of a single run by a method on the corresponding CGRA configuration.

The x axis of the graph represents latency normalized to ASAP length and the y axis represents the number of transformed nodes normalized to the number of operation nodes in the original graph. In other words, each point in the graph is basically the outcome latency and number of transformed nodes of each run by a certain method. The points corresponding to the method *RED* and *SLoB* show that they find similar latencies with almost similar number of transformations. The wide range of latencies and transformations in method *SLoBS* prove that this method can explore the solution space better. Not surprisingly, the method *SLoBS* finds the best latency with least number of transformations.

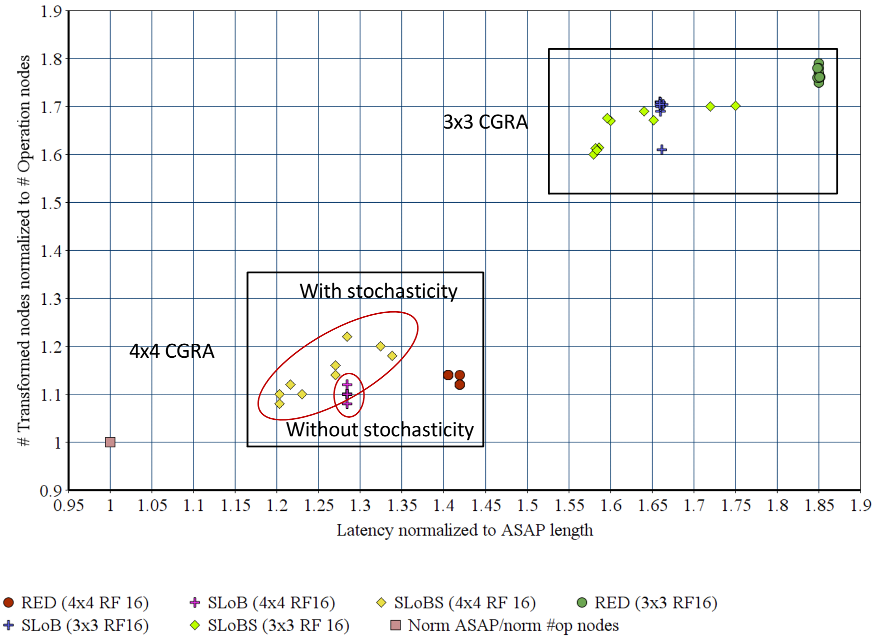


Fig. 3.11 Architectural coverage between methods

3.2.2 CDFG mapping

First, we formulate the problem of CDFG mapping and propose a *register allocation* based solution accordingly. Subsequently, we discuss the steps involving the mapping of CDFG.

Definition and problem formulation

Data in an application is separated into two categories.

1. The standard input and output data (mostly the array inputs and outputs) are mapped as memory operands. The inputs and outputs are allotted by load-store operations. In our sample program in Figure 3.2, m, n are the input arrays and p are the output array, which are managed by load and store operations.

2. The internal variables of a program are mapped onto the registers of the processing elements, and managed by the register allocation based approach [24].

Following, we introduce several definitions concerning register allocation approach:

Definition 3.2.1. *Symbol Variables and location constraints:* In compilation, the recurring variables (repeatedly written and read) are managed in local register files of the PEs to avoid multiple access of local memory. The recurring variables which have occurrences in multiple basic blocks need special attention since the integrity of these variables must be kept intact throughout the mapping process for different basic blocks. These variables are defined as *Symbol variables*. The register locations for the symbol variables are referred to as *location constraints*. For instance, variable c in the CDFG (Fig. 3.2) is written in BB_3, and read in BB_4, BB_5 and BB_6. In mapping all these basic blocks the register location for c must be same. Similarly, $X1$, $X2$, $X3$, $X4$, $X5$, i, a and b must be location constrained. The locations for such *symbol variables* are denoted with an overline, as $\overline{variable\_name}$.

Depending on the order of the basic blocks mapped (i.e. traversing the CDFG), some location constrains may be reused in the mapping process or may be kept reserved for later use. These two types of location constraints are discussed in the following.

Definition 3.2.2. *Target Location Constraints (TLC):* We consider a scenario *scenario_1*, where BB_6 is mapped first, BB_3 is mapped next and so on. While mapping BB_6, variables c and $X5$ are placed at $\overline{c}$ and $\overline{X5}$. While mapping BB_3, $\overline{c}$ and $\overline{X5}$ which are already mapped in BB_6, must be considered because $\overline{c}$ will be used to map c in BB_3. In other words, the placement of the variables in the registers must be respected. Also, $\overline{a}$, $\overline{b}$, $\overline{X1}$ and $\overline{X2}$ must not reuse $\overline{X5}$. Otherwise, $X5$ will have wrong value when executing BB_6. Let's consider *scenario_2* with another order of basic blocks mapped, like first BB_3 and then BB_6 and so on. In this order of mapping, it is necessary to pass $\overline{c}$ and $\overline{X5}$ from BB_3 to BB_6 mapping. To keep c and $X5$ alive in BB_6 both $\overline{c}$ and $\overline{X5}$ must be used in mapping of BB_6. The placement or binding information which are passed from the mapping of one basic block to the mapping of the other basic block is referred to as *constraint* (e.g. *scenario_1*: $\overline{c}$ and $\overline{X5}$ passed from BB_6 to BB_3). The location constraints related to the data that are used within a basic block mapping phase (e.g. *scenario_1*: $\overline{c}$ in BB_3 mapping) are referred to as *target location constraints* (TLC).

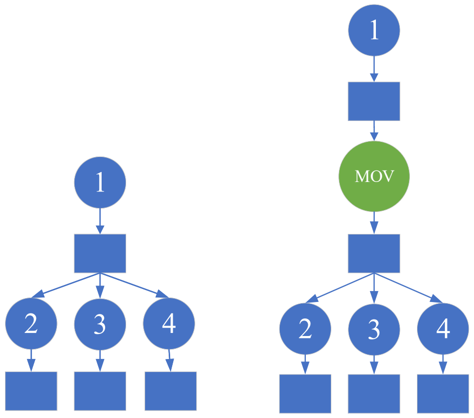
Definition 3.2.3. *Reserved Location Constraints (RLC):* As we have seen in the previous examples, some of the location constraints must be reserved in the mapping of basic blocks for the sake of data integrity. To keep the symbol variables alive, it is necessary to exclude the memory elements from placement. Accordingly, these resources will not override while mapping the basic block (e.g. *scenario_1*: $\overline{X5}$ in BB_3 mapping). These are referred to as *reserved location constraints* (RLC).

If the number of RLC and TLC is high, mapping becomes complex. As TLC forces to use resources, and RLC forces to exclude resources from placement. Hence, the primary goal for our compiler is to minimize the number of TLCs and RLCs by choosing an efficient traversal of the CDFG.

Register allocation approach: The basic solution to deal with the symbol variables is to introduce memory operations. The symbol variables are stored in the memory where they are generated and are loaded from the memory when used as operands. This basic solution is referred to as *systematic load-store based approach*. This method is presented in the Figure 3.1(d). For the symbol variable c in the CDFG shown in Figure 3.1(c), it stores variable c in the memory in BB_3, and loads in BB_4, BB_5 and BB_6. Figure 3.1 refers to the mapping of the transformed CDFG in this approach. This basic solution reduces the complexity of the mapping as there are no constraints in the basic block mapping. On the other hand, it requires a huge memory bandwidth, significantly reducing the energy efficiency of the system. As an alternative, we propose *register allocation approach*, where the symbol variables are stored in the register files when they are written and retrieved from the registers when used as operands. While doing so, the effects of the constraints in mapping are unavoidable. RLC restrict the use of some resources, and TLC force to reuse some resources. If there is only a single TLC in a basic block mapping, it becomes easier to start mapping from the known place. However, several TLC and RLC complicates the mapping. Forced and blocked placements by these constraints induce extra routing effort (dynamically transforming the graph in compilation).

Impact of the constraints on DFG mapping: Since location constraints in the register allocation approach forces to use and block some of the locations while mapping the variables, we have tailored the placement algorithm in DFG mapping. The modified binding approach uses a database of the RLC and TLC to find placements of the current data nodes. If no solution is found due to the constraints the DFG is dynamically transformed emulating additional routing of the targeted data node. We introduce Assignment routing (Figure 3.12), which adds an assignment node (*mov* operation node) to increase the physical distance between the source and sink of symbol variables by one. Due to TLC or RLC, when the physical distance between the source and sink of the symbol variable becomes more than one, the compiler dynamically adds one *mov* operation node to the DFG.

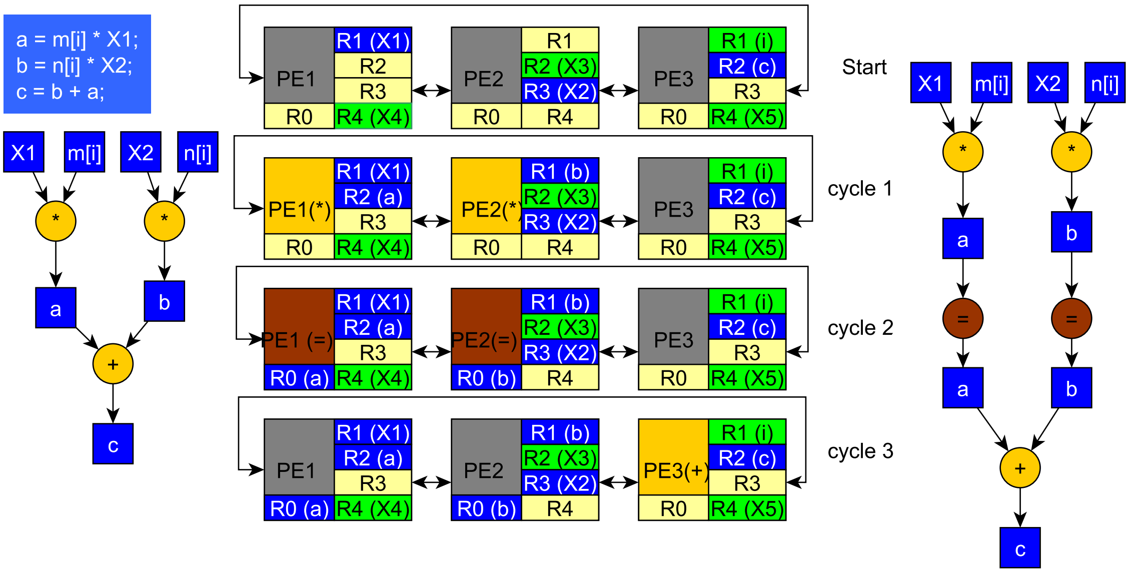
To illustrate the excess data routing due to RLC and TLC, we consider a scenario where BB_1 and BB_4 in Figure 3.2 are already mapped (variables $X1$, $X2$, $X3$, $X4$, $X5$, c, i already mapped). The mapping of BB_3 must be done considering the TLC $\overline{c}$, $\overline{X1}$, $\overline{X2}$ and RLC $\overline{X5}$, $\overline{X3}$, $\overline{X4}$, $\bar{i}$. The variables a, and b in BB_3 must be mapped satisfying all these constraints. Consequently, additional data move might be necessary. A graphical view of this circumstance is presented in Fig. 3.13, where BB_3 is being mapped onto a 3×1 PEA with 4 registers in the RFs of each PE (R0 is the output register). In this PEA, we assume that the register files are local to the PEs.


1
2
3
4
1
MOV
2
3
4

(a) Sample DFG (b) Assignment Routing

Fig. 3.12 Assignment routing graph transformation

$\overline{a}$ and $\overline{b}$ will be mapped in the respective PEs where $\overline{X1}$ and $\overline{X2}$ are allocated. Extra routing effort may be necessary to bring a and b to the PE where $\overline{c}$ is allocated. Hence, the graph must be transformed dynamically, adding extra *mov* operation, when such situation arises. The mapping can be done because the addition operation must generate c in $\overline{c}$ which is a location in the register file (RF) of the corresponding PE.


a = m[i] * X1;
b = n[i] * X2;
c = b + a;
X1
m[i]
X2
n[i]
*
*
a
b
+
c
PE1
R1 (X1)
R2
R3
R0
R4 (X4)
PE2
R1
R2 (X3)
R3 (X2)
R0
R4
PE3
R1 (i)
R2 (c)
R3
R0
R4 (X5)
Start
PE1(*)
R2 (a)
PE2(*)
R1 (b)
cycle 1
PE1 (=)
R0 (a)
PE2(=)
R0 (b)
cycle 2
PE3(+)
cycle 3
=
=

(a) (b) (c)

Fig. 3.13 (a) DFG BB_3. (b) mapping of BB_3 onto a 3x1 PEA starting with TLCs $\overline{c}$, $\overline{X1}$ and $\overline{X2}$ and RLCs $\overline{X5}$, $\overline{X3}$, $\overline{X4}$ and $\overline{i}$. (c) transformed BB_3 after mapping.

As we can see in Figure 3.13 (b), the execution starts with TLC $\bar{c}$ $(PE3-R2)$, $\overline{X1}$ $(PE1-R1)$ and $\overline{X2}$ $(PE2-R3)$, and the RLC $\overline{X3}$ $(PE2-R2)$, $\overline{X4}$ $(PE1-R4)$, $\overline{X5}$ $(PE3-R4)$ and $\bar{i}$ $(PE3-R1)$. Due to the constraints, the original BB_3 was transformed to the basic block presented in Figure 3.13 (c), and mapping was settled in 3 cycles. The TLCs force to map a, b in $PE1$ and $PE2$ in cycle 1. Let's assume they are mapped in $PE1-R2$ and $PE2-R1$ respectively. In cycle 2, a and b cannot be accessed to produce c in $PE3-R2$. Hence, graph transformation is necessary to route a, b from the register files to the output registers, which is done in cycle 2. In cycle 3, c is generated in $\bar{c}$ which is $(PE3-R2)$. Hence, the mapping of the operation attached to c in this case, experiences longer schedule due to the several TLC and RLC. The increased number of the constraints during the basic block mapping affects the complexity and the quality of the mapping. Hence, it is necessary to wisely select the basic blocks to reduce the impact of the constraints on the mapping. In the next section, we present a suitable traversal of CDFG to minimize the number of RLCs and TLCs.

