AN OPEN-SOURCE FRAMEWORK FOR THE GENERATION OF RISC-V
PROCESSOR + CGRA ACCELERATOR SYSTEMS

by

Xiaoyi Ling

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical & Computer
Engineering
University of Toronto

© Copyright 2021 by Xiaoyi Ling

Abstract

An Open-Source Framework for the Generation of RISC-V Processor + CGRA
Accelerator Systems

Xiaoyi Ling

Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical & Computer Engineering.

University of Toronto

2021

Course-Grained Reconfigurable Architectures (CGRAs) are programmable hardware devices containing coarse-grained processing elements that can perform ALU-like functions and having word-wide programmable interconnect. CGRAs are promising platforms for the implementation of domain-specific accelerators. In this thesis, we present a framework for building hybrid processor/accelerator systems to realize compute-kernel acceleration. Each system comprises a RISC-V processor and a CGRA. The proposed framework can generate the system automatically integrating CGRAs generated by the open-source CGRA-ME tool [4], and the RISC-V processor from the PULP project [8]. In an experimental study, several RISC-V+CGRA systems are generated, and we observe an order-of-magnitude speedup vs. software and considerable speedup vs. a vector processor on four. With the hybrid system generation capabilities integrated into the CGRA-ME framework, it now permits hybrid system generation with different types of CGRA architectures, enabling a variety of new research on processor+accelerator architectures and programming models.

Acknowledgements

I would like to express my deep and sincere gratitude to my research supervisor, Professor Jason Anderson, for giving me the opportunity to do research and providing invaluable guidance throughout the project. Without your sincere and selfless support, prompt and useful advice, I would never be where I am now. You have taught me the methodology to carry out the research and to present the work precisely and clearly. It was a great honor and privilege and work with you. I would also like to thank you for your friendship, thoughtfulness, and great sense of humor. Thank you for all that you have done.

I would also like to extend my great thanks to people I have worked with during my research – Taka, Vimal, Jin Hee, Julie, Rami, Omar, Linda, Gabriel and Austin. I have learned so much from you all on critical thinking and deep insight. I could always get inspired from your advice and I have always enjoyed talking with you all. A special thanks to Nicholas Giamblanco, who recommended me to Jason's group for Meng project. Thanks for making my research experience possible.

I am extremely thankful to all the friends and classmates I have met throughout my time here at the University of Toronto, for adding so much fun in my research life. Special thanks to my roommates, Manling and Qiutong, who were always there when I needed help. Thank you Qiutong for always cooking for me so that I did not starve to death and thank you Manling for always washing the dishes so that I did not need to do it. My life could not be as bright without you.

Finally, I want to sincerely thank my family for their support and encouragement. Thank you for always be confident on me and believe that I can do anything. I love you, mom, dad and dear grandma.

Contents

List of Tables	vi
List of Figures	vii
List of Abbreviations	ix
1 Introduction	1
1.1 Contributions	4
1.2 Organiztion	4
2 Background	6
2.1 Coarse-Grained Reconfigurable Arrays	6
2.2 CGRA-ME (Modelling and Exploration) Framework	7
2.3 CGRA Architectures Used in this Research	8
2.4 RISC-V	11
2.4.1 RISC-V Instruction Set Architecture	11
2.4.2 RISC-V Processor	12
2.5 Processor+CGRA Accelerator Coupling	14
2.6 Related Work	15
2.7 Summary	16
3 RISC-V+CGRA Systems: Design and Automated Generation	18

3.1	Hybrid System Architecture	19
3.2	CGRA Control	23
3.2.1	Counter Control	24
3.2.2	Data Control	25
3.3	Automated Hybrid System Generation	26
3.4	Summary	28
4	Case Study	29
4.1	Data-Based Control	35
4.2	Counter Control	35
4.3	Summary	36
5	Experimental Study and Results	37
5.1	Setup	37
5.1.1	Applications	37
5.2	Results	41
5.2.1	RISC-V+ADRES CGRA Hybrid System vs. RISC-V Software Alone	41
5.2.2	Area and Critical-Path Analysis	45
5.2.3	Comparison with a RISC-V Vector Processor	47
5.2.4	RISC-V+HYCUBE CGRA Hybrid System vs. RISC-V Software Alone	50
5.3	Summary	51
6	Conclusion and Future Work	52
A	Application DFGs	54
	Bibliography	60

List of Tables

2.1	A listing of standard RV32I instructions	12
3.1	List of memory-mapped registers and their function in the control unit for an ADRES 4×4 CGRA with the hybrid system having 22 bits of addressable memory.	20
3.2	Example of memory banks for ADRES 4×4 CGRA and 4MB of memory (22-bits of address), connections to the RISC-V processor and CGRA accelerator.	23
5.1	Cycle latency of benchmarks executing: 1) in software on RISC-V, 2) on two types of hybrid RISC-V+CGRA systems, 3) on a RISC-V processor with vector extensions.	42
5.2	Cycle decomposition of <i>MM</i> application on hybrid CGRA-based systems.	44
5.3	Cycle decomposition of <i>MM</i> application in RISC-V software.	44
5.4	Timing analysis. Timing results for RISC-V and CGRAs are post-layout; timing results for the vector units are post-synthesis.	45
5.5	Bypass delays (<i>ns</i>)	47
5.6	Bypass counts and critical path delays on ADRES CGRA.	47
5.7	Cycle latency of benchmarks on hybrid system with 4×4 HyCUBE CGRA.	50

List of Figures

2.1	Performance vs. flexibility tradeoff for hardware devices.	7
2.2	CGRA-ME framework flow.	9
2.3	4×4 ADRES CGRA architecture.	10
2.4	4×4 HyCUBE CGRA architecture.	10
2.5	Block diagram for CV32E40 RISC-V processor.	14
2.6	CGRA coupling methods	15
3.1	Block diagram of the hybrid RISC-V + CGRA system.	20
3.2	Hybrid system generation flow.	27
4.1	DFG code of FIR for 4×4 CGRA using data control.	30
4.2	DFG graph of FIR for 4×4 CGRA using data control.	31
5.1	Cycle count speedups of benchmarks compare to software on RISC-V with 1) three types of hybrid RISC-V+CGRA systems, 2) on a RISC-V proces- sor with vector extensions.	43
5.2	Area analysis. Results for RISC-V and CGRAs are post-layout; results for the vector units are post-synthesis.	46
A.1	DFG for FIR on 4×4 CGRA with counter control	54
A.2	DFG for FIR on 8×8 CGRA with counter control	55
A.3	DFG for KNN on 4×4 CGRA with counter control	55
A.4	DFG for KNN on 8×8 CGRA with counter control	56

A.5	DFG for MM on 4×4 CGRA with counter control	56
A.6	DFG for MM on 8×8 CGRA with counter control	57
A.7	DFG for POLY on 4×4 CGRA with counter control	57
A.8	DFG for POLY on 8×8 CGRA with counter control	58
A.9	DFG for FIR on 4×4 CGRA with data control	58
A.10	DFG for FIR on 8×8 CGRA with data control	59

List of Abbreviations

ADRES Architecture for Dynamically Reconfigurable Embedded Systems. 3, 4, 8, 9, 11, 18, 19, 22, 23, 29, 35, 36, 38, 46, 50

ALU Arithmetic Logic Unit. 3, 8, 14, 45, 49

API Application Programming Interface. 7

ASIC Application-Specific Integrated Circuit. 3, 6, 7, 14, 47

CAD Computer-Aided Design. 3

CGRA Coarse-Grained Reconfigurable Array. vi–viii, 2–4, 6–11, 13, 14, 16, 18–50, 52–59

CGRA-ME CGRA Modelling and Exploration. 3, 7–9, 16, 18, 23–29, 32, 34, 35, 46, 52