Following, we discuss the compilation flow steps implementing the register allocation approach.

Basic block selection

Once all the nodes of the BB have been scheduled and bound, the compiler selects one partial mapping among the several mappings generated and selects the next basic block to be mapped. As discussed previously, it is necessary to maintain data integrity over several basic block mappings. The data mapping problem for CDFG mapping is now described before going into the detailed basic block selection step.

As the selection of the basic blocks during the mapping is important, we compare the number of TLC and RLC for several CDFG traversal strategies in this section. Table 3.2 presents the comparison between the number of different constraints in the forward and backward CDFG traversal for Breadth First Search (BFS) and Depth First Search (DFS) strategies. As the trend is similar for other kernels we present the results for sobel and separable 2D filter only. The numbers show that DFS strategy generates a lower number of RLC than the BFS in both forward and backward traversal. The number of RLC for sobel filter is much higher in BFS due to several sequential loops present in the kernel. The numbers of TLC are similar in both the strategies for different traversal mechanisms. Also, for the different search strategies forward and backward traversal perform similarly. The DFS strategy is thus used.

Table 3.2 Comparison of RLC and TLC numbers between different CDFG traversal

Kernels	**Forward Traversal**				**Backward Traversal**			
	BFS		**DFS**		**BFS**		**DFS**	
	# RLC	**# TLC**	**# RLC**	**# TLC**	**# RLC**	**# TLC**	**# RLC**	**# TLC**
Sep 2D Filter	22	35	17	35	22	35	17	35
sobel Filter	64	85	35	85	69	85	35	85

Backtracking

For a basic block to be mapped except the first one, this stage selects the first map out of several mappings generated for the last basic block mapped. The selected map updates the constraints for the current basic block mapping. If one basic block does not find a mapping due to the constraints, this stage selects the second map from previous basic block to update the constraints and restart mapping of the new basic block. The process continues up to the first basic block mapped until a valid mapping is found for the current basic block.

Update Constraints

In this stage, the compiler creates and updates a constraint database. This database is used in the placement algorithm, to place the data nodes and corresponding operation nodes according to the TLC and RLC. In the current basic block mapping variables are not placed in RLCs, and TLCs are used to map the symbol variables. When mapping a current basic block, new variables cannot be placed in RLCs, while TLCs are used to map the symbol variables. If the symbol variable in the current basic block mapping is not present in the constraint database, then the variable is mapped using available resources, and the respective placement is used to update the constraint database prior to next basic block mapping.

Once all the basic blocks are mapped the compiler generates the assembly file containing a single map for the whole CDFG.

3.2.3 Assembler

Assembler holds the key to differentiate from the PEA model used in the compiler and the actual hardware implementation. The assembler takes the ASCII text assembly generated by the compiler and the instruction set architecture (ISA) and produces machine code, which can then be used to configure the PEs in the hardware. The ISA provides the added hardware information to the PEA model used in the compiler. As an example, the PEs in the IPA use an added constant register file (CRF) for storing the constants. The introduction of the CRF in the PEA model minimizes the instruction length by storing the immediates of the instruction into the internal registers, giving a low power solution. That is how the assembler separates the model used in the compiler from the actual implementation of the hardware. One can define their own PEA model and derive an architecture from that for actual implementation. Thus, the compiler can be used for a wide range of PEA variations.

3.3 Conclusion

In this chapter, we presented a compilation flow targeting the mapping of both control and data flow portions of kernels onto the IPA. The proposed approach maps a complete CDFG with least number of memory operations. A *Register allocation* approach was introduced for maintaining data locality throughout the CDFG mapping. We also showed the effect of the constraints raised due to the register allocation approach on different traversal of the CDFG. For mapping the basic blocks in the CDFG, the proposed approach leverages on simultaneous scheduling and binding steps respectively based on a heuristic and an exact method. Stochastic pruning was introduced to reduce the impact of the exact binding approach. The formal graph model of the basic blocks, obtained after compilation, is backward traversed and dynamically transformed to allow for a better exploration of the design space. In the next chapter, we present the efficiency of the compilation flow.

Chapter 4

IPA performance evaluation

In this chapter, first, we analyse the implementation of the IPA, providing performance, area, and energy consumption on several signal processing kernels. We perform an architectural exploration to find the optimal configuration in terms of number of load-store units and number of TCDM banks for a IPA with 4×4 PE array. Performance, area and energy efficiency are compared with that of the or1k CPU [59].

Second, we carry out experiments to show the efficiency of the register allocation approach compared to the state of the art predication techniques, considering a wide range of control dominated kernels. The proposed mapping flow has been fully automated through a software tool implemented by using Java and Eclipse Modeling Framework (EMF). GCC 4.8 is used to generate CDFGs from applications described in C language. Finally, we present the efficiency of the compilation flow, executing a smart visual trigger application enriched with data-flow and control-flow intensive kernels on the IPA, compared with the state of the art architectures.

4.1 Implementation of the IPA

This section presents the implementation results of the IPA using STMicroelectronics 28nm UTBB FD-SOI technology libraries. For area reference we consider low power or1k [59] CPU. Both the designs were synthesized with Synopsys design compiler 2014.09-SP4. The IPA consists of a 4×4 array with 16 PEs, each one consisting 64×20-bit instruction register file, a 8×32-bit regular register file and 16×32-bit constant register file, as shown in Table 4.1. For area comparison, the CPU includes 32kB [1]of data memory, 4kB of instruction memory, and 1 kB of instruction cache, which is equivalent to the design parameters of the IPA.

---

[1] The size is considered both in size and power

For the memory access optimization, we compare the performance and energy efficiency of different configurations in the IPA with the CPU. The different configurations of the IPA are the variation of the number of LSUs present in the PEA and the number of TCDM banks present in the data memory. Table 4.2 presents the code-size (instructions and constants) and maximum depth of the loops present in the kernels used for the following experiments. Thanks to the simpler architecture and tiny processing elements, at the target operating voltage of 0.6V, the IPA runs at 100 MHz while or1k can only reach 45MHz in the same operating point. Synopsys PrimePower 2013.12-SP3 was used for timing and power analysis at the supply of 0.6V, 25°C temperature, in typical process conditions. The cycle information was achieved simulating the RTL with Mentor Questa Sim-64 10.5c.

Table 4.1 Specifications of memories used in TCDM and each PE of the IPA

Name	Type	Size
Global context memory	SRAM	8KB
TCDM	SRAM	32KB
Instruction Register File (IRF)	Registers	0.16KB
Regular register file (RRF)	Registers	0.032KB
Constant register fie(CRF)	Registers	0.128KB

4.1.1 Area Results

Figure 4.1 shows the area of the whole array and memory with different numbers of TCDM banks, where the total amount of memory is kept constant at 32kB. As the area of LSUs is negligible if compared to the overall system area, we show the area results for the worst-case scenario with maximum number of LSUs present in the PE array (i.e. 16). As shown in Figure 4.1, in the minimal configuration with 4 TCDM banks, the IPA area is dominated by the array of PEs (60%) and by the local data storage (35%), while the remaining 5% is consumed by the interconnect. Increasing the number of TCDM banks imposes a significant area overhead on the size of the interconnect. Also, the area of the TCDM increases as well due to the higher area/bit of small SRAM cuts necessary to implement 32kB of memory with several banks. Hence, it is fundamental to properly balance the number of LSUs and TCDM banks with the bandwidth requirements of applications.

4.1.2 Memory Access Optimization

This section provides an extensive comparison with respect to the CPU computational model and an evaluation of the performance of the IPA while varying the number of LSUs and

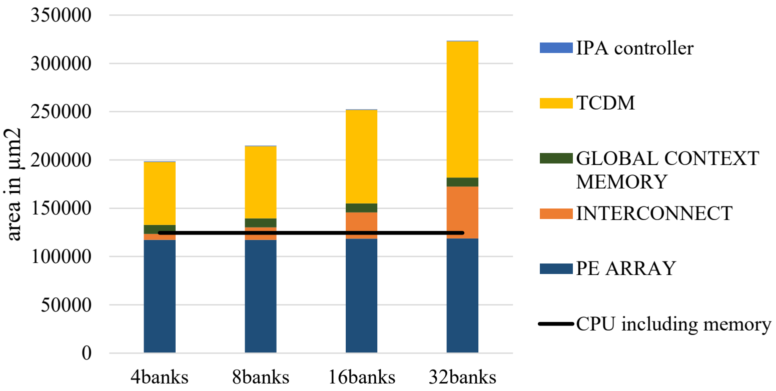


Fig. 4.1 Synthesized area of IPA for different number of TCDM banks

TCDM banks, a critical parameter for data-hungry accelerators. To carry out the exploration, we selected seven compute intensive signal processing kernels featuring a high bandwidth towards the TCDM. Table 4.2 presents the code-size (instructions and constants) of all the kernels used in the following experiments. The cost of the IRF is considered both in size and power.

Table 4.2 Code size and the maximum depth of loop nests for the different kernels in the IPA

Kernels	**FIR**	**MatMul**	**Conv**	**Sep filter**	**Non-Sep filter**	**FFT**	**DC filter**	**cordic**	**sobel**	**gcd**	**sad**	**deblock**	**manhdist**
Code size (KB)	0.568	0.704	0.704	0.720	0.784	0.696	1.16	0.496	0.336	1.448	0.600	2.016	0.624
Max depth loop nests	2	3	3	3	4	2	2	1	1	1	2	3	2

Performance

Generally speaking, the IPA performs well when significant parallelism can be extracted from a kernel. This concept is well shown in Figure 4.2, which compares the performance of the IPA with that of the or1k processor on a matrix multiplication when growing the size of the matrices from 2×2 to 32×32. It is possible to note that the increase of the kernel size increases the average utilization of the PEs as well, which in turn helps to enhance performance. It also demonstrates that the initial configuration time, which is dominant for small kernel size is well amortized for larger kernels, further contributing to improve performance.

Figure 4.3 presents the total execution time (clock cycles) of seven compute-intensive kernels. The execution time is normalized with respect to that of or1k processor, where the kernels are compiled with -O3 optimization flag. The IPA outperforms the CPU by up to 20.3×, with an average speed-up of 9.7×. A quantitative performance comparison with respect to the CPU is presented in Table 4.3. The table presents the configuration and execution cycles in the IPA for different kernels. It also presents the average utilization of PEs over the total execution period and total number of instructions executed in the IPA. The instruction count includes the instructions that are replicated on all the active PEs for keeping the PE in synch across conditionals and jumps. It also includes NOPs that are used when some PEs are stalled due to manipulation of index variables. However, during NOP execution PEs are clock gated and do not consume dynamic power. The IPA achieves a maximum of 18× and an average of 9.23× energy gain over the CPU.

Table 4.3 Overall instructions executed and energy consumption in IPA vs CPU

Kernels		**FIR**	**MatM (16×16)**	**Convolution**	**SepFilter**	**NonSepFilter**	**FFT**	**DC Filter**
IPA	**Configuration cycles**	71	88	88	90	98	87	145
	Execution cycles	6071	11940	56241	827685	1852382	8076	4748
	Total number of instructions executed	44294	110946	531815	7349843	17486486	76310	28868
	Active PEs/cycle(%)	46.1	58.5	59.2	55.5	59	59.7	39.5
	Energy (μJ)	0.022	0.043	0.202	2.98	6.669	0.032	0.017
	Energy (μJ) in non-clock-gated IPA	0.047	0.077	0.479	7.152	11.704	0.063	0.045
CPU	**Execution cycles**	37677	96256	616805	5982730	9084101	164480	50085
	Energy (μJ)	0.132	0.337	2.159	20.94	31.794	0.576	0.175
	Speed-up	6.21x	8.06x	10.97x	7.23x	4.9x	20.3x	10.55x
	Energy-gain	6x	7.84x	10.69x	7.03x	4.77x	18x	10.29x

To establish the impact of the memory bandwidth over performance and energy efficiency, we vary the number of LSUs in the PE array from 4 to 16 and the number of TCDM banks from 4 to 32. The number of LSUs defines the available bandwidth from the TCDM to the

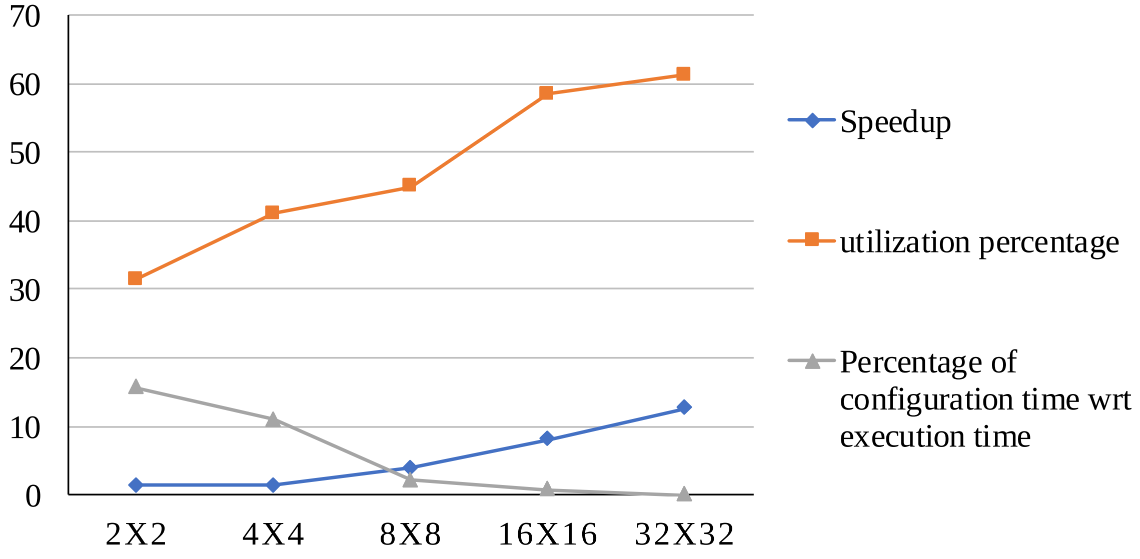


Fig. 4.2 Performance of IPA executing matrix multiplication of different size

array, while increasing the number of TCDM banks reduces the banking conflict probability, improving performance. To perform the exploration without any bias towards configurations, the innermost loops of the kernels are unrolled to get a maximum of 16 load-store operations in one cycle (as the highest number of LSUs considered is 16, in the exploration). In Figure 4.3, each configuration is represented as a 2-dimensional number, where the first one represents the number of LSUs, and the second one represents the number of TCDM banks.

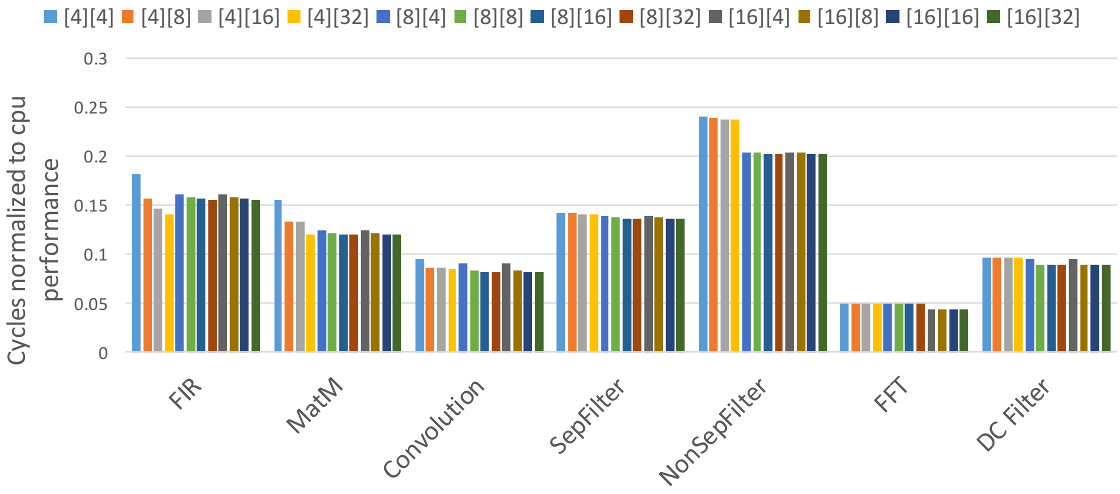


Fig. 4.3 Latency performance in different configurations ([#LSUs][#TCDM Banks])

Results show that, as opposed to tightly coupled clusters of processors which require a banking factor of 2 (i.e. number of TCDM banks is twice the number of cores) [91], IPA performance is almost insensitive to the number of TCDM banks, and a configuration with a banking factor of 0.5 is sufficient to minimize the impact of contention on the shared memory banks for most applications. Indeed, while the typical processor execution requires several load/store operations for variables exceeding the size of the register file, direct CDFG mapping on the IPA does not add extra memory operations except primary inputs and outputs (e.g. arrays), since all the temporary variables are stored in the register file of the PEs. Moreover, flexible point-to-point connections within the array allow to efficiently exchange data among PEs, further reducing the pressure on the TCDM. This concept is well explained in Figure 2.10 and Figure 2.4, which show the typical mapping of an application on the IPA.

Energy Efficiency

Figure 4.4 shows the average breakdown of power consumption for different configurations of the IPA. As expected, the PE array is the most dominant power consumer for all the configurations. The configurations with 4 TCDM banks achieve the best power advantages in each group, since increasing the number of TCDM banks increases the complexity of the interconnect, causing timing pressure on the array, which increases the sizing of the cells, hence power consumption.

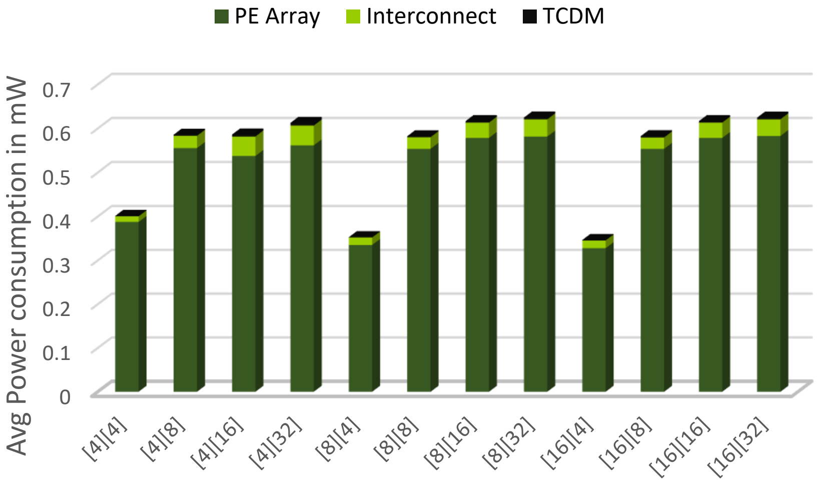


Fig. 4.4 Average power breakdown in different configurations ([#LSUs][#TCDM Banks])

Figure 4.5 shows the average energy efficiency (MOPS/mW) for different configurations. Million Operations Per Second (MOPS) only considers the active PEs during execution, since a PE may be idle due to TCDM bank access conflicts, consecutive NOPs, or not mapped (not used in the application execution). Executions with high number of active PEs/cycle achieve large MOPS. As depicted in Figure 4.5, for different number of LSUs in the PE array, the configuration with 4 TCDM banks achieves the best energy efficiency, since this is the least number of banks in each configuration, it causes lowest power consumption. At the same time, the active number of PEs/cycle does not get significantly impacted due to the least memory access policy of the compilation. As a result, the best efficiency is achieved at 2306 MOPS/mW for matrix multiplication, in a configuration with 8 LSUs and 4 TCDM banks. The minimum energy efficiency is achieved at 1112 MOPS/mW for separable filter in a configuration with 4 LSUs and 16 TCDM banks.

To investigate the power gain in the fine-grained clock gating we present the energy consumption of the clock gated IPA and the non clock gated IPA in Table 4.3. In an average, the clock gated design consumes an average of 2× less power compared to that of the non clock gated design. Due to the regular architecture of the PE array, fine grained power management is much more suitable to implement. Moreover, thanks to the efficient execution of CDFG on the array, the smaller energy required to execute an instruction in the IPA with respect to a CPU (5.6E-07 μJ vs 3.49E-06 μJ), and the effectiveness of the fine-grained power management the IPA outperforms the or1k CPU's energy efficiency by up to 18× (Table 4.3). The energy per instruction execution in the IPA is much less than that of the CPU due to its simple instruction set architecture. Also, the lower number of memory operations executed in the IPA helps reducing on the average energy consumption.

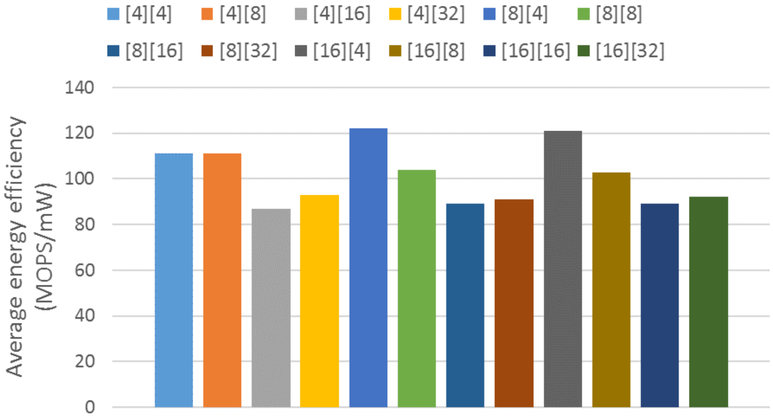


Fig. 4.5 Average energy efficiency for different configurations ([#LSUs][#TCDM Banks])

4.1.3 Comparison with low-power CGRA architectures

Table IX shows a comparison with existing CGRAs. For some papers, energy efficiency figures could not be extracted, so 'NA' is put in the corresponding cell. The energy efficiency

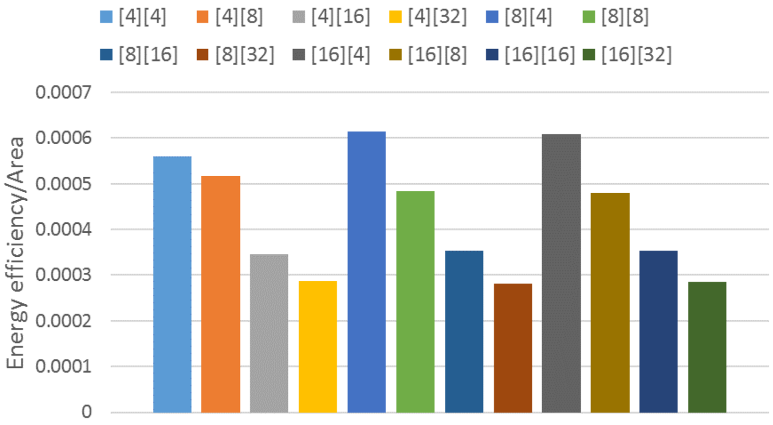


Fig. 4.6 Energy efficiency/area trade-off between several configurations ([#LSUs][#TCDM Banks])

results for Morphosys, Imagine and ReMAP presented in the table are studied in [21]. The energy efficiency figures of the other architectures are provided both in the original manufacturing technology node and scaled to the 28nm technology, according to the power scaling factor $C * V^2$. C and V represent the effective capacitance (approximated with the channel length of the technology) and the supply voltage of the designs, normalized to the nominal parameters of the 28nm technology node. It should be noted that this simplified scaling factor penalizes our design, since deep-submicron technologies such as 28nm, where the load capacitance of gates is typically dominated by wires require much more buffering than mature technology nodes, which penalizes energy efficiency. Nevertheless, IPA provides leading-edge energy efficiency, surpassing by more than one order of magnitude other architectures (ADRES, Morphosys, XPP, AsAP) featuring a C based mapping flow. The driving factors for this gain are (a) architectural simplicity with less complex interconnect network, (b) low power instruction processing, (c) lowest possible number of memory operations in application execution, (d) fine grained power management architecture, described in previous sections. One distinguishing characteristic of the proposed accelerator is the flexible execution model capable of implementing CDFG on the array without the need of a host processor, coupled with a fully automated mapping flow that starts from a plain ANSI C description of the application. Moreover, the memory architecture, based on a shared multi-banked TCDM enables easy integration within ultra-low-power tightly coupled clusters of processors, while fine-grained power management allows to improve energy efficiency by up to 2×. The average power consumption on the IPA is 0.49mW, which is compatible with the ultra-low power target.

Table 4.4 Comparison with the state of the art low power targets

Ref	**Arch**	**Maps**	**Source**	**Access local mem**	**Tech [nm]**	**Supp volt**	**Area [mm^2]**	**Power [mW]**	**Freq [MHz]**	**Area eff [MOPS /mm^2]**	**Energy eff [MOPS /mW]**	**Energy eff scaled to 28nm tech [MOPS /mW]**	**Perf [MOPS]**
High performance targets													
[21]	Morphosys	DFG	ANSI C	PE	150	1.8V	256	4000	450	113	7.20	150	28800
[21]	Imagine	DFG	NA	PE	150	1.5V	144	4000	296	165	12.40	150	23700
[103]	RAW	CDFG	ANSI C	PE	150	1.8V	256	2288	100	NA	NA	NA	NA
[95]	TRIPS	DFG	NA	PE	130	1.0V	336	35868	366	NA	NA	NA	NA
[21]	ReMAP	CDFG	NA	PE	180	1.62V	8.28	312	200	386	10.30	173	3200
Low power targets													
[56]	TCPA	CDFG	Customized	VLIW	90	1.0V	15	12.48	200	106	112.00	360	1587
[86]	Layers	CDFG	NA	PE	65	1.0	0.35	44.45	488	2786	21.94	72	975
[64]	SmartCell	CDFG	Customized	PE	130	1.0V	8.2	160	100	13.04	37.8	176	6048
[41]	PipeRench	DFG	Customized	PE	180	1.8V	55.5	675	120	NA	NA	NA	NA
[82]	SYSCORE	CDFG	NA	PE	90	1.0V	5.73	18.5	100	NA	NA	NA	NA
[8]	ADRES	DFG	ANSI C	VLIW	90	1.0V	15	80	100	94	17.51	56	1409
[11]	XPP	CDFG	ANSI C	PE	90	1.0V	42	93	150	310	10.00	32	13000
[108]	AsAP	CDFG	ANSI C	PE	180	1.8V	23.76	84	116	40	11.00	229	942
[93]	MUCCRA-3	DFG	Customized	VLIW	65	1.2V	8.82	11	41.4	NA	NA	NA	NA
Ultra-low power targets													
[70]	Lopes et al	DFG	NA	PE	90	1.0V	0.45	3.47	100	222	28.8	92.6	100
[73]	CMA	DFG	Customized	μC	65	0.5V	25	1.6	85	^{2}3	^{2}186	^{2}430	^{2}74
[33]	SIMD-CGRA	DFG	ANSI C	PE	65	0.9	0.59	NA	1	NA	NA	NA	NA
[53]	ULP-SRP	DFG	ANSI C	VLIW	40	0.5V	NA	0.21	7	NA	NA	NA	NA
This work	IPA	CDFG	ANSI C	PE	28	0.6V	0.25	0.36	100	3036	1617.00	1617.00	759

4.2 Compilation

The results of the exploration show that a configuration of the IPA with 8 load-store units and 4 TCDM banks achieves the optimal performance/energy trade-off featuring an average speed-up of 9.7× (max 20.3×, min 4.9×) compared to a general-purpose processor. Thanks to the optimized architecture and mapping flow, the proposed accelerator achieves an average energy efficiency of 1617 MOPS/mW over a wide range of sensor signal processing kernels, surpassing other CGRA architectures featuring a C based mapping flow by more than one order of magnitude.