CLB Combinational Logic Block. 2

CPU Central Processing Unit. 6, 13

DFG Dataflow Graph. vii, viii, 7, 8, 23, 24, 26, 29–32, 35, 36, 38, 42, 48, 50, 54–59

DMA Direct Memory Access. 16

DSP Digital Signal Processing. 2

FIR Finite Impulse Response Filter. vii, viii, 29–32, 35–38, 42, 44, 50, 54, 55, 58, 59

FPGA Field-Programmable Gate Array. 2, 3, 6, 7, 12, 14, 32

FU Functional Unit. 15

GNU GNU’s not Unix. 2

GPU Graphics Processing Unit. 6

I/O Input/Output. 2, 3, 18–20, 25, 26, 28, 29, 32, 34, 35, 39, 40, 50, 52

ISA Instruction Set Architecture. 1, 11–13, 16

KNN K Nearest Neighbor. vii, 37, 39, 42, 50, 55, 56

LLVM Low-Level Virtual Machine. 2, 53

LSU Load and Store Unit. 18, 20

LUT Lookup Table. 2

MAC Multiply Accumulate. 38, 39

MLIR Multi-Level Intermediate Representation. 52

MM Matrix Multiply. viii, 37, 42, 50, 56, 57

PE Processing Element. 8–10, 16, 29, 45, 46, 50

PLL Phase-Locked Loop. 2

POLY Polynomial. viii, 40–42, 57, 58

RC Reconfigurable Cell. 16

RF Register File. 15

RISC Reduced Instruction Set Computer. 1, 16

RTL Register-Transfer Level. 3, 7, 8, 26, 38, 48

SIMD Single Instruction Multiple Data. 47

SRAM Static Random-Access Memory. 47

SRP Samsung Reconfigurable Processor. 9

UCB Bar University of California, Berkeley Architecture Research. 13

VHDL VHSIC Hardware Description Language. 13

VLIW Very-Long Instruction Word. 15

XML Extensible Markup Language. 7

Chapter 1

Introduction

Applications such as machine learning and streaming media continue to drive demand for increased computational performance and energy efficiency. However, standard processors are no longer getting faster, with clock frequencies plateauing in the mid-2000s. This trend has led to the modern “accelerator era”, where processors are coupled with custom application-specific accelerators to deliver higher application performance. Moreover, in certain application domains such as machine learning, the underlying algorithms are evolving rapidly. This drives the need for flexible accelerators, whose specific functionality can be tweaked as requirements change.

RISC-V [24] is an instruction set architecture (ISA) established on reduced instruction set computer (RISC) principles. Unlike other proprietary ISA designs, such as those from ARM and Intel, RISC-V is provided under an open-source license, allowing any individual or organization to freely use the RISC-V ISA. RISC-V is a family of ISAs, with the simplest being a 32-bit integer ISA with no multiplication/division instructions (labelled “RV32I”), ranging to more complex 64-bit ISAs with floating-point and atomic capabilities. The RISC-V ISA also supports custom extensions tailored for specific applications. The ISA is thus very flexible and well-suited to a wide range of different application domains, including embedded, low-power, or high-performance.

The software ecosystem for RISC-V is under active development and maturing rapidly. Current software tools include compiler support via the GNU toolchain (GCC), as well as the LLVM toolchain [15], and simulators such as Spike¹. Commercial and open-source hardware implementations of RISC-V have also appeared. SiFive (www.sifive.com), for example, is a start-up offering RISC-V processor IP cores. FPGA (e.g. [21]) and ASIC (e.g. [26]) implementations of RISC-V have been released by several universities. Despite the advantages of RISC-V, it is not immune to the trends surrounding the coming obsolescence of Moore’s Law and Dennard scaling [12]. Microprocessor performance improvements have plateaued, leading the proliferation of application-specific compute accelerators to raise computational performance and energy efficiency.

Reconfigurable architectures are one of the promising solutions for realizing application-specific accelerators. A reconfigurable architecture contains a configurable fabric that combines the flexibility of software and high performance of hardware to perform compute-intensive tasks. The reconfigurability notion implies that the architecture can be reprogrammed to perform some other task after a current task is completed, thereby offering software flexibility with hardware speed.

Two popular styles of reconfigurable architectures are field-programmable gate arrays (FPGAs) and coarse-grained reconfigurable arrays (CGRAs). FPGAs are usually made up of a programmable fabric that consist of configurable logic blocks (CLBs), programmable interconnect, hard IP blocks (such as DSP blocks and PLLs) and I/Os. A CLB typically has one or more K-input lookup tables (K-LUT), each being a small fine-grained memory capable of implementing any K-input logic function. CLBs also contain flip-flops, arithmetic, and other circuitry. The interconnect in FPGAs is configurable at the level of an individual logic signal.

CGRAs, on the other hand, are comprised of an array of large coarse-grained processing element blocks and word-wide configurable interconnect. The processing element

¹<https://github.com/riscv-software-src/riscv-isa-sim>

block typically consists of a functional unit similar to an arithmetic-logic unit (ALU) in a processor that carries out arithmetic and logical operations. Compared to FPGAs, CGRAs are more hardware efficient from the perspectives of area, performance and power, but less programmable and therefore, less flexible.

CGRAs first appeared in the literature in the 1990s (e.g. [9]), and a significant number of CGRA architectures have been presented by both academia (surveyed in [19]), and industry (e.g. [14]). Some representative CGRA architectures include ADRES [22], HyCUBE [13] and Mosaic [30]. Most of the previous CGRA projects describe one specific CGRA architecture and develop Computer-Aided Design (CAD) tools specifically for that CGRA architecture.

CGRA-ME [4] (*CGRA modelling and exploration*) is an open-source software framework for research on CGRA architectures and CAD. CGRA-ME offers capabilities analogous to what VTR [25] and Gem5 [3] provide for FPGAs and processors, respectively. Meaning, CGRA-ME is capable of modeling a variety of hypothetical CGRA architectures described by a human architect, and mapping applications onto them, allowing exploration and comparison of CGRA architectures, as well as facilitating new research on CGRA CAD tools.

This thesis centers on integrating an existing open-source RISC-V implementation with CGRA accelerators generated by CGRA-ME, realizing a software framework for push-button generation of RISC-V+CGRA-accelerator systems. The processor and CGRA communicate through memory-mapped I/Os and shared memory. Verilog RTL is automatically generated for the CGRA architecture, memory blocks as well as the communication system between the processor and CGRA, allowing simulation and verification, as well as physical implementation using an ASIC standard-cell methodology or as an FPGA overlay. Unlike recent related work on RISC-V+CGRA systems (e.g. [7]), our generation framework is not tied to the use of a specific CGRA. Rather, *any* CGRA that can be modelled by the CGRA-ME framework can be used. We believe the framework

will be broadly useful for research on processor+CGRA architectures, domain-specific compute acceleration and programming methodologies.

1.1 Contributions

The contributions of this thesis are as follows:

- A software framework that offers push-button automated generation of RISC-V+CGRA hybrid systems.
- A programming methodology for the generated hybrid systems.
- An experimental study using the ADRES CGRA [22] as a representative example, with the RISC-V+CGRA mapped into 45nm standard cells [29]. The study includes 4 applications mapped onto CGRAs of different sizes. We compare the performance and area of the generated hybrid systems with: 1) software running on the RISC-V, and 2) a RISC-V with vector extensions. Results show the RISC-V+CGRA provides an order-of-magnitude speedup vs. software alone, and $\sim 2\times$ speedup vs. the vector processor.
- A demonstration of the framework's flexibility and genericity by interchanging the ADRES CGRA with the HyCUBE CGRA architecture [13] and evaluating the performance.

A portion of this work has been published at the 2021 IEEE Euromicro Conference on Digital System Design (DSD) [17].

1.2 Organiztion

The thesis is organized as follows: Chapter 2 introduces relevant background material and describes related work. Chapter 3 discusses the hybrid system model assumed in this

research and overviews automated hybrid system generation. Chapter 4 includes a case study for using and programming the hybrid system. Chapter 5 presents an experimental study, results and discussion. Chapter 6 concludes and offers avenues for future work.

Chapter 2

Background

2.1 Coarse-Grained Reconfigurable Arrays

Coarse-Grained Reconfigurable Arrays (CGRAs) are programmable hardware devices situated between custom Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) when considering efficiency and flexibility, as shown in Figure 2.1. ASICs are designed for a specific application to deliver high performance and energy efficiency; they can not be configured for other applications. FPGAs, on the other hand, can be reconfigured for many applications at the cost of reduced performance. GPUs and CPUs are more generic with respect to computing, with commensurate performance and energy efficiency reductions. CGRAs, compared to other hardware devices, endeavour to achieve a good balance between performance and programmability, which proves to be attractive for hardware accelerators. They are especially good at performing loop-heavy tasks and gain benefits from leveraging spatial, pipeline and memory-access parallelism. Another advantage of CGRAs is that they usually have rapid compile times compared to FPGAs because of their coarse-grained nature.

Figure 2.1: Performance vs. flexibility tradeoff for hardware devices.

2.2 CGRA-ME (Modelling and Exploration) Framework

CGRA-ME is an architectural modelling and exploration (ME) framework. It is currently under active development at the Department of Electrical and Computer Engineering, University of Toronto. With CGRA-ME, an architect can describe the CGRA they wish to model in an XML-based language or C++ API. The XML-based model or C++ API are parsed by CGRA-ME and transformed into a graph-based, in-memory representation of the CGRA device. Compute kernels (application benchmarks) are written in C and are transformed into Dataflow Graphs (DFGs), using the LLVM compiler. DFGs can then be mapped onto the CGRA and a configuration bitstream is automatically generated.

Using the CGRA-ME framework, one can map a basket of applications drawn from a domain of interest to a variety of CGRAs. The framework also provides automatic generation of Verilog RTL for the modeled architecture. This allows the architect to simulate for verification purposes, and perform synthesis to either an ASIC or FPGA-overlay implementation to assess speed, area, and power consumption. Prior to this work, the CGRA-ME framework allowed one to model and evaluate CGRAs in isolation,

i.e. CGRAs not connected to any other hardware models. The main focus of this thesis is to extend the CGRA-ME framework by integrating a RISC-V processor connected to the modelled CGRA, and to allow such hybrid systems to be automatically generated, simulated, evaluated, and physically implemented.

Figure 2.2 shows the aspects of the CGRA-ME flow leveraged in this research. The blue boxes show the inputs to the framework: 1) a specification of a CGRA architecture, and 2) an application DFG. The DFG represents a compute-intensive kernel (e.g., a compute-intensive loop body) destined for CGRA acceleration. The architecture specification is interpreted to yield a CGRA device model – the central green block in Figure 2.2. The mapper in CGRA-ME is architecture agnostic [31]: it accepts the device model as input, as well as the application DFG, producing mapping statistics and a configuration bitstream for the modelled CGRA. The in-memory device model can also be fed to a Verilog RTL generator. The red boxes in Figure 2.2 represent commercial EDA tools. We use ModelSim for RTL simulation to verify functionality and acquire cycle-count data. We use Synopsys Design Compiler and Cadence Innovus to generate a standard-cell implementation, to assess post-layout area and critical-path delay.

2.3 CGRA Architectures Used in this Research

While the CGRA-ME framework is generic in the sense that it is not tied to a single CGRA architecture, we selected two CGRAs to be used as test vehicles for demonstrating the framework and for the experimental study. To this end, we first opted to use the ADRES CGRA architecture [22], as it is one of the most highly cited, and it had already been modelled within CGRA-ME. Figure 2.3 shows a 4×4 ADRES CGRA. There are 4 rows and 4 columns of processing elements (PEs). Each PE contains an ALU capable of: add, subtract, multiply, shift, and logical operations. PEs in the top row of the CGRA share a register file (DRF); PEs in other rows each have their own local register

Figure 2.2: CGRA-ME framework flow.

file (LRF). PEs in the top row also have access to I/O ports. Each row of PEs has access to a memory port, shown on the left side of the figure.

Regarding interconnect, ADRES PEs connect to their 4 neighbours, as well as diagonal neighbours (not shown in Figure 2.3 for clarity). Toroidal connections “wrap-around” from the top/bottom (left/right) of each column (row). The number of I/O ports is determined the number of columns of the CGRA architecture, and the number of memory ports depends on the number of rows of the CGRA architecture. In our experiments, we use 4×4 and 8×8 ADRES CGRAs, which provide 4 and 8 sets of I/O ports and memory ports, respectively. Notably, ADRES has been commercialized by Samsung as the Samsung Reconfigurable Processor (SRP) [14].

In addition to ADRES, we conduct experiments with a modified version of the HyCUBE CGRA [13] (also modelled in CGRA-ME [5]) to verify the genericity of our frame-

Figure 2.3: 4×4 ADRES CGRA architecture. Figure 2.4: 4×4 HyCUBE CGRA architecture.

work. As illustrated in Figure 2.4, HyCUBE has a considerably different PE structure. It contains a crossbar switch with 6 inputs that come from the four neighbouring PEs,

and from the ALU’s output in combinational/registered form and 6 outputs to the same neighboring PEs and to the two ALU operands. Multiple paths through the crossbar can be used concurrently, increasing routing flexibility vs. ADRES. For 4×4 CGRA, HyCUBE has 8 memory ports in total on left and right, and 8 I/O ports on the top and bottom, respectively. Compared to 4×4 ADRES, the increased number of memory ports will demonstrate the impact of available ports in CGRA application mapping.

2.4 RISC-V

2.4.1 RISC-V Instruction Set Architecture

The RISC-V ISA was originally developed at the University of California, Berkeley, and is now maintained by the RISC-V foundation¹. RISC-V has a modular and extensible ISA design with groups of instructions ranging from a base integer set (“I”) which includes operations such as addition, subtraction, to standard extension sets of multiplication and division (“M”), atomic instruction (“A”), single-precision floating-point (“F”), double-precision floating-point (“D”), and so on. Apart from these set designations, the RISC-V specification also defines 32, 64 and 128-bit variants. The number of bits refers to the width of the address bus. A RV32I processor, for example, would be 32-bit supporting solely the base integer instruction set. An RV64IFA processor would be 64-bit, with support for integer, floating point, and atomic instructions.

We use the RV32I instruction set in this research. This ISA has 32 registers, of which 31 are general purpose, and 1 is constant 0. The ISA includes arithmetic instructions that operate on registers, load and store instructions to access memory, and branch instructions. To give the reader a sense of the types of instructions in the RISC-V ISA, Table 2.1 shows some representative instructions and their usage from arithmetic, load/store and control flow instruction categories of the 32-bit base Integer ISA.

¹<https://riscv.org/>

Table 2.1: A listing of standard RV32I instructions

	Example Instruction	Instruction Name	Meaning
Arithmetic	add x1, x2, x3	Add	$Regs[x1] \leftarrow Regs[x2] + Regs[x3]$
	sll x1, x2, 5	Shift Left Logical	$Regs[x1] \leftarrow Regs[x2] \ll 5$
	slt	Set Less Than	$if(Regs[x2] < Regs[x3])$ $Regs[x1] \leftarrow 1$ $else Regs[x1] \leftarrow 0$
Load/Store	lw x1, 80(x2)	Load Word	$Regs[x1] \leftarrow Mem[80 + Regs[x2]]$
	sw x3, 500(x4)	Store Word	$Mem[500 + Regs[x4]] \leftarrow Regs[x3]$
	sb x2, 41(x3)	Store Byte	$Mem[41 + Regs[x3]] \leftarrow Regs[x2][7 : 0]$
Control Flow	jal x1, offset	Jump and Link	$Regs[x1] \leftarrow PC + 4;$ $PC \leftarrow PC + (offset \ll 1)$
	jalr x1, x2, offset	Jump and Link Register	$Regs[x1] \leftarrow PC + 4;$ $PC \leftarrow PC + (offset \ll 1)$
	beq x3, x4, offset	Branch Equal Zero	$if(Regs[x3] == Regs[x4])$ $PC \leftarrow PC + (offset \ll 1)$

The open-source aspect and the flexibility are attractive attributes of RISC-V, and many hardware platforms are being developed based on it. For example, PolarFire SoC FPGAs from Microchip contain a hardened multicore RISC-V processor on the FPGA die [1].