4.2.1 Performance evaluation of the compilation flow

This section analyses the performance and energy consumption results compiling kernels using the compilation flow described in the previous chapter. First, we compare the *register allocation* approach with different *predication* techniques to handle control in applications. Next, we perform some experiments while running several kernels involved in a smart visual trigger application, as the context of smart visual applications the shortcoming of traditional CGRAs is quite severe, since after brute-force morphological filtering (e.g. erosion, dilatation, sobel convolution), these algorithms usually require the execution of highly control intensive code for high-level feature extraction. In this experiment we perform trigger based feature extraction in the IPA compiling kernels using the flow.

All the designs used in the following experiments, were synthesized with Synopsys design compiler 2014.09-SP4 in STMicroelectronics 28nm UTBB FD-SOI technology. Synopsys PrimePower 2013.12-SP3 was used for timing and power analysis at the supply voltage of 0.6V, 25.C temperature, in typical process conditions.

4.2.2 Comparison of the register allocation approach with state of the art predication techniques

To evaluate the efficiency of the register allocation approach to handle the control flow we compare the execution of six control intensive kernels compared to the state of the art partial and full predication techniques. The results, presented in Table 4.5 show that the register based approach achieves a maximum of 1.33× (with minimum of 1.04× and average of 1.13×) and 1.8× (with minimum of 1.37× and average of 1.59×) performance gain compared to partial predication and full predication techniques. The maximum gain

---

[2] PEs perform 8-bit operations, hence energy efficiency is normalized to equivalent 32-bit operations, does not include the power of controlling processor.

achieved over existing methods are highlighted in bold in the table. The smaller number of executed instructions allows the register allocation approach to outperform the partial and full predication techniques by an average of 1.54× (with min 1.35×, max 2×) and 1.71× (with min 1.44×, max 2×) respectively in terms of energy efficiency. The table also presents a comparison with respect to or1k CPU and C64 DSP processor [51] from TI. The register allocation approach achieves a maximum of 3.94×, 15.8× performance gain and 7.52×, 32.77× energy gain over or1k and C64 processor, respectively. Due to the abundance of branches in these kernels, the DSP processor performs worst. Finally, we compare with the basic systematic load-store (SLS) based approach for control mapping. It is depicted from the Table 4.5 and 4.6 that the register allocation approach performs an average of 1.16× (with max of 1.46×, min of 1.05×) better than the SLS based approach, while gaining an average of 1.31× energy efficiency with a maximum gain of 2× and minimum gain of 1.07×.

Table 4.5 Performance (cycles) comparison between the register allocation approach and the state of the art approaches

Kernels	**# loops**	**# conditionals**	**IPA**				**CPU**	**C64 DSP**
			reg based	**SLS based**	**partial pred**	**full pred**		
cordic	1	2	328	408	396	542	513	286
cordic	1	2	328	408	396	542	513	286
sobel	4	11	179617	**262282**	188253	245583	454028	669794
gcd	1	1	55312	58596	**73747**	92852	67545	92184
sad	2	1	15962	16824	16573	**28776**	**62932**	**252193**
deblocking	5	7	472258	495081	518722	727243	834683	1310220
manh-dist	1	1	6288	6826	6738	9522	15394	55317
	max gain			1.46x	1.33x	1.8x	3.94x	15.8x

4.2.3 Compiling smart visual trigger application

Performance and energy consumption: This section provides performance comparison of IPA running at 100 MHZ with respect to a or10n CPU [40] running at 45 MHZ clock frequency, that are the operating frequency of the two architectures at the operating voltage of 0.6V. The experiment is carried out on a smart visual surveillance application [77] performing on 160x120 resolution of images, consisting 9 different motion detection kernels including morphological filters (e.g. finding minimum and maximum pixel, erosion, dilatation, Sobel convolution), and a smart trigger kernel asserting an alarm if the size of the detected objects surpasses a defined threshold, the latter kernel composed of highly control intensive code. To compile the applications for the IPA, we use the compilation flow. Table 4.7 shows the

Table 4.6 Energy consumption (μJ) comparison between the register allocation approach and the state of the art approaches

Kernels	# loops	# conditionals	IPA				CPU	C64 DSP
			reg based	SLS based	partial pred	full pred		
cordic	1	2	328	408	396	542	513	286
cordic	1	2	0.001	**0.002**	**0.002**	**0.002**	0.004	0.002
sobel	4	11	0.736	1.102	1	1.058	3.531	5.656
gcd	1	1	0.227	0.246	0.392	0.4	0.525	0.778
sad	2	1	0.065	0.071	0.088	0.124	**0.489**	**2.13**
deblocking	5	7	1.936	2.079	2.754	3.134	6.492	11.064
manh-dist	1	1	0.026	0.029	0.036	0.041	0.12	0.467
	max gain			2x	2x	2x	7.52x	32.77x

performance comparison executing the application in the IPA (programmed in plain ANSI C code) and a highly optimized core with the support for vectorization and DSP extensions that can only be exposed optimizing the source code with intrinsics [40]. The IPA surpasses the CPU by 6x and 10x in performance and energy consumption, respectively. It is interesting to notice that while DSP instructions do not improve the performance of the core during execution of the smart trigger kernel, its implementation on the IPA provides even more benefits with respect to the data-flow part of the application (motion detection), improving performance by 10x with respect to execution on the processor.

Table 4.7 Performance comparison

Applications	CPU	CPU	CPU (optimized)	IPA
Motion Detection	cycles	2 237 124	1 308 036	261 120
	energy[μJ]	10.179	5.952	0.679
	perf gain	9x	5x	
	energy gain	15x	9x	
Smart Trigger	cycles	480 000	480 000	48 000
	energy[μJ]	2.184	2.184	0.125
	perf gain	10x	10x	
	energy gain	17x	17x	
Overall	cycles	2 707 200	1 785 600	309 120
	energy[μJ]	12.318	8.124	0.804
	performance gain	9x	6x	
	energy gain	15x	10x	

Comparison with the state of the art architectures: Table 4.8 presents the performance comparison of the smart visual trigger application running on a CPU and two state of the art reconfigurable array architectures [77] [89], chosen due to the availability of the target application, with similar features to other state of the art CGRAs. Results show that, although the two state of the art CGRAs deliver huge performance when dealing with the data-flow portion of the application, thanks to highly optimized and pipelined datapath that allows to implement operations on binary images as Boolean operations [77], they are not able to implement the control dominated kernel, which runs on the CPU forming a major bottleneck for performance when considering the whole application. On the other hand, the superior flexibility of the IPA allows to implement the whole application on the accelerator, allowing to surpass performance of other CPU + CGRA systems by 1.6x. It is important to note that in the context of more complex smart vision applications, such as context understanding and scene labelling, it is common that control intensive kernels dominate the overall execution time share, further improving performance with respect to CGRA accelerators only able to map DFGs.

Table 4.8 Performance comparison of smart visual surveillance application [cycles/pixel]

Reference	Motion Detection	Smart Trigger	Total
CPU	116	25	141
CPU (Optimized)	68	25	93
Mucci et al [77]	2.09	25	27.09
Rossi et al [89]	1.27	25	26.27
IPA	13.6	2.5	16.1

4.3 Conclusion

With respect to state of the art partial and full predication techniques, the proposed compilation flow improves performance by 1.54× on average (min 1.35×, max 2×) and energy efficiency by 1.71× on average (min 1.44×, max 2×). The experiment on the target smart visual trigger application show that the IPA achieves an average performance of 507 MOPS with average energy efficiency of 142 MOPS/mW at 0.6V surpassing a general purpose processor by 6x in performance and 10x in energy efficiency. The proposed IPA also surpasses the state of the art CGRA architectures performance by 1.6x, thanks to the capability of efficiently implementing control intensive code. In the next chapter, we integrate the IPA in a multi-core cluster and present the energy efficiency aspects of heterogeneous computing.

Chapter 5

The Heterogeneous Parallel Ultra-Low-Power Processing-Platform (PULP) Cluster

High performance and extreme energy efficiency are strict requirements for many deeply embedded near-sensor processing applications such as wireless sensor networks, end-nodes of the Internet of Things (IoT) and wearables. One of the most traditional approaches to improve energy efficiency of deeply embedded computing systems is achieved exploiting architectural heterogeneity by coupling general-purpose processors with application- or domain-specific accelerators in a single computing fabric. On the other hand, most recent ultra-low power designs exploit multiple homogeneous programmable processors operating in near-threshold [88]. Such an approach, which joins parallelism with low-voltage computing, is emerging as an attractive way to join performance scalability with high energy efficiency.

The concepts of parallelism and heterogeneity in ultra-low power designs inherit from traditional high-end embedded platforms such as NVIDIA Tegra [79], IBM PowerEN processor [57], Qualcomm Snapdragon S4 Pro [99], STMicroelectronics P2012 [6], Kalray MPPA [26].

In this chapter, we present a heterogeneous architecture which integrates a near-threshold tightly-coupled cluster of processors [88] augmented with the Integrated Programmable Array (IPA) presented in [24]. We synthesized the architecture in a 28nm FD-SOI technology, and we carried out a quantitative exploration combining physical synthesis results (i.e. frequency, area, power) and benchmarking on a set of signal processing kernels typical of end-nodes applications. One interesting finding of our exploration is that (1) the performance of the IPA is much less sensitive to memory bandwidth than parallel processor clusters [24] and (2) the simpler nature of its architecture allows to run 2× faster than the rest of the system. Experimental results show that the heterogeneous architecture achieves significant performance improvement for both compute and control intensive benchmarks with respect to the software cluster.

5.1 PULP heterogeneous architecture

The PULP platform project is a collaborative effort of several academic and industrial institutions[1], whose goal is to design an ultra-low power achieving high levels of energy efficiency by combining near-threshold computing and parallel computing and by exposing low power features of the technology up the technological stack, at the architecture and software levels.

Ultra-low power operation and extreme energy efficiency are the key features of the implementation of PULP, which exploits *near threshold* computing. The PULP SoC utilizes multi-core parallelism with explicitly-managed shared L1 memory to overcome performance degradation at low voltage, while keeping the flexibility typical of instruction processors. Moreover, enabling the cores to operate on-demand over a wide supply voltage and body bias ranges allows to achieve high energy efficiency over a wide spectrum of computational demands.

5.1.1 PULP SoC overview

Figure 5.1 shows the main building blocks of a single-cluster PULP SoC. The PULP cluster features 8 32-bit RISC-V cores based on a four pipeline stages micro-architecture optimized for energy-efficient operation [39] sharing a 64KB multi-banked scratchpad memory through a low-latency interconnect [85]. The ISA of the cores is extended with instructions targeting energy efficient digital signal processing such as hardware loops, load/store with pre/post increment, SIMD operations. The cores share a 4KB private instruction cache to boost performance and energy efficiency for tightly coupled clusters of processors typically relying on data parallel computational models [69]. Off-cluster data transfers are managed by a lightweight multi-channel DMAs optimized for energy-efficient operation [90]. Both the (I$) and DMA connects to an AXI4 cluster bus. A peripheral interconnect is used to communicate with on-cluster peripherals such as a timer, an event-unit used to accelerate synchronization among the cores and other memory mapped peripherals such as application-specific accelerators. To operate at the best operating point for a given workload the cluster can be integrated in an independent voltage and frequency domain, featuring dual-clock FIFOs and level-shifters at its boundary.

---

[1] Includes the University of Bologna, ETH Zurich, STMicroelectronics, EPFL Lausanne, Politecnico di Milano and others.

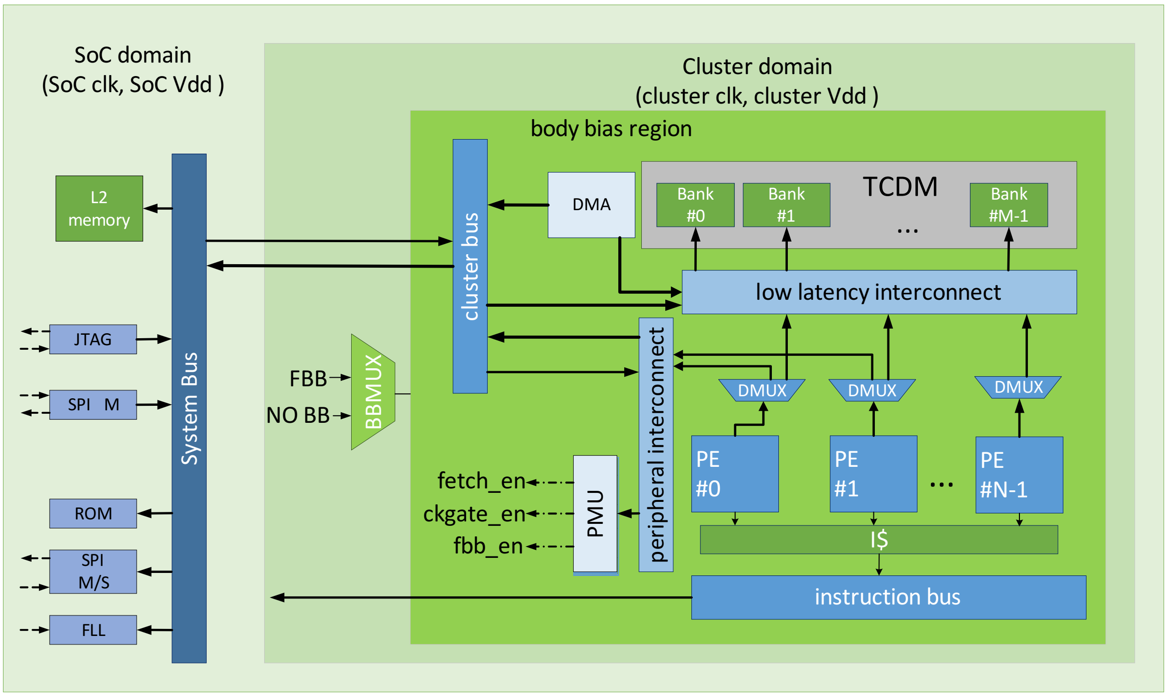

SoC domain
(SoC clk, SoC Vdd)
Cluster domain
(cluster clk, cluster Vdd)
body bias region
L2 memory
JTAG
SPI M
ROM
SPI M/S
FLL
System Bus
FBB
NO BB
BBMUX
cluster bus
DMA
Bank #0
Bank #1
TCDM
...
Bank #M-1
low latency interconnect
peripheral interconnect
DMUX
DMUX
DMUX
PE #0
PE #1
...
PE #N-1
fetch_en
ckgate_en
fbb_en
PMU
I$
instruction bus

Fig. 5.1 PULP SoC. Source [91]

5.1.2 Heterogeneous Cluster

The PULP cluster is augmented with the Integrated Programmable Array accelerator, as shown in Figure 5.2. Figure 5.3 shows a detailed block diagram of the subsystem embedding the IPA array. The IPA array is configured through a global context memory (GCM), responsible for storing locally the configuration bitstream of the PEs. The GCM is connected through a DMA-capable AXI-4 port to the cluster bus, enabling pre-fetching of IPA contexts from L2 memory. The size of GCM is considered twice the size of configuration bitstream of the IPA in the worst case, in this way it is possible to employ a double-buffering mechanism and load a new bitstream from the L2 to the GCM when the current one is being loaded on the array, completely hiding time for reconfiguration. More details on the structure of the IPA array bitstream can be found in [25]. A set of memory mapped control registers allow to load a new context to the IPA array, trigger the execution of a kernel and synchronize with the other processors in the cluster.

As opposed to many CGRA architectures, the IPA can access a multi-banked shared memory through 8 master ports connected to the low-latency interconnect. This eases data

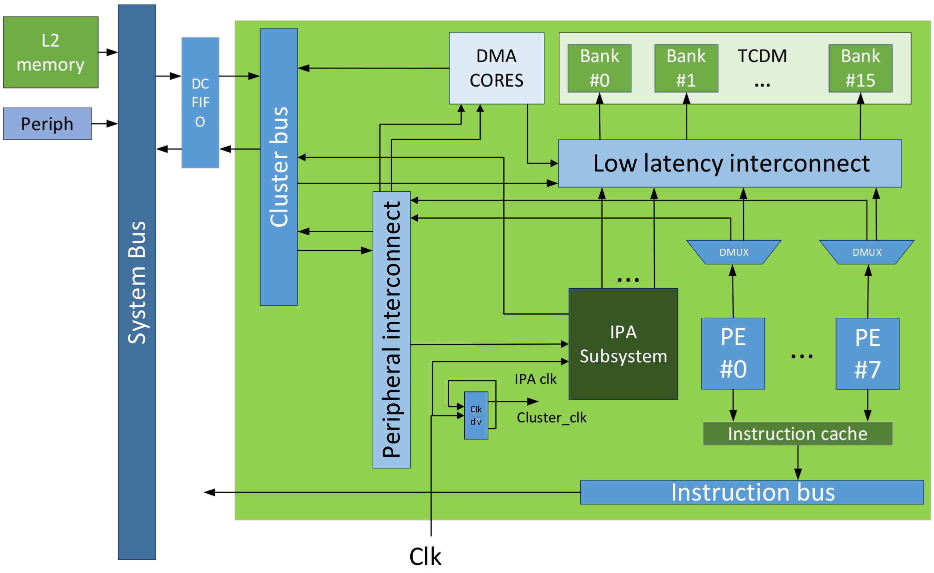

L2 memory
Periph
System Bus
DC FIFO
Cluster bus
DMA CORES
Bank #0
Bank #1
TCDM
...
Bank #15
Low latency interconnect
Peripheral interconnect
DMUX
DMUX
...
IPA Subsystem
PE #0
...
PE #7
IPA clk
Clk div
Cluster_clk
Instruction cache
Instruction bus
Clk

Fig. 5.2 Block diagram of the heterogeneous cluster.

sharing with the other processors of the cluster, following the computational model described in [18]. The optimal number of port has been chosen to optimize the trade-off between the size of the interconnect and the bandwidth requirements of the IPA. Following the analysis conducted in [24], where it is shown that the IPA can operate 2× faster than the processors, we have extended the architecture of the cluster in a way that the IPA can work at twice the frequency of the rest of the cluster. This approach allows to operate each component in the cluster at the optimal frequency, without paying the overheads of dual-clock FIFOs, requiring a significant amount of logic and synchronization overhead. On the contrary, the hardware support for the dual-frequency mode includes a clock divider to generate the two different edge aligned clocks, and two modules needed to adapt the request-grant protocol of the low-latency interconnect [85] to deal with the frequency domain crossing, as shown in Figure 5.4.

5.2 Software infrastructure

To offload jobs to the IPA and synchronize the execution, the cores access the *control registers* of the IPA, by memory mapped operations. The control registers are composed of a *command register* and a *status register*. We designed a simple Application Programming Interface (APIs) to perform the offload and synchronize tasks with the IPA. The main functions are described in Table 5.1. Before execution starts in the IPA accelerator, the cores load the corresponding context and data from the L2 memory to the GCM and L1

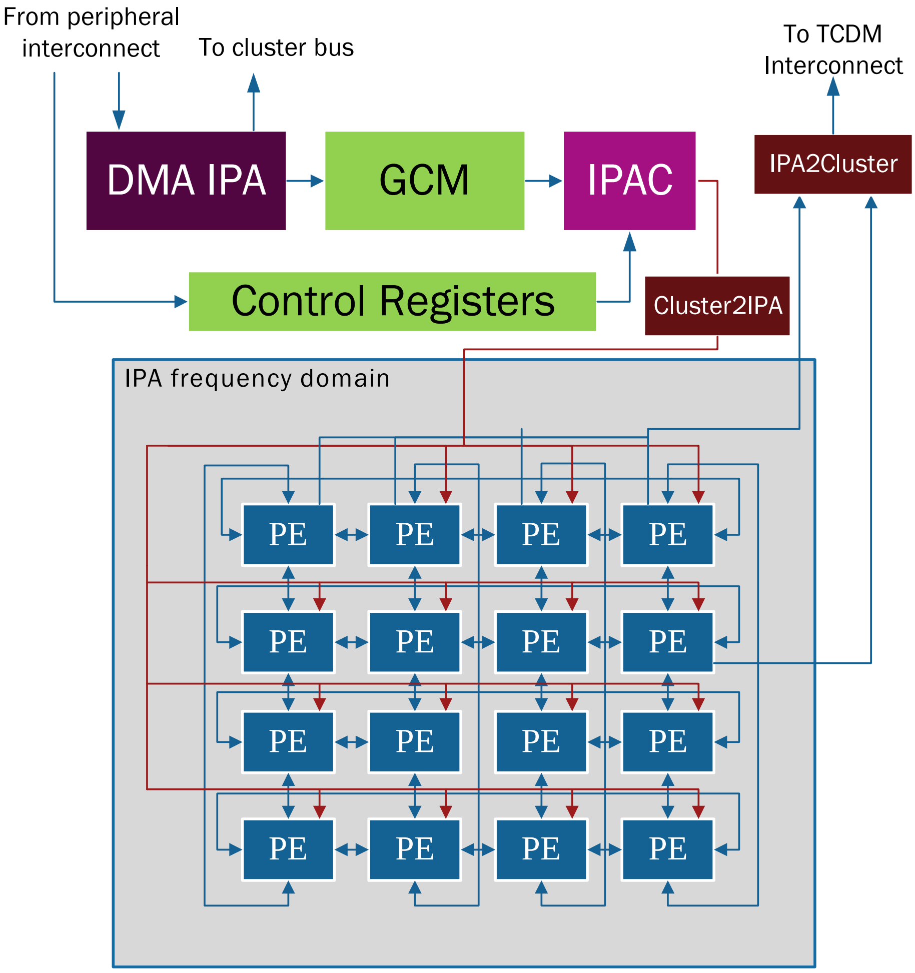

From peripheral interconnect
To cluster bus
To TCDM Interconnect
DMA IPA
GCM
IPAC
IPA2Cluster
Control Registers
Cluster2IPA
IPA frequency domain
PE

Fig. 5.3 Block diagram of the IPA subsystem.

memory, programming the system DMA and the IPA DMA, respectively. The context for the IPA consists of instructions and constants for the PEs, generated by the compilation flow proposed in [24]. The functions *load_data_l2totcdm* and *load_context_l2togcm* contain a set of routines to write data and context into TCDM and GCM respectively. The *ipa_start_-execution* writes *execute* command into the *command register* of the IPA. The completion of the execution is notified by updating the *status register*. The core is synchronized with the IPA execution by calling the *ipa_check_status* function, which checks for the updates in the status register.

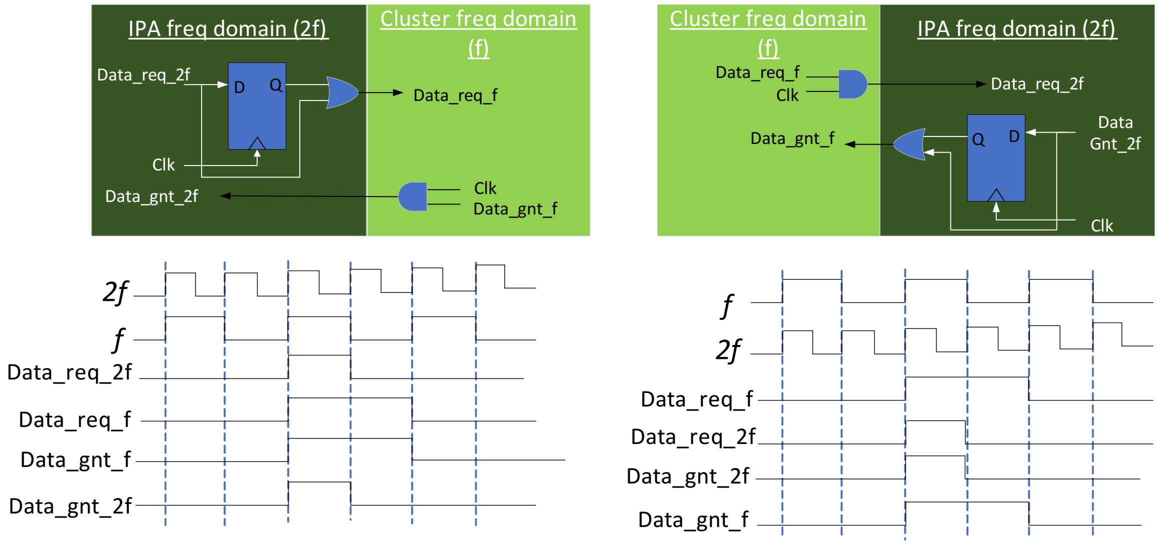

IPA freq domain (2f)
Cluster freq domain (f)
Data_req_2f
D
Q
Data_req_f
Clk
Data_gnt_2f
Clk
Data_gnt_f
2f
f
Data_req_2f
Data_req_f
Data_gnt_f
Data_gnt_2f
Cluster freq domain (f)
IPA freq domain (2f)
Data_req_f
Clk
Data_req_2f
Data_gnt_f
Q
D
Data Gnt_2f
Clk
f
2f
Data_req_f
Data_req_2f
Data_gnt_2f
Data_gnt_f

Fig. 5.4 Synchronous interface for reliable data transfer between the two clock domains.

Table 5.1 List of APIs for controlling IPA

Function	Description
void load_data_l2totcdm (int DMA_CORE_ID, int size, unsigned int l2_addr, unsigned int tcdm_addr)	Writes data from L2 memory to the TCDM banks through DMA_CORE
void load_context_l2togcm (int DMA_IPA_ID, int size, unsigned int l2_addr, unsigned int gcm_addr)	Writes context from L2 memory to the GCM through DMA_IPA
int ipa_start_execution ()	Initiate IPA execution by writing in the command register
void ipa_check_status(in id)	Core synchronization
void free_ipa (int id)	Release IPA

5.3 Implementation and Benchmarking

In this section we present the implementation results of the heterogeneous PULP cluster. The three possible modes considered in these comparisons are: (a) single-core: running applications in a single core, (b) ipa: running applications in the IPA where the core takes part in offloading only, (c) multi-core: running applications in parallel cores. All the benchmarks are coded in fully portable C, using the OpenMP programming model to express parallelism for PULP.