2.4.2 RISC-V Processor

We evaluated a number of open-source RISC-V ISA hardware implementations, and selected the CV32E40P processor [8], developed as part of the PULP platform [27]² from ETH Zurich and the University of Bologna. The processor is a 32-bit in-order RISC-V core with a 4-stage pipeline with no memory attached to the core. The core is able to implement the RV32IMC (32-bit integer RISC-V with multiply/divide and compressed extension) ISA with optional floating-point extensions. It also supports several optional Xpulp custom extensions, for tasks such as bit manipulations and other ALU operations to achieve higher code density, performance, and energy efficiency. Considering that

²<https://pulp-platform.org/>

heavy computations are intended to be offloaded to the CGRA instead of being handled by the processor, we removed the floating-point, instruction-compression, and the custom Xpulp features for this work, thereby altering the processor to implement the RV32IM ISA.

To find a suitable processor for our hybrid system, we also explored other candidates. The Rocket Chip Generator [2] is an open-source System-on-Chip design generator that provides an in-order RISC-V core generator (Rocket) and an out-of-order core generator (BOOM). It is under development by SiFive and Berkeley Architecture Research (UCB Bar) with a complete testing environment and software tools. These RISC-V generators produce a processor in the Chisel hardware description language, developed by UCB Bar, instead of typical Verilog or VHDL. As such, we opted not to use these as Chisel is not yet widely supported by industrial CAD tools.

Other open-source options include Hummingbirdv2 E203³ by Nuclei System Technology with a 2-stage pipeline that implements the RV32EMAC ISA, and biRISC-V⁴ by ultraEmbedded with a 32-bit dual-issue RISC-V CPU that supports RV32IM. They are pre-designed processors written in Verilog. We opted not to use them for our hybrid system because their architectures are too complex for a control processor and they support many RISC-V extensions that are not used in our project. We finally chose the CV32E40P processor for several reasons. First, it is open-source and has been used in a wide range of projects (e.g. [26]). Second, it has strong documentation and provides a complete verification environment. Also the System Verilog code that defines the processor was clear and highly parameterized, making it easy to modify depending on one's needs.

Figure 2.5 gives a high-level block diagram of the CV32E40 processor. The pipeline is quite traditional: the first stage is instruction fetch; the second is decode; third contains the ALUs; and, the forth is for write-back to memory.

³https://github.com/riscv-mcu/e203_hbirdv2

⁴<https://github.com/ultraembedded/biriscv>

Figure 2.5: Block diagram for CV32E40 RISC-V processor.

2.5 Processor+CGRA Accelerator Coupling

Hardware accelerators typically have better computational throughput and consumes less power compared to software running on a processor. Hardware accelerators may include specialized processors such as vector processors, FPGAs, ASICs, CGRAs, or GPUs. A hardware accelerator can be connected to the processor in various ways, some characterized as *tightly coupled* and others *loosely coupled*, shown in Figure 2.6.

In the tightly coupled approach, the CGRA may for example have direct access to the processor’s register file or its internal pipeline and/or ALU registers. This is illustrated abstractly for a generic accelerator in Figure 2.6(a). Examples of this include DySER [10] and DynaSpAM [18], which integrate a CGRA into the backend of a processor pipeline, using the CGRA to execute ISA extensions.

While a tightly coupled approach may offer advantages in terms of requiring fewer explicit data transfers between the processor and CGRA, we believe it would be challenging to make generic and automated in terms of the range of CGRA architectures that could be used with the processor in an auto-generated fashion. Moreover, in the

(a) Tightly coupled CGRA model

(b) Loosely coupled CGRA model

Figure 2.6: CGRA coupling methods

loosely coupled approach, the CGRA and processor can execute simultaneously and independently, and the CGRA can have a different memory interface than the processor, such as the use of multiple memory ports. We therefore opted for a more loosely coupled approach, where the CGRA is kept separate from the processor, illustrated abstractly in Figure 2.6. Observe that the CGRA does not have access to the processor’s register file, nor is it closely integrated with the processor’s datapath.

2.6 Related Work

The ADRES [22] system tightly couples a Very Long Instruction Word (VLIW) processor to a CGRA. The processor and the CGRA are tightly coupled through the processor’s functional units (FUs) and register files (RF). In VLIW mode, one row of the CGRA’s FUs are used to realize the ALUs in vector lanes. In CGRA mode, the entire 2D CGRA

is available, with nearest-neighbour connectivity between the PEs.

Morphosys [28] is a typical loosely coupled hybrid system targeting media applications. It consists of a RISC processor and an 8×8 reconfigurable array with identical processing elements (PEs) called reconfigurable cell(RC). Each RC is configured by 32-bit context words stored in a configuration memory. The CGRA system also has a direct memory access (DMA) controller to transfer data between the processor’s main memory and a local CGRA storage buffer. The processor includes custom instructions in its ISA for managing the CGRA control.

There are also related work that combine RISC-V cores with CGRAs. Fiolhais et al. [7] describe a hybrid system with multiple RISC-V and CGRA cores. Two types of RISC-V are used, an area-optimized variant, and a pipelined variant with higher performance. A specific type of CGRA, called Versat [20], is used in the design. Versat targets kernels like transforms and filters and does not require large CGRA arrays, so it instead uses a linear-style CGRA array. Das et al. [6] describe a RISC-V+CGRA system for sensor data analysis and they compare their system with a multi-core processor system.

In relation to the above, our work offers a generic framework for RISC-V+CGRA system generation, that is not tied to a specific CGRA or application domain.

2.7 Summary

In this Chapter, we introduced CGRAs, and the CGRA-ME framework developed for modelling and exploring CGRA architectures, with a specific emphasis on aspects of CGRA-ME used in this research. We reviewed RISC-V ISA and several open-source RISC-V implementations, including the one selected for this research. We also discussed methods typically used for coupling a processor with an accelerator, and explained why we opted to use a loosely-coupled method in our work. Finally we covered the related

work in this field.

Chapter 3

RISC-V+CGRA Systems: Design and Automated Generation

In this chapter, we address a key design decision for any hybrid processor/accelerator system is: *how should the processor and accelerator connect to, and communicate with one another?* In addition, we describe how such hybrid systems can be automatically generated.

A processor communicates with memory through its Load and Store Unit (LSU), as shown in Figure 2.5. There are four ports to facilitate this indicating whether: 1) the memory access is a read or a write, 2) the memory access address, the 3) read-data and the 4) write-data, accordingly. The CGRA-ME framework offers two primitives for specifying CGRA external connectivity: I/O ports and memory ports. As shown in Figures 2.3 and 2.4, both ADRES and HyCUBE CGRA architectures have I/O ports and memory ports. An I/O port interface has a 32-bit input port and a 32-bit output port. A memory port interface has four ports that align with those of the processor’s LSU. The hardware design of the connectivity between the host processor and the accelerator is based on using memory-mapped registers that connect to CGRA I/O ports and memory banks attached to the CGRA ’s memory ports. I/O ports are useful for the cases where

input data does not need to be updated every clock cycle. Memory ports, on the other hand, are useful for streaming input and output data.