5.3.1 Implementation Results

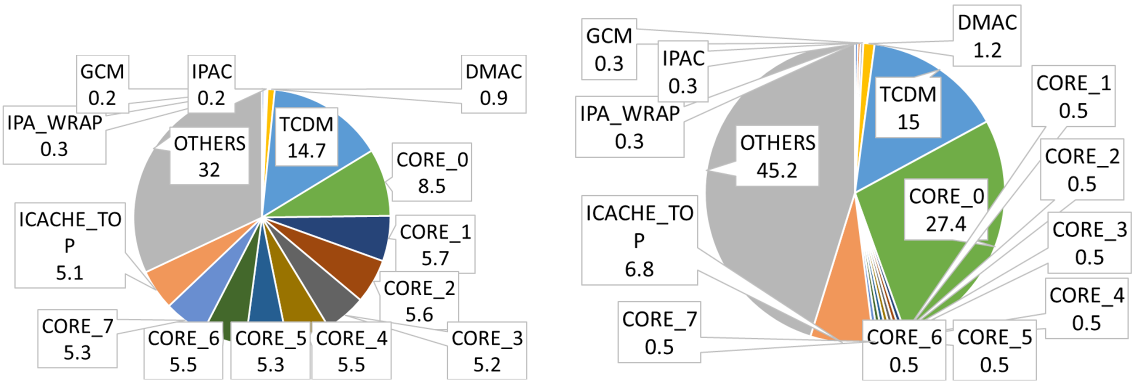
Table 5.2 presents the details of the memories used in the cluster. It consists of 8 cores featuring 4 kB of shared I$, one IPA with 16 PEs and a GCM of 4KB, while the TCDM is composed of 16 banks of 4 kB each, leading to an overall TCDM size of 64 kB. These architectural parameters were chosen to fit the constraints of the wide range of signal processing benchmarks. The SoC was synthesized with Synopsys Design Compiler 2013.12-SP3 on a STMicroelectronics 28nm UTBB FD-SOI technology library. Since the achievable frequency of the PEs in the IPA is higher than the RI5KY cores used in the cluster, the IPA is clocked at 100 MHz, while the rest of the cluster runs at 50 MHz (in the SS, 0.6V, −40°C corner). Synopsys PrimePower 2013.12-SP3 was used for timing and power analysis at the supply voltage of 0.6V, 25°C temperature, in typical process conditions. Table 5.3 presents the area information of the components in the cluster. Although the total area of the IPA with 16 PEs is almost similar to the area of the 8 cores combined, the area occupied by the GCM is much less than the total cache memory, which in turn provides better area efficiency while running applications in IPA.

Table 5.2 Cluster Parameters and memories used

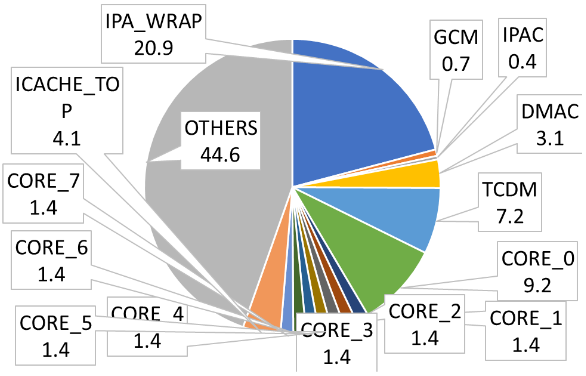
Name	**Type**	**Size**
L1 Memory (16 banks)		
TCDM	SRAM	64 KB
Cores (8)		
Instruction Cache	SRAM	4KB
IPA (16 PEs)		
Global context memory	SRAM	8KB
Instruction Register File (IRF)	Registers	0.16KB
Regular register file (RRF)	Registers	0.032KB
Constant register fie(CRF)	Registers	0.128KB

5.3.2 Performance and Energy Consumption Results

Table 5.4 reports the execution time in nano seconds for different benchmarks running on a single-core, on 8 cores and on the IPA. The IPA execution time includes the time taken for loading the context into the PEs. Comparing to the performance of execution in single-core, the accelerator achieves a maximum of 8× (with a minimum of 2.49× and an average of 5.4×) speed-up. The control intensive kernel like GCD does not exhibit parallelism, hence parallel software execution does not improve performance of the homogeneous cluster. On the other hand, the execution on the IPA improves the performance by almost 5×, exploiting also

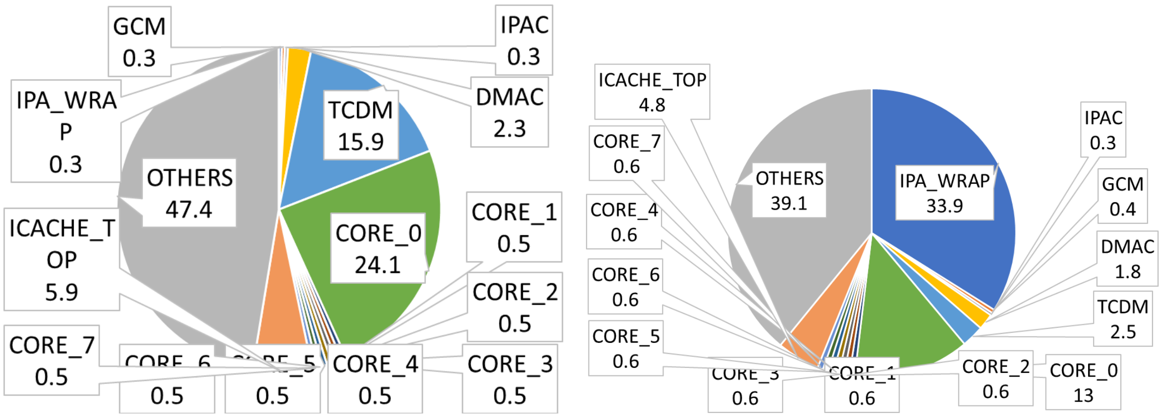


(a) 0.78mW (b) 0.44mW



(c) 0.50mW

Fig. 5.5 Power consumption breakdown in percentage: Executing Matrix-Multiplication in (a) Multi-Core; (b) Single-Core; (c) IPA. Executing GCD in (d) Single-Core; (e) IPA. (OTHERS contain peripherals, interconnect, clk-gate, bbmux)



(a) 0.39mW (b) 0.41mW

Fig. 5.6 Power consumption breakdown in percentage: Executing GCD in (d) Single-Core; (e) IPA. (OTHERS contain peripherals, interconnect, clk-gate, bbmux)

Table 5.3 Synthesized area information for the PULP heterogeneous cluster

Components	Area (μm^2)	% of cluster area
CORES	160,352	18
ICACHE	190,089	22
DMA_CORE	41,406	5
IPA	156,323	18
DMA_IPA	32,636	4
GCM	18,704	2
TCDM	149,638	17
CLUSTER_INTCNCT	63,126	7
CLUSTER_PERIPHERALS	21,610	2
OTHERS	37,932	4
Total	871,816	100

instruction-level parallelism rather than data-level parallelism only. The performance gain in the accelerator for the compute intensive kernels like matrix multiplication, convolution, FIR and separable filters is limited if compared to the performance of parallel-cores. However, the relatively small performance gain compared to the parallel cluster is compensated by the gain in energy consumption (Table 5.6) due to the simpler nature of the compute units of the IPA with respect to full processors, to the smaller number of power-hungry load/store operations (Table 5.7), and to the fine-grained power management architecture that allows clock gate the inactive PEs during execution (Table 5.6).

Table 5.5 presents the performance improvement of the IPA when moving from iso-frequency to the 2× frequency domain execution in the IPA. This shows that, although there is a reduction of memory bandwidth (see loss due to additional stalls column in Table 5.5), since the TCDM operates at the same frequency as the rest of the cluster (i.e. half frequency w.t.t. the IPA array), an average of 1.82× speed-up (with maximum of 1.92× and a minimum of 1.73×) can be achieved with this dual-frequency cluster architecture.

The power consumption profiles for the different modes of execution presented in Figure 5.5 and 5.6, which shows the percentage of contribution by the several components in the cluster. Figure 5.5 (a), (b), (c) represents the power breakdown while executing matrix multiplication in multi-core, single-core and IPA respectively, representative for other compute intensive benchmarks. Similarly, Figure 5.6 (a) and (b) present the profiles for executing GCD, a control intensive benchmark, in single-core and IPA respectively. In Figure 5.5 (a), (b), (c), the TCDM contributes to 14.7%, 15% and 7.2% in the multi-core, single-core and IPA configurations, respectively. The reduced memory access in IPA execution helps to achieve better energy efficiency. While executing GCD in single-core and the IPA, the TCDM

Table 5.4 Performance evaluation in execution time (ns) for different configuration in the heterogeneous platform

Kernels	**Single-core (ns)**	**Multi-core (ns)**	**Speed-up in multi-core (x)**	**IPA (ns)**	**Speed-up in IPA (x)**
MatMul	3,358,740	435,180	7.72	432,630	7.76
Convolution	9,733,380	1,520,840	6.4	1,494,860	6.51
FFT	767,640	142,720	5.38	94,510	8.12
FIR	182,500	33,460	5.45	33,410	5.46
Separable Filter	39,870,420	6,404,160	6.23	6,334,700	6.29
Sobel Filter	117,024,880	40,894,260	2.86	28,865,890	4.05
GCD	2,951,160	2,951,160	1	61,1300	4.83
Cordic	9,000	7,000	1.29	3,610	2.49
Manh Dist	244,640	164,640	1.49	70,300	3.48

Table 5.5 Performance comparison between iso-frequency and 2× frequency execution in IPA

Benchmarks	**#cycles in iso frequency**	**#cycles in 2x frequency**	**Loss due to stalls**	**overall execution speed-up**
MatMul	39,330	43,263	3,933	1.82
Convolution	130,896	149,486	18,590	1.75
FFT	8,182	9,451	1,269	1.73
FIR	3,122	3,341	219	1.87
Separable filter	575,882	633,470	57,588	1.82
Sobel Filter	2,634,172	2,886,589	252,417	1.83
GCD	58,573	61,130	2,557	1.92
Cordic	328	361	33	1.82
ManhDistance	6,391	7,030	639	1.82
Average				1.82

consumed around 15.9% and 2.5% of the total power in the two analysed configurations, respectively. Also, the IPA consumes around 33.9% of the total power while executing the GCD kernel, due to heavy usage of internal registers to support control flow dependencies. The simpler nature of the compute units, low burden on the TCDM and data exchange through PEs explains the energy gain of 7× in the IPA execution.

Table 5.6 Energy consumption evaluation in μ J for different configuration in the heterogeneous platform

Kernels	**Single-core**	**Multi-core**	**IPA**	
			Energy	**of Active PEs/cycle**
MatMul	1.247	0.313	0.208	58.5
Convolution	2.876	1.095	0.658	59.2
FFT	0.292	0.087	0.042	59.7
FIR	0.08	0.026	0.026	46.1
Separable filter	16.663	4.611	4.28	55.5
Sobel Filter	51.491	29.444	12.701	51.2
GCD	1.151	1.151	0.257	6.25
Cordic	0.004	0.003	0.001	50
ManhDistance	0.1	0.095	0.03	48.5

Table 5.7 Comparison between total number of memory operations executed

Benchmarks	**multi-core**	**single-core**	**IPA**
MatMul	66,584	66,561	35,032
Convolution	135,280	135,114	75,600
FFT	12,528	11,733	6528
FIR	5,904	5,893	3,990
Separable filter	142,840	142,800	95,200
Sobel Filter	148,240	148,224	120,000
GCD	64,531	64,531	2
Cordic	32	28	15
ManhDistance	2,158	2,049	2,048

5.4 Conclusion

In this chapter, we presented a novel approach towards heterogeneous computing, augmenting ultra-low power reconfigurable accelerator in the PULP multi-core cluster. The experiments integrating IPA in the PULP platform suggests that architectural heterogeneity is a powerful approach to improve energy profile of the computing systems. We have presented three possible executions of the benchmarks in the IPA integrated PULP platform. The heterogeneous cluster achieves achieving up to 4.8× speed-up and up to 4.4× better energy efficiency with respect to an 8-core homogeneous cluster.

Summary and Future work

Coarse-Grained Reconfigurable Architectures (CGRAs) are appealing choice of reconfigurable accelerator platforms to explore both performance and energy efficiency, the two most critical metrics in embedded computing domain. Since both energy and performance refinement require better exploring the underlying architectures, design of the compiler is much of importance. On the one hand the design of the computation unit, interconnect network strategy along with the computation model, decides the compiler complexity and flexibility of computing. On the other hand, compiler capabilities to truly explore the micro-architecture determine the final performance. Hence, the combined design flow is necessary to satisfy performance and power constraints.

In this dissertation, we addressed ultra-low-power acceleration through CGRA approach. In this regard, we have explored several architectural aspects like computation unit, interconnect network, synchronization mechanism and power management issues to design an Integrated Programmable Array (IPA) accelerator operating at 0 to 3 mW power envelop achieving significant performance improvement over ultra-low power processor cores. We also discussed about the compilation approach to accelerate kernels with a pressing concern of minimized memory access in ultra-low-power execution environment. In addition, the compilation approach along with the hardware synchronization makes the framework compatible with applications containing several loops and conditionals nests.

The key aspects of the thesis are listed below:

- *Data and control dependent execution:* In this dissertation, we have pointed out that the framework of a CGRA acceleration must possess the capability to handle both the data and control flow of the application, to dislodge the communication overhead with the host and achieve increased flexibility of execution. We have introduced a *register allocation* approach for supporting the execution of control and data dependence. This approach works independent of program optimizations giving freedom to explore several inner-most loop optimizations (unrolling, software pipelining, pattern oriented optimizations) without involving the host for the initiation of outer loops.

- *Data locality:* One of the key approaches for energy efficient execution is to keep the data as close as possible. This helps to increase latency performance as well as save energy consumption. In this dissertation, the execution of applications considers only the array input and outputs of the application to be processed by load-store operations. All the variables and constants are accessed involving the internal registers of the processing elements. Since the register files of the PEs are distributed, we have formulated the mapping problem on CGRAs while efficiently using registers, we present a unified and precise formulation of the problem of variable placement and register allocation and an effective and efficient placement augmenting the exact binding approach.

- *Two way synchronization:* While mapping and executing applications consisting several basic blocks, it is of utmost importance to synchronize between PEs. In compilation, the PEs are synchronized following the *register allocation* augmented placement algorithm. While executing, the PEs get synchronized to the same basic block in a single cycle following a lightweight synchronization mechanism, reducing performance and energy consumption penalty.

- *Constant management:* Managing constants in an application is one of the major challenges for energy efficient acceleration. Signal processing applications usually use 16 bits constants. For the sake of wide range of application domain support, in this thesis we considered constants maximum of 32 bits width. On the one side, accommodating the constants in the instructions increases its length, on the other hand memory based access of constants escalates the number of memory operations which in turn increase the energy consumption. In this dissertation, we have introduced the concept of *distributed constant register file*, where the constants are loaded as a part of the context load. These are accessed by the PEs at the time of execution as register operands.

- *Two fold interconnect network:* The computation model in this dissertation contemplates the sequential arrangement of context (instruction and constants) load and execution. Since the ratio between the context load time and execution time is very small, we deployed a bus-based network for efficient distribution of context into the PEs. The execution uses a different 2D torus-based network. However, Since the execution of the application is mapped by the compiler, only the 2D torus network is exposed to the compiler. This way we manage to keep configuration time as less as possible, while keeping the energy consumption in the execution checked by using a low cost interconnect network.

- *Coupling in a heterogeneous platform:* The experiments integrating the designed CGRA in the PULP platform suggests that architectural heterogeneity is a powerful approach to improve energy profile of the computing systems. We have presented three possible executions of the benchmarks in the IPA integrated PULP platform. The accelerator execution achieves a maximum of 4.5× (with a minimum of 2× and an average of 3×) energy consumption improvement over the execution in single core and parallel cores respectively.

Directions for Future Research

We believe, there are several directions of research that can be accomplished based on the framework, we have presented in this dissertation.

First, latency improvement through upgrading the binding algorithm in the compilation flow. The placement of operation and the data is managed in the binding algorithm where it takes the location constraints derived in the previous operation and data binding. The underlying graph is transformed if no placement is found in this algorithm. If the graph is transformed due to location constraints, then the latency is increased in each transformation. In this situation, a *guided placement* can help to reduce the number of graph transformations, hence improving the latency.

Second, performance improvement by exploring different loop optimizations. In this thesis, we have only explored the performance gain depending on the loop unrolling optimizations performed on the innermost loop. It may be another research direction to explore performance improvement by other loop optimizations like pattern based *polyhedral model* [67], software-pipelining or combining the possible optimizations.

Third, exploring different application domains other than the signal processing. The emerging domains of approximate computing, cryptographic application domain, machine learning etc., may be the interesting choices to explore for both performance and energy efficiency. Since the framework we have presented in this dissertation has the potential to execute wide range of application domain, it will be highly productive while exploring these emerging domains based on the hardware approach and compilation flow.

Fourth, investigating the potential of IPA by supporting floating point arithmetic. In this dissertation, we rely on computing using integer arithmetic. Supporting floating points could be another research direction considering the IPA architecture as the reference. Since the compiler has the flexibility to adapt several architectural configuration revisions, it will be fruitful to update the IPA architecture to support flexible floating point computation.

Fifth, Just-in-Time (JIT) compilation of kernels at runtime. The IPA integrated in the PULP platform uses pre-compiled contexts of the application. It will be another research direction to introduce JIT compilation at run-time and offload them onto the IPA.

In general, the dissertation presents a heterogeneous approach integrating reconfigurable accelerators into a state of the art multi-core computing platform, which addresses the rising concern of energy efficiency. Based on the framework presented in this thesis, there are several challenging research directions in the domain of ultra-low-power embedded computing which can be exploited.

References

[1] Osama T Albaharna, Peter YK Cheung, and Thomas J Clarke. On the viability of fpga-based integrated coprocessors. In *FCCM*, pages 206–215, 1996.

[2] Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010. ISSN 0001-0782. doi: 10.1145/1735223.1735245. URL http://doi.acm.org/10.1145/1735223.1735245.

[3] Mythri Alle, Keshavan Varadarajan, Alexander Fell, Nimmy Joseph, Saptarsi Das, Prasenjit Biswas, Jugantor Chetia, Adarsh Rao, SK Nandy, Ranjani Narayan, et al. Redefine: Runtime reconfigurable polymorphic asic. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(2):11, 2009.

[4] M. L. Anido, A. Paar, and N. Bagherzadeh. Improving the operation autonomy of simd processing elements by using guarded instructions and pseudo branches. In *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*, pages 148–155, 2002. doi: 10.1109/DSD.2002.1115363.

[5] Jonathan Babb, Russell Tessier, and Anant Agarwal. Virtual wires: Overcoming pin limitations in fpga-based logic emulators. In *FPGAs for Custom Computing Machines, 1993. Proceedings. IEEE Workshop on*, pages 142–151. IEEE, 1993.

[6] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012.

[7] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[8] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 978-3-540-71430-9. URL http://dl.acm.org/citation.cfm?id=1764631.1764633.

[9] Janina A Brenner, JC Van Der Veen, Sándor P Fekete, J Oliveira Filho, and Wolfgang Rosenstiel. Optimal simultaneous scheduling, binding and routing for processor-like reconfigurable architectures. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–6. IEEE, 2006.

[10] Geoffrey Burns, Paul Gruijters, Jos Huisken, and Antoine van Wel. Reconfigurable accelerator enabling efficient sdr for low-cost consumer devices. In *SDR Technical Forum*, 2003.

[11] F. Campi, R. König, M. Dreschmann, M. Neukirchner, D. Picard, M. Jüttner, E. Schüler, A. Deledda, D. Rossi, A. Pasini, M. Hübner, J. Becker, and R. Guerrieri. RTL-to-layout implementation of an embedded coarse grained architecture for dynamically reconfigurable computing in systems-on-chip. In *2009 International Symposium on System-on-Chip*, pages 110–113, Oct 2009. doi: 10.1109/SOCC.2009.5335665.

[12] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[13] Kyungwook Chang and K. Choi. Mapping control intensive kernels onto coarse-grained reconfigurable array architecture. In *2008 International SoC Design Conference*, volume 01, pages I–362–I–365, Nov 2008. doi: 10.1109/SOCDC.2008.4815647.

[14] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *2012 International Conference on Field-Programmable Technology*, pages 285–292, 2012.

[15] Liang Chen and Tulika Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3): 21, 2014.

[16] Pawel Chodowiec and Kris Gaj. Very compact fpga implementation of the aes algorithm. In *Ches*, volume 2779, pages 319–333. Springer, 2003.

[17] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 379–384. ACM, 2012.

[18] Francesco Conti, Andrea Marongiu, and Luca Benini. Synthesis-friendly techniques for tightly-coupled integration of hardware accelerators into shared-memory multi-core clusters. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 5:1–5:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1417-3. URL http://dl.acm.org/citation.cfm?id=2555692.2555697.

[19] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. Gaut: A high-level synthesis tool for dsp applications. In *High-Level Synthesis*, pages 147–169. Springer, 2008.

[20] Darren C Cronquist, Chris Fisher, Miguel Figueroa, Paul Franklin, and Carl Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pages 23–40. IEEE, 1999.

[21] Peng Dai, Xin'an Wang, Xing Zhang, Qiuqi Zhao, Yan Zhou, and Yachun Sun. A high power efficiency reconfigurable processor for multimedia processing. In *2009 IEEE 8th International Conference on ASIC*, pages 67–70, Oct 2009. doi: 10.1109/ASICON.2009.5351604.

[22] Anindya Sundar Das, Satyajit Das, and Jaydeb Bhaumik. Design of rs (255, 251) encoder and decoder in fpga. *International Journal of Soft Computing and Engineering (IJSCE) ISSN*, pages 2231–2307, 2013.

[23] S. Das, T. Peyret, K. Martin, G. Corre, M. Thevenin, and P. Coussy. A scalable design approach to efficiently map applications on cgras. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 655–660, July 2016. doi: 10.1109/ISVLSI.2016.54.

[24] S. Das, K. J. M. Martin, P. Coussy, D. Rossi, and L. Benini. Efficient mapping of cdfg onto coarse-grained reconfigurable array architectures. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132, Jan 2017. doi: 10.1109/ASPDAC.2017.7858308.

[25] Satyajit Das, Davide Rossi, Kevin J. M. Martin, Philippe Coussy, and Luca Benini. A 142mops/mw integrated programmable array accelerator for smart visual processing. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pages 1–4. IEEE, 2017.

[26] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed runtime environment for the kalray mppa®-256 integrated manycore processor. *Procedia Computer Science*, 18:1654–1663, 2013.

[27] Bjorn De Sutter, Paul Coene, Tom Vander Aa, and Bingfeng Mei. Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays. In *ACM Sigplan Notices*, volume 43, pages 151–160. ACM, 2008.

[28] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. Coarse-grained reconfigurable array architectures. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010. ISBN 978-1-4419-6344-4. doi: 10.1007/978-1-4419-6345-1_17. URL http://dx.doi.org/10.1007/978-1-4419-6345-1_17.

[29] Masoud Dehyadegari, Andrea Marongiu, Mohammad Reza Kakoee, Siamak Mohammadi, Naser Yazdani, and Luca Benini. Architecture support for tightly-coupled multi-core clusters with shared-memory hw accelerators. *IEEE Transactions on Computers*, 64(8):2132–2144, 2015.

[30] Gregory Donohoe. Reconfigurable data path processor, April 19 2005. US Patent 6,883,084.

[31] Gregory W Donohoe, David M Buehler, K Joseph Hass, William Walker, and Pen-Shu Yeh. Field programmable processor array: Reconfigurable computing for space. In *2007 IEEE Aerospace Conference*, pages 1–6. IEEE, 2007.

[32] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[33] Loris Duch, Soumya Basu, Rubén Braojos, Giovanni Ansaloni, Laura Pozzi, and David Atienza. Heal-wear: An ultra-low power heterogeneous system for bio-signal analysis. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2017.

[34] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[35] Kevin Fan, Manjunath Kudlur, et al. Modulo scheduling for highly customized datapaths to increase hardware reusability. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 124–133. ACM, 2008.

[36] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. A case for specialized processors for scale-out workloads. *IEEE Micro*, 34(3):31–42, 2014.

[37] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Spr: an architecture-adaptive cgra mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 191–200. ACM, 2009.

[38] Fabio Garzia, Waqar Hussain, and Jari Nurmi. Crema: A coarse-grain reconfigurable array with mapping adaptiveness. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 708–712. IEEE, 2009.

[39] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–14, 2017. ISSN 1063-8210. doi: 10.1109/TVLSI.2017.2654506.

[40] Michael Gautschi, Andreas Traber, Antonio Pullini, Luca Benini, Michele Scandale, Alessandro Di Federico, Michele Beretta, and Giovanni Agosta. Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of openrisc cores. In *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 25–30, Oct 2015. doi: 10.1109/VLSI-SoC.2015.7314386.

[41] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. Piperench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[42] Lance Greggain. Cost benefit tradeoffs for asic versus programmable logic device. In *ASIC Seminar and Exhibit, 1990. Proceedings., Third Annual IEEE*, pages P1–5. IEEE, 1990.

[43] M. Hamzeh, A. Shrivastava, and S. Vrudhula. Epimap: Using epimorphism to map applications on cgras. In *DAC Design Automation Conference 2012*, pages 1280–1287, June 2012. doi: 10.1145/2228360.2228600.

[44] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Epimap: Using epimorphism to map applications on cgras. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1280–1287. IEEE, 2012.

[45] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Proceedings of the 50th Annual Design Automation Conference*, page 18. ACM, 2013.

[46] K. Han, J. K. Paek, and K. Choi. Acceleration of control flow on cgra using advanced predicated execution. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 429–432, Dec 2010. doi: 10.1109/FPT.2010.5681452.

[47] K. Han, S. Park, and K. Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1367–1372, March 2012. doi: 10.1109/DATE.2012.6176704.

[48] Kyuseung Han, Junwhan Ahn, and Kiyoung Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Trans. Archit. Code Optim.*, 10(2):8:1–8:25, May 2013. ISSN 1544-3566. doi: 10.1145/2459316.2459319. URL http://doi.acm.org/10.1145/2459316.2459319.

[49] Frank Hannig, Holger Ruckdeschel, and Jürgen Teich. The paula language for designing multi-dimensional dataflow-intensive applications. In *MBMV*, pages 129–138, 2008.

[50] Reiner W Hartenstein and Rainer Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, pages 479–484. IEEE, 1995.

[51] Texas Instruments. Tms320c64x/c64x+ dsp cpu and instruction set reference guide. *Texas Instruments, User manual SPRU732C*, 2005.

[52] M. Karunaratne, A. K. Mohite, T. Mitra, and L. S. Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017. doi: 10.1145/3061639.3062262.

[53] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):22, 2014.

[54] Yoonjin Kim, Mary Kiemb, Chulsoo Park, Jinyong Jung, and Kiyoung Choi. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Design, Automation and Test in Europe*, pages 12–17. IEEE, 2005.