3.1 Hybrid System Architecture

Figure 3.1 shows a block diagram of the RISC-V+CGRA hybrid system, elaborated upon in this and subsequent sections. The processor is on the left (green); the CGRA on the right (yellow); memory (blue) and control (red) are in the centre.

The CGRA is designed to communicate with the RISC-V via two types of interfaces: memory-mapped I/O ports on the CGRA and through shared memory between the processor and CGRA. As is apparent in Figures 2.3 and 2.4, different-sized ADRES and HyCUBE CGRAs will have different numbers of I/O s and memory ports. The hybrid system is modified accordingly (and automatically, described below), based on the number of available CGRA I/O s and memory ports.

Memory-mapped I/O is a widely used approach for connecting a processor with peripheral I/O devices. In this approach, specific memory addresses *outside* of the actual memory space are used for communication to/from I/O devices. A hardware register typically resides at each of these specific memory-mapped addresses. Writes to such addresses communicate data from the processor to the device; reads from such addresses retrieve data from the device. The elegance of the approach is that communication with I/O devices is reduced to loads/stores from the specific addresses. Additional hardware is added to the system to recognize such addresses and steer the memory accesses to the appropriate I/O devices.

Table 3.1 shows the memory-mapped registers in the control unit, when the CGRA is a 4×4 ADRES architecture and the addressable memory is assumed to lie in the range from 0x000000 to 0x3FFFFFF. The base address of the memory-mapped registers is thus set to 0x400000, and the locations are sequential and adjacent, starting at this base

IO_0_in	0x400000	input I/O port to ADRES	1.RISC_V processor write_data 2. ADRES IO_0_in
IO_0_out	0x400000	output I/O port to the processor	1.RISC_V processor read_data 2. ADRES IO_0_out
...	...	...	...
IO_3_in	0x40000c	input I/O port to ADRES	1.RISC_V processor write_data 2. ADRES IO_3_in
IO_3_out	0x40000c	output I/O port to the processor	1.RISC_V processor read_data 2. ADRES IO_3_out
counterPort	0x400010	1. enable/disable CGRA 2.value written into this port indicates the counter number	1.RISC_V processor write_data 2. ADRES CGRA_Enable
dataPort	0x400014	1. enable/disable CGRA 2.value written into this port determines the output port that report the “Done” signal	1.RISC_V processor write_data 2.ADRES CGRA_Enable

CGRA_Reset	0x40001c	reset CGRA	1.RISC_V processor write_data 2.ADRES CGRA_Reset
Done	0x400020	signal indicating CGRA is finished	RISC_V processor read_data

In the hybrid system, we employ a Harvard memory architecture, where the instruction and data memories are separated. The instruction memory is single ported, as only the processor accesses this. The data memory is dual-port and partitioned into banks, with two underpinning rationales: First, the RISC-V and CGRA should be able to work independently, meaning the RISC-V processor should be capable of continuing executing (and accessing data) while the CGRA concurrently performs operations. This necessitates dual-port memory functionality. In the present hybrid system design, the programmer is responsible for ensuring that the processor and CGRA do not access the same memory address while the CGRA is executing. Otherwise, unpredictable behaviour may result. This requirement is straightforward to meet in cases where the processor simply waits for the CGRA to complete its computational work. In future, a more complicated memory-consistency protocol may be pursued for concurrent processor/CGRA memory accesses.

Observe in Figure 3.1 that while the processor “sees” a single addressable memory space, the CGRA sees multiple separate banks of memory. Each CGRA memory port is connected to one memory bank, allowing concurrent access to all banks, thereby increasing memory-access parallelism to achieve higher computational throughput.

Table 3.2 shows the example memory banks and their connections for a 4×4 ADRES architecture with 4MB of addressable memory (22 bits of address space). For the processor, multiplexers are used to select the memory bank being accessed. For the example

Table 3.2: Example of memory banks for ADRES 4×4 CGRA and 4MB of memory (22-bits of address), connections to the RISC-V processor and CGRA accelerator.

Memory Bank	Address Range	Processor Connection	CGRA Connection
Bank_0	0x000000 to 0x0fffff	write_data read_data address write_request	write_data_0 read_data_0 address_0 write_request_0
Bank_1	0x100000 to 0x1fffff	write_data read_data address write_request	write_data_1 read_data_1 address_1 write_request_1
Bank_2	0x200000 to 0x2fffff	write_data read_data address write_request	write_data_2 read_data_2 address_2 write_request_2
Bank_3	0x300000 to 0x3fffff	write_data read_data address write_request	write_data_3 read_data_3 address_3 write_request_3

shown in Table 3.2, if the address from the processor falls into the address range of Bank_0, data transfers between the processor and Bank_0 and other banks are not affected. Each CGRA memory port connects to a separate memory bank. The memory is divided into four equal-sized banks because the 4×4 ADRES CGRA architecture has four memory ports. For an 8×8 CGRA, the memory would be partitioned into eight banks.

3.2 CGRA Control

For the proposed hybrid system, the processor is responsible for controlling and transferring data to/from the CGRA. The RISC-V processor executes a software program and invokes the CGRA for a computationally-intensive program hot spot – a compute kernel. The CGRA is pre-configured to accelerate the specific application. That is, the CGRA is configured with a bitstream for the kernel’s DFG, as produced by the mapper in CGRA-ME, shown in Figure 2.2.

As shown in Figure 3.1, a control unit resides between the RISC-V and the CGRA. The control unit contains memory-mapped registers and a counter. Apart for the memory-mapped registers corresponding to the CGRA’s I/O ports, two additional memory-mapped locations, *Enable* and *Reset*, allow the processor to enable and reset the CGRA. These locations are tied to the enable and reset inputs of the CGRA.

CGRAs generated by the CGRA-ME framework are capable of streaming dataflow-style computations – the CGRAs do not support presently control-flow instructions, e.g., branch. The processor and the control unit must manage the CGRA execution, including determining when the CGRA needs to start the computations and when the CGRA is “done” its work. Starting the CGRA involves the processor writing to the memory-mapped register that connects to CGRA Enable signal. The control unit stops the CGRA when it finishes the computation. A CGRA itself does not have hardware to decide if it has finished the computation, so two methods are developed to indicate the completion of the work. One is handled through a counter and the other is by using data input into the CGRA from memory. The processor determines if the counter or data is used to stop the CGRA by writing to one of the two memory-mapped locations, *counterPort* or *dataPort*. A third memory-mapped location, *Done* is used to indicate completion of the CGRA’s work. The *Done* signal is asserted when the control unit determines CGRA is “done” its work, and the processor proceeds to the next step after it reads the *Done* signal from the corresponding memory-mapped register.

3.2.1 Counter Control

When a DFG is mapped onto a CGRA, the CGRA-ME framework reports the latency (in clock cycles) for the mapping, and also the initiation interval (II). With this information, the programmer can determine how many clock cycles the CGRA needs to process a specified number of data inputs. For example, if the mapping latency is 3, and the II=1, and the programmer intends to have the CGRA execute 10 data inputs, the total number

of CGRA cycles required is $10 \times 1 + (3 - 1) = 12$. This value is written to the counter, as an initial value, before enabling the CGRA. The counter counts down and when it reaches 0, the CGRA is automatically disabled by the control unit, and a ‘1’ is written to the *Done* register. The processor can poll the *Done* register to determine that the CGRA has completed its work.

This counter-based method requires the user who writes the software for the RISC-V processor to also have sufficient hardware knowledge regarding the CGRA so that they can provide an accurate cycle count to stop the CGRA. In future, when the CGRA-ME framework is enhanced to support control-flow, the control can be made more automatic and transparent to the programmer.

3.2.2 Data Control

To avoid requiring the user to determine the number of CGRA execution clock cycles as mentioned above, an alternative approach to determine whether the CGRA has completed its work is developed by using data in the memory. This method is limited to applications where the CGRA loads data from memory. A special data value – a *sentinel* value – is chosen to indicate the end of computation. For example, consider an application where the expected valid data is comprised of non-negative integers. In such a case, a value of -1 could be chosen as the sentinel value. The -1 would be placed in memory at the end of the list of valid data inputs. The application implemented on the CGRA contains a compare operation, comparing the data read from memory with the sentinel constant. The output of the compare is mapped to a CGRA I/O port. When a ‘1’ appears on a designated I/O port, the control unit automatically stops the CGRA and asserts the *Done* signal to notify the processor.

By using data to indicate the completion of work, the user working on the software does not need to know the details of the CGRA computations. The tradeoff is that the processor and CGRA are using a more sophisticated protocol to indicate completion. In

addition, extra components are added to application DFGs for the comparison part and one of the CGRA I/O ports is used for the compare output.

3.3 Automated Hybrid System Generation

We modified CGRA-ME to automatically generate RISC-V+CGRA systems. The automation flow is illustrated in Figure 3.2. The inputs are the CGRA architecture specification (as before) and the desired memory size, $\mathbf{M}$, of the hybrid system: the size of the shared data memory between the processor and CGRA. Existing CGRA-ME functionality parses the architecture description into an in-memory device model. From this in-memory model, our hybrid-system writer extracts the number of CGRA I/O ports, $\mathbf{I}$, as well as the number of CGRA memory ports, $\mathbf{P}$. I/O s and memory ports are primitive elements in CGRA-ME that an architect may use in defining their desired CGRA architecture.

The hybrid system writer produces three Verilog RTL files. The first is RTL for the RISC-V/CGRA interface, as shown in the middle of Figure 3.1. This RTL module contains registers for every CGRA I/O. It also contains the control unit for starting/stopping the CGRA execution, as described above. The I/O registers, as well as the registers for *Enable*, *Reset*, *Done*, the counter control and the data control are memory mapped.

The second RTL file produced is the dual-port banked memory. The user-specified memory of size $\mathbf{M}$ is partitioned into $\mathbf{P}$ banks automatically, with each bank automatically connected to one of the $\mathbf{P}$ memory ports of the CGRA. Presently, we require that banks be of equal size, so that the memory is evenly divisible into the banks. Finally, as shown in Figure 3.2, we also generate a top-level Verilog model instantiating and connecting the RISC-V processor, memory, interface module, and the CGRA. This top-level model is used for RTL simulation of the complete RISC-V+CGRA system.

The hybrid system writer also produces a header file to assist with writing programs

Figure 3.2: Hybrid system generation flow.

executed on RISC-V processor. The header file, illustrated in the next chapter, defines addresses for memory-mapped registers and the starting address for each memory bank. The header file benefits the user by setting the addresses to names so that the user does not need to calculate the addresses manually or specify addresses in hex, which can be error prone.

With this generic approach to hybrid system generation, there is no dependence on the specific CGRA architecture used in the hybrid system. The Verilog for the CGRA itself is produced by the existing Verilog writer within CGRA-ME. The CGRA's internal composition may be *any* CGRA modellable by the CGRA-ME framework.

3.4 Summary

This chapter introduced the architecture of the hybrid system, including the connectivity between the processor and the CGRA. We described how the processor and CGRA communicate with one another, through memory-mapped I/O ports, or through shared memory. We also discussed how the processor invokes the CGRA, and two methods that may be used to indicate the CGRA has completed its work. The first uses a counter to track CGRA execution cycles; the second uses a sentinel-value approach. Finally, we also described the automated approach to hybrid system generation, implemented within CGRA-ME. The system generation is not tied to a specific CGRA architecture and is generic enough to be used for any CGRA modelled in CGRA-ME.

Chapter 4

Case Study

In this Chapter, we present a detailed case study on how the hybrid system can be used to implement an application, namely, a finite-input response (FIR) filter.

The *FIR* benchmark computes 200 outputs of a 3-tap FIR filter, whose coefficients are embedded in the DFG input to the CGRA-ME mapper. The mapper places the coefficients into constant units present in each ADRES PE (omitted in Figure 2.3 for clarity). For the 4×4 CGRA, one output is produced each cycle in equation 4.1 and for an 8×8 CGRA, two outputs are generated in parallel (equation 4.2). Outputs are saved into memory banks by the CGRA.

$$Z[i] = 4 * a_0[i] + 6 * a_1[i] + 2 * a_2[i] \quad (4.1)$$

$$\begin{aligned} Z_0[i] &= 4 * a_{00}[i] + 6 * a_{01}[i] + 2 * a_{02}[i] \\ Z_1[i] &= 4 * a_{10}[i] + 6 * a_{11}[i] + 2 * a_{12}[i] \end{aligned} \quad (4.2)$$

Figures 4.1 and 4.2 shows the dataflow graph code and its visualization, respectively, of the *FIR* application for the 4×4 CGRA using data control. For the DFG, each node is labeled with its opcode such as *input*, *constant*, *add*, *load*, etc. Nodes with opcodes *input* and *output* will be mapped to I/O ports, and nodes with opcode *load* and *store* will


```

digraph G {
  addr_size [opcode=const, constVal="4"];
  const0 [opcode=const, constVal="4"];
  const1 [opcode=const, constVal="3"];
  const2 [opcode=const, constVal="2"];
  const [opcode=const, constVal="-1"];
  addr [opcode=add];
  load0 [opcode=load];
  load1 [opcode=load];
  load2 [opcode=load];
  store [opcode=store];
  end [opcode=output];
  mul0 [opcode=mul];
  mul1 [opcode=mul];
  mul2 [opcode=mul];
  add0 [opcode=add];
  add1 [opcode=add];
  add2 [opcode=add];
  addr_size -> addr [operand=any2input];
  addr -> load0 [operand=addr];
  addr -> load1 [operand=addr];
  addr -> load2 [operand=addr];
  load0 -> mul0 [operand=RHS];
  const0 -> mul0 [operand= any2input];
  load1 -> mul1 [operand=any2input];
  const1 -> mul1 [operand=any2input];
  load2 -> mul2 [operand=any2input];
  const2 -> mul2 [operand=any2input];
  mul0 -> add0 [operand=any2input];
  mul1 -> add0 [operand=any2input];
  mul1 -> add1 [operand=any2input];
  mul2 -> add1 [operand=any2input];
  add0 -> add2 [operand=any2input];
  add1 -> add2 [operand=any2input];
  add2 -> store [operand=data];
  const -> compare [operand=LHS];
  load0 -> compare [operand=RHS];
  compare -> end;
  addr -> store [operand=addr];
  addr -> addr [operand=any2input];
}

```

Figure 4.1: DFG code of FIR for 4×4 CGRA using data control.

be mapped to memory ports. Edges between nodes represent data dependencies. In the code for the edges, *any2input* appears for arithmetic operations that are commutative, such as *add*, *multiply*. For non-commutative operations, like *subtract* and *compare*, *RHS* and *LHS* are used to define the correct input ordering. For memory operations, the specific port, *addr* or *data*, is given for the edge. For example in Figure 4.1, *load1* and *load2* connect to *mul1* with opcode *mul*. The edges are marked as *any2input*. This means that data from *load1* and *load2* are multiplied together and the product is represented by *mul1*.

In this application, observe that a node *addr* is used to increment the memory-access address by a specific size, *addr_size*. The increment is reflected by the node having a loop-back edge to itself in the DFG graph. In our case, the base address increases by 4 as

Figure 4.2: DFG graph of FIR for 4×4 CGRA using data control.

shown in constant node *addr_size*. Addresses updates happen inside the CGRA in parallel with computations, as well as loading/storing data. Since the 4×4 ADRES CGRA, used in our case study has four memory interface ports, we are able to map a 3-tap FIR filter onto the CGRA with inputs from three memory interface ports and output the fourth one, leveraging all 4 memory ports each clock cycle. In a larger 8×8 CGRA, two copies of the DFG can be accommodated, using 8 memory ports. Equations 5.1 and 5.2 reflect the designs on the 4×4 and 8×8 CGRA, respectively. The two CGRAs with different sizes implement the same application, but the 8×8 CGRA increases throughput by producing double the number of outputs each clock cycle.

This case study uses the data-based control (sentinel) to indicate completion. Notice that the dataflow graph contains a *compare* node whose output is fed to an output node

labelled *end*. When the signal on *end* becomes true, the CGRA has completed its work.

As mentioned in the previous chapter, each CGRA memory port is attached to one memory bank. The data must be assembled in the banks in the order expected by the CGRA application. As such, the processor must “know” which load/store operation is assigned to which specific CGRA memory port. This is analogous to pin location constraints in the FPGA context, where I/O signals can be locked to specific FPGA I/O ports. A text file containing the constraints is provided to the CGRA-ME mapper, which constrains the named DFG node to a specific CGRA location. Likewise, the *end* output must be mapped to a specific CGRA I/O port that the control unit “knows” to check to determine whether the CGRA has completed its work. The following shows the constraint file for FIR on 4×4 CGRA using data control, with DFG nodes on the left and memory and I/O port names on the right separated by the colon sign:

```

1      load0 : mem_0
2      load1 : mem_1
3      load2 : mem_3
4      store : mem_2
5      end : top_0
```

The text file specifies that each load and store in the DFG is mapped a specific memory port, and thereby, to a specific memory bank. It can be seen from the DFG graph in Figures 4.1 and 4.2 that *load0*, *load1*, *load2* represent a_0 , a_1 , a_2 , respectively in Equation 4.1. As such, the FIR input data must be assembled in banks as follows: memory bank 0 stores data for a_0 , memory bank 1 stores data for a_1 , and memory bank 3 stores data for a_2 . Memory bank 2 has data computed and stored by the CGRA. This data layout allows the CGRA to stream data from all banks every clock cycle in the correct order. Data in banks 0, 1 and 3 are multiplied by their embedded constants, respectively, and results stored to bank 2.

For the RISC-V processor, hand-written assembly is used since there is presently no

compiler for a high-level language that can satisfy the special needs of the hybrid system connectivity and control – this is intended for our future work. The assembly is compiled by the RISC-V C cross-compiler toolchain provided by the RISC-V foundation [24]. The RISC-V program prepares the input data in the memory banks, and then invokes the CGRA to compute the results and store them in the specified memory bank. The example application code used for data control is as follows:

```

1  #include "hybrid.h"
2  prep:
3      addi    a0,zero,200;
4      addi    a7,zero,0;
5      addi    t1,zero,0;
6      li      a1,mem_0;
7      li      a2,mem_1;
8      li      a3,mem_3;
9  store:
10     sw      t1,0(a1);
11     addi    t1,t1,1;
12     sw      t1,0(a2);
13     addi    t1,t1,1;
14     sw      t1,0(a3);
15     addi    t1,t1,1;
16     addi    a1,a1,4;
17     addi    a2,a2,4;
18     addi    a3,a3,4;
19     addi    a7,a7,1;
20     bne     a7,a0,store;
21     li      t1,-1;

```

```

22      sw      t1,0(a1);
23      li      a0,dataport;
24      addi     t6,zero,1;
25      li      a1, done;
26      li      a2, selport;
27      addi     a3,zero,0;
28      sw      a3,0(a2);
29      sw      t6,0(a0);
30  fetch:
31      lw      a7,0(a1);
32      bne     a7,t6,fetch;

```

The immediate values loaded to registers in line 6, 7 and 8 are defined by the included header file, which is automatically created by the CGRA-ME framework when generating the hybrid system. In this header file, the addresses for I/O ports and memory banks are calculated based on the number of I/O and memory ports. An example header file for a 4×4 ADRES CGRA is given here:

```

1  .set io_top_0 , 4194304
2  .set io_top_1 , 4194308
3  .set io_top_2 , 4194312
4  .set io_top_3 , 4194316
5  .set counter , 4194320
6  .set dataport , 4194324
7  .set reset , 4194328
8  .set selport , 4194332
9  .set done , 4194336
10 .set mem_0 , 0
11 .set mem_1 , 1048576

```

```

12  .set mem_2,      2097152
13  .set mem_3,      3145728

```

4.1 Data-Based Control

As mentioned, the case study example uses data to manage CGRA termination. As shown in Figure 4.2, the value *load0* is compared with a constant sentinel, *const* (-1), and the result is sent to the *end* port. The memory ports used to compare with the *const* are decided by the processor and the input text file.

Walking through the assembly code, lines 9-20 assemble synthetic data for input to the FIR filter: there are 200 entries in each of the three input arrays. Line 22 writes the sentinel value to the end of the *a1* array. Line 26 and 29 select data-based control to terminate CGRA. In lines 25-28, the output port *top_0* is specified as the I/O port to indicate termination. This tells the control unit to load from CGRA node *top_0* to determine if the computations are finished. Line 28 starts the CGRA computation. The loop shown on lines 29-31 polls for the *Done* flag in the control unit.

4.2 Counter Control

The example above illustrates the data-based control. In the alternative, counter-based control, the compare operation would not be part of the application DFG. And, the CGRA would have no role in determining whether it has completed its work. Instead, this is handled by the processor and the counter inside the control unit.

The CGRA 3-tap FIR application's DFG has an $II=1$ after mapping with the CGRA-ME mapper onto the 4×4 ADRES CGRA, and a latency of 4. With one output/cycle, the total number of clock cycles needed for computing 100 results is $200 + (4 - 1) = 203$ cycles. The RISC-V stores this value in the control unit's down counter. When the counter

reaches 0, the control unit automatically disables the CGRA. The RISC-V invokes the CGRA then polls on the memory-mapped *Done* register to determine when the CGRA has completed its work.

4.3 Summary

In this chapter, we presented an application case study for using the hybrid RISC-V+CGRA system. A 3-tap FIR filter was implemented, using all 4 memory ports on a 4×4 ADRES CGRA: three for input values, one for computed output values. We described the application DFG for implementation on the CGRA, and the corresponding RISC-V assembly for the software part. Two methods of managing CGRA termination were discussed – the data-based (sentinel) approach, and a counter-based approach.

Chapter 5

Experimental Study and Results

5.1 Setup

We evaluate the RISC-V+CGRA hybrid systems on four applications and compare with two different baselines: 1) RISC-V running software alone, and 2) RISC-V with a vector unit [23]. For the hybrid systems, we compare two systems with different sizes of CGRA: 4×4 and 8×8 . For the vector processor, we evaluate two systems with different numbers of lanes: 4 lanes and 8 lanes. We use ModelSim to simulate the designs for functional correctness, and also to gather the number of clock cycles required for each benchmark application. The RISC-V, CGRAs and the vector processor use a common clock, with the rationale being that the three platforms exhibit roughly the same critical path delay, as demonstrated below in the ASIC implementation results.

5.1.1 Applications

Four applications are used: matrix multiply (*MM*), K-nearest neighbor (*KNN*), a finite impulse response (*FIR*) filter and polynomial evaluation (*FIR*) via Taylor expansion of e^x . The applications were selected to leverage both memory-mapped and shared-memory communication between the processor and CGRA. In the previous chapter, we introduced

FIR; here we overview the other three applications.

RISC-V binary programs for the benchmark applications are stored in the instruction memory. The CGRA configuration bitstream for each application is stored in a Verilog file and fed into the CGRA during the configuration process. We perform a cycle-accurate RTL simulation of a processor-only version of each application, the processor+CGRA version of each application, and also the vector processor.

Matrix Multiply

We implement a multiply of matrix A , with size 20×40 , and matrix B , with size 40×20 , producing a total of 400 outputs. The CGRA is programmed with an application that computes one value of the output matrix. That is, the CGRA performs the dot product of a row of A and a column of B : 40 multiply-accumulate (MAC) operations.

In this application, two input ports are used to indicate the address where the CGRA begins to fetch data (for a row of A and a column of B). Internally, the CGRA increments this base address each cycle according to the word size. One output port is used to send the computed value back to the processor. Since the 4×4 ADRES CGRA, used in our case study, has four memory interface ports, we are able to map two MAC operations onto the CGRA, leveraging all 4 memory ports each clock cycle. In a larger 8×8 CGRA, two copies of the DFG can be accommodated, using 8 memory ports. Equations 5.1 and 5.2 illustrate the computations realized by the 4×4 and 8×8 ADRES CGRAs, respectively. The two CGRAs, with different sizes, implement the same application, but the larger CGRA should use fewer clock cycles for the row/column dot product. It is the number of available memory ports that limits us from adding additional compute kernel copies into the CGRA application.

For the 4×4 CGRA, matrix input data must be assembled in banks as follows: memory banks 1 and 3 store data from the rows of Matrix A and memory banks 2 and 4 store data from the columns of Matrix B . Banks 1 and 3 contain matrix data in row-

major order; banks 2 and 4 have column matrix data in column-major order. This data layout allows the CGRA to stream data from all banks every clock cycle in the correct order. Data in banks 1 and 3 is multiplied by data in banks 2 and 4, respectively, and accumulated. The RISC-V program passes the starting addresses of rows/columns to the CGRA via the memory-mapped I/Os, and then invokes the CGRA to compute one entry in the result matrix. With two MAC s/cycle, the total number of clock cycles needed for a dot product containing 40 MAC operations is $20 + (5 - 1) = 24$ cycles with a latency of 5. After the MAC operations of a row of A and a column of B are computed, the processor reads from the CGRA's output port and writes the dot product result back to memory, then it updates starting addresses to the CGRA input ports, resets the CGRA to clear the output port and enables it again to compute the next dot product of a row and a column until all 400 outputs are produced.

For the 4×4 CGRA:

$$output+ = Bank_1[i] * Bank_2[i] + Bank_3[i] * Bank_4[i] \quad (5.1)$$

For the 8×8 CGRA:

$$\begin{aligned} output+ = & Bank_1[i] * Bank_2[i] + Bank_3[i] * Bank_4[i] + \\ & Bank_5[i] * Bank_6[i] + Bank_7[i] * Bank_8[i] \end{aligned} \quad (5.2)$$

K-Nearest Neighbour

KNN uses Euclidean distance to classify objects using a training set of 200 examples with 2 features. Feature data for the training examples is stored in memory and feature data for the objects is input to CGRA I/O ports from the RISC-V. The 4×4 CGRA uses two memory banks for the 200 examples and two input ports for the objects as shown in Equation 5.3. The CGRA outputs the sum of the square of the distance between

the example and the object for both features to a third memory bank. Four objects are evaluated with the same 200 examples by changing values of object features to input ports. For the 8×8 CGRA, two instances are implemented in parallel to produce two outputs, as shown in Equation 5.4. In this case, features of two objects are fed into four CGRA I/O ports, and two memory banks store the example data of the training set. Additional instances could not be accommodated in the 8×8 , owing to the # of memory ports required and routability. All outputs are saved into memory banks by the CGRA.

The processor first stores the data into the memory banks, accordingly, it then sends the input data to the I/O ports and starts the CGRA computation. After it fetches the *Done* signal, it resets the CGRA to clear the internal registers and outputs of memory ports. Then it resends the object feature data and starts the computations again. It repeats the operations two more times for the 4×4 CGRA. The 8×8 CGRA can classify two objects at the same time, so the CGRA is reset once after the processor fetches the first *Done* signal. The processor then restarts the CGRA to finish the evaluation.

$$Z[i] = (Fexample_0[i] - Fobj_0)^2 + (Fexample_1[i] - Fobj_1)^2 \quad (5.3)$$

$$\begin{aligned} Z_0[i] &= (Fexample_0[i] - Fobj_{00})^2 + (Fexample_1[i] - Fobj_{01})^2 \\ Z_1[i] &= (Fexample_0[i] - Fobj_{10})^2 + (Fexample_1[i] - Fobj_{11})^2 \end{aligned} \quad (5.4)$$

Polynomial Evaluation of e^x

POLY computes the first four terms of a Taylor expansion for e^x . This is performed 60 times for different values of x , with x automatically incremented from 0 to 59 each cycle within the CGRA. The computations are performed in fixed point. For the 4×4 CGRA, one output for e^x is produced each cycle, while for the 8×8 CGRA, three outputs are produced every clock cycle. Outputs are saved into memory banks by the CGRA. Equations 5.5 and 5.6 illustrate the computations for the 4×4 and 8×8 CGRAs,

respectively. For the 4×4 CGRA, x increases by 1 each cycle and for the 8×8 CGRA, x increases by 3 each cycle. The CGRA handles all computations without any input from the processor in *POLY*. The processor only starts the CGRA and waits for the *Done* signal. Routability limits more copies of the computations to be mapped on the CGRA.

$$Z[i] = 64 + 64 * x + 32 * x^2 + 12 * x^3 (x = 0, 1, 2...59) \quad (5.5)$$

$$\begin{aligned} Z_0[i] &= 64 + 64 * x + 32 * x^2 + 12 * x^3 (x = 0, 3, 6, \dots, 57) \\ Z_1[i] &= 64 + 64 * x + 32 * x^2 + 12 * x^3 (x = 1, 4, 7, \dots, 58) \\ Z_2[i] &= 64 + 64 * x + 32 * x^2 + 12 * x^3 (x = 2, 5, 8, \dots, 59) \end{aligned} \quad (5.6)$$

5.2 Results

We first compare the hybrid CGRA systems with software alone on the RISC-V, then compare with the vector processor.

5.2.1 RISC-V+ADRES CGRA Hybrid System vs. RISC-V Software Alone

Cycle Latency Speedup

Cycle-latency results are shown in Table 5.1. The first column lists the application name. The second column gives the cycle count for software alone on the RISC-V. The third and fourth columns give cycle counts for the hybrid systems with 4×4 and 8×8 CGRAs, respectively. In these columns, the numbers in paratheses give the cycle-count speedup vs. software alone. The last row of the table shows the average speedup across all benchmarks relative to RISC-V software.

A bar chart to visualize the speedups is shown in Figure 5.1. We observe speedups from $11-19\times$ for the hybrid system with the 4×4 CGRA. Larger speedups from $14-37\times$

Table 5.1: Cycle latency of benchmarks executing: 1) in software on RISC-V, 2) on two types of hybrid RISC-V+CGRA systems, 3) on a RISC-V processor with vector extensions.

Benchmark	System				
	RISC-V software	Hybrid 4x4	Hybrid 8x8	4-lane Vec. Proc.	8-lane Vec. Proc.
MM	178918	16518 (10.83)	12518 (14.29)	29600 (6.04)	15200 (11.77)
KNN	18042	1296 (13.92)	670 (26.93)	2814 (6.41)	1410 (12.8)
FIR	4005	208 (19.25)	108 (37.08)	545 (7.35)	281 (14.25)
POLY	844	68 (12.41)	28 (30.14)	157 (5.38)	82 (10.29)
Avg. Speedp	1	14.1	27.11	6.3	12.3

are observed for hybrid system with the 8×8 CGRA. The larger CGRA can accommodate two parallel instances of the DFGs for the *MM*, *KNN* and *FIR* benchmarks, and three parallel instances of the DFG for the *POLY* benchmark. However, the relative speedup comparing the larger and smaller CGRA are not precisely $2\times$ and $3\times$, owing to some fixed overhead shared by the hybrid systems that cannot be accelerated.

Table 5.2 presents a cycle decomposition for the *MM* application on the hybrid systems. For *MM*, the internal loop