[55] Yoonjin Kim, Rabi N Mahapatra, Ilhyun Park, and Kiyoung Choi. Low power reconfiguration technique for coarse-grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(5):593–603, May 2009. ISSN 1063-8210. doi: 10.1109/TVLSI.2008.2006039.

[56] Dmitrij Kissler, Andreas Strawetz, Frank Hannig, and Jürgen Teich. Power-efficient reconfiguration control in coarse-grained dynamically reconfigurable architectures. *Journal of Low Power Electronics*, 5(1):96–105, 2009.

[57] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the ibm poweren processor: Architecture and performance. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 389–400. ACM, 2012.

[58] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM. ISBN 1-59593-292-5. doi: 10.1145/1117201.1117205. URL http://doi.acm.org/10.1145/1117201.1117205.

[59] Damjan Lampret, Chen-Min Chen, Marko Mlinar, Johan Rydberg, Matan Ziv-Av, Chris Ziomkowski, Greg McGary, Bob Gardner, Rohit Mathur, and Maria Bolado. Openrisc 1000 architecture manual. *Description of assembler mnemonics and other for OR1200*, 2003.

[60] Ronald Laufer, R Reed Taylor, and Herman Schmit. Pci-piperench and the swordapi: A system for stream-based reconfigurable computing. In *Field-Programmable Custom Computing Machines, 1999. FCCM'99. Proceedings. Seventh Annual IEEE Symposium on*, pages 200–208. IEEE, 1999.

[61] Ganghee Lee, Kiyoung Choi, and Nikil D Dutt. Mapping multi-domain applications onto coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(5):637–650, 2011.

[62] J. Lee, S. Seo, H. Lee, and H. U. Sim. Flattening-based mapping of imperfect loop nests for cgras? In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Oct 2014. doi: 10.1145/2656075.2656085.

[63] Giorgio Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341–352, 1973.

[64] Cao Liang and Xinming Huang. Smartcell: A power-efficient reconfigurable architecture for data streaming applications. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 257–262. IEEE, 2008.

[65] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N Do, and Deming Chen. High-level synthesis: productivity, performance, and software constraints. *Journal of Electrical and Computer Engineering*, 2012:1, 2012.

[66] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. Automatic nested loop acceleration on fpgas using soft CGRA overlay. *CoRR*, abs/1509.00042, 2015. URL http://arxiv.org/abs/1509.00042.

[67] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Polyhedral model based mapping optimization of loop nests for cgras. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–8. IEEE, 2013.

[68] Leibo Liu, Junbin Wang, Jianfeng Zhu, Chenchen Deng, Shouyi Yin, and Shaojun Wei. Tlia: Efficient reconfigurable architecture for control-intensive kernels with triggered-long-instructions.

[69] Igor Loi, Davide Rossi, Germain Haugou, Michael Gautschi, and Luca Benini. Exploring multi-banked shared-l1 program cache on ultra-low power, tightly coupled processor clusters. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 64:1–64:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3358-0. doi: 10.1145/2742854.2747288. URL http://doi.acm.org/10.1145/2742854.2747288.

[70] J. Lopes, D. Sousa, and J. C. Ferreira. Evaluation of cgra architecture for real-time processing of biological signals on wearable devices. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7, Dec 2017. doi: 10.1109/RECONFIG.2017.8279789.

[71] K. T. Madhu, S. Das, N. S., S. K. Nandy, and R. Narayan. Compiling hpc kernels for the redefine cgra. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 405–410, Aug 2015. doi: 10.1109/HPCC-CSS-ICESS.2015.139.

[72] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143. ACM, 1999.

[73] Koichiro Masuyama, Yu Fujita, Hayate Okuhara, and Hideharu Amano. A 297mops/0.4 mw ultra low power coarse-grained reconfigurable accelerator cma-sotb-2. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[74] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*, pages 166–173. IEEE, 2002.

[75] Timothy N Miller, Xiang Pan, Renji Thomas, Naser Sedaghati, and Radu Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[76] Ethan Mirsky, Andre DeHon, et al. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, volume 96, pages 17–19, 1996.

[77] Claudio Mucci, Luca Vanzolini, Antonio Deledda, Fabio Campi, and Gerard Gaillat. Intelligent cameras and embedded reconfigurable computing: a case-study on motion detection. In *System-on-Chip, 2007 International Symposium on*, pages 1–4, Nov 2007. doi: 10.1109/ISSOC.2007.4427440.

[78] Stephen Neuendorffer and Fernando Martinez-Vallina. Building zynq® accelerators with vivado® high level synthesis. In *FPGA*, pages 1–2, 2013.

[79] Tegra NVIDIA. K1: A new era in mobile computing. *Nvidia, Corp., White Paper*, 2014.

[80] Nobuaki Ozaki, Y Yoshihiro, Yoshiki Saito, Daisuke Ikebuchi, Masayuki Kimura, Hideharu Amano, Hiroshi Nakamura, Kimiyoshi Usami, Mitaro Namiki, and Masaaki Kondo. Cool mega-array: A highly energy efficient reconfigurable accelerator. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8, Dec 2011. doi: 10.1109/FPT.2011.6132668.

[81] Hyunchul Park, Kevin Fan, Scott A Mahlke, Taewook Oh, Heeseok Kim, and Hongseok Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[82] Kunjan Patel and Chris J Bleakley. Coarse grained reconfigurable array based architecture for low power real-time seizure detection. *Journal of Signal Processing Systems*, 82(1):55–68, 2016.

[83] Thomas Peyret, Gwenolé Corre, Mathieu Thevenin, Kevin Martin, and Philippe Coussy. Efficient application mapping on cgras based on backward simultaneous scheduling/binding and dynamic graph transformations. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 169–172. IEEE, 2014.

[84] Erwan Raffin, Ch Wolinski, François Charot, Krzysztof Kuchcinski, Stéphane Guyetant, Stéphane Chevobbe, and Emmanuel Casseau. Scheduling, binding and routing system for a run-time reconfigurable operator based multimedia architecture. In *Design and Architectures for Signal and Image Processing (DASIP), 2010 Conference on*, pages 168–175. IEEE, 2010.

[85] Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, and Luca Benini. A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[86] Zoltán Endre Rákossy, Dominik Stengele, Gerd Ascheid, Rainer Leupers, and Anupam Chattopadhyay. Exploiting scalable cgra mapping of lu for energy efficiency using the layers architecture. In *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*, pages 337–342. IEEE, 2015.

[87] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[88] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Teman, J. Constantin, A. Burg, I. Miro-Panades, E. Beignè, F. Clermidy, P. Flatresse, and L. Benini. Energy-efficient near-threshold parallel computing: The pulpv2 cluster. *IEEE Micro*, 37(5): 20–31, September 2017. ISSN 0272-1732. doi: 10.1109/MM.2017.3711645.

[89] Davide Rossi, Fabio Campi, Antonello Deledda, Simone Spolzino, and Stefano Pucillo. A heterogeneous digital signal processor implementation for dynamically reconfigurable computing. In *2009 IEEE Custom Integrated Circuits Conference*, pages 641–644, Sept 2009. doi: 10.1109/CICC.2009.5280747.

[90] Davide Rossi, Igor Loi, Germain Haugou, and Luca Benini. Ultra-low-latency lightweight dma for tightly coupled multi-core clusters. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 15. ACM, 2014.

[91] Davide Rossi, Antonio Pullini, Igor Loi, Michael Gautschi, Frank K. Gürkaynak, Andrea Bartolini, Philippe Flatresse, and Luca Benini. A 60 GOPS/W, -1.8 V to 0.9 V body bias ULP cluster in 28 nm UTBB fd-soi technology. *Solid-State Electronics*, 117: 170 – 184, 2016. ISSN 0038-1101. doi: http://dx.doi.org/10.1016/j.sse.2015.11.015. URL //www.sciencedirect.com/science/article/pii/S0038110115003342.

[92] Davide Rossi, Antonio Pullini, Igor Loi, Michael Gautschi, Frank K Gürkaynak, Andrea Bartolini, Philippe Flatresse, and Luca Benini. A 60 gops/w,- 1.8 v to 0.9 v body bias ulp cluster in 28nm utbb fd-soi technology. *Solid-State Electronics*, 117: 170–184, 2016.

[93] Yoshiki Saito, Toru Sano, Masaru Kato, Vasutan Tunbunheng, Yoshihiro Yasuda, Masayuki Kimura, and Hideharu Amano. Muccra-3: a low power dynamically reconfigurable processor array. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pages 377–378. IEEE Press, 2010.

[94] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[95] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 422–433. ACM, 2003.

[96] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W Keckler, Robert G McDonald, and Charles R Moore. Trips: A polymorphous architecture for exploiting

ilp, tlp, and dlp. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1 (1):62–93, 2004.

[97] Charles Selvidge, Anant Agarwal, Matt Dahl, and Jonathan Babb. Tiers: Topology independent pipelined routing and scheduling for virtualwireŹ compilation. In *Field-Programmable Gate Arrays, 1995. FPGA'95. Proceedings of the Third International ACM Symposium on*, pages 25–31. IEEE, 1995.

[98] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49 (5):465–481, May 2000. ISSN 0018-9340. doi: 10.1109/12.859540.

[99] S4 Snapdragon. Processors: System on chip solutions for a new mobile age. *White paper, ARM*, 2011.

[100] Greg Stitt. Are field-programmable gate arrays ready for the mainstream? *IEEE Micro*, 31(6):58–63, 2011.

[101] Honlian Su, Yu Fujita, and Hideharu Amano. Body bias control for a coarse grained reconfigurable accelerator implemented with silicon on thin box technology. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2014. doi: 10.1109/FPL.2014.6927486.

[102] Alexandru Tanase, Michael Witterauf, Ericles Sousa, Vahid Lari, Frank Hannig, and Jürgen Teich. Loopinvader: A compiler for tightly coupled processor arrays.

[103] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 170–171 vol.1, Feb 2003. doi: 10.1109/ISSCC.2003.1234253.

[104] Michael B Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1131–1136. IEEE, 2012.

[105] Brian C Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Energy-efficient specialization of functional units in a coarse-grained reconfigurable array. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 107–110. ACM, 2011.

[106] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

[107] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 225–235. ACM, 2000.

[108] Zhiyi Yu, Michael J Meeuwsen, Ryan W Apperson, Omar Sattari, Michael Lai, Jeremy W Webb, Eric W Work, Dean Truong, Tinoosh Mohsenin, and Bevan M Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits*, 43(3):695–705, 2008.

UNIVERSITE
BRETAGNE
LOIRE / MATHSTIC

Université
Bretagne Sud

Titre : Architecture et modèle de programmation pour accélérateurs reconfigurables dans les systèmes embarqués multi-cœurs.

Mots clés : CGRA, accélérateur matériel, low power, compilation,

Résumé : La complexité des systèmes embarqués et des applications impose des besoins croissants en puissance de calcul et de consommation énergétique. Couplé au rendement en baisse de la technologie, le monde académique et industriel est toujours en quête d'accélérateurs matériels efficaces en énergie. L'inconvénient d'un accélérateur matériel est qu'il est non programmable, le rendant ainsi dédié à une fonction particulière. La multiplication des accélérateurs dédiés dans les systèmes sur puce conduit à une faible efficacité en surface et pose des problèmes de passage à l'échelle et d'interconnexion. Les accélérateurs programmables fournissent le bon compromis efficacité et flexibilité. Les architectures reconfigurables à gros grains (CGRA) sont composées d'éléments de calcul au niveau mot et constituent un choix prometteur d'accélérateurs programmables.

Cette thèse propose d'exploiter le potentiel des architectures reconfigurables à gros grains et de pousser le matériel aux limites énergétiques dans un flot de conception complet.

Les contributions de cette thèse sont une architecture de type CGRA, appelé IPA pour Integrated Programmable Array, sa mise en œuvre et son intégration dans un système sur puce, avec le flot de compilation associé qui permet d'exploiter les caractéristiques uniques du nouveau composant, notamment sa capacité à supporter du flot de contrôle. L'efficacité de l'approche est éprouvée à travers le déploiement de plusieurs applications de traitement intensif. L'accélérateur proposé est enfin intégré à PULP, a Parallel Ultra-Low-Power Processing-Platform, pour explorer le bénéfice de ce genre de plate-forme hétérogène ultra basse consommation.

Title : Architecture and Programming Model Support For Reconfigurable Accelerators in Multi-Core Embedded Systems

Keywords : CGRA, hardware accelerator, low-power, compilation

Abstract: Emerging trends in embedded systems and applications need high throughput and low power consumption. Due to the increasing demand for low power computing and diminishing returns from technology scaling, industry and academia are turning with renewed interest toward energy efficient hardware accelerators. The main drawback of hardware accelerators is that they are not programmable. Therefore, their utilization can be low as they perform one specific function and increasing the number of the accelerators in a system on chip (SoC) causes scalability issues. Programmable accelerators provide flexibility and solve the scalability issues. Coarse-Grained Reconfigurable Array (CGRA) architecture consisting of several processing elements with word level granularity is a promising choice for programmable accelerator.

Inspired by the promising characteristics of programmable accelerators, potentials of CGRAs in near threshold computing platforms are studied and an end-to-end CGRA research framework is developed in this thesis. The major contributions of this framework are: CGRA design, implementation, integration in a computing system, and compilation for CGRA. First, the design and implementation of a CGRA named Integrated Programmable Array (IPA) is presented. Next, the problem of mapping applications with control and data flow onto CGRA is formulated. From this formulation, several efficient algorithms are developed using internal resources of a CGRA, with a vision for low power acceleration. The algorithms are integrated into an automated compilation flow. Finally, the IPA accelerator is augmented in PULP - a Parallel Ultra-Low-Power Processing-Platform to explore heterogeneous computing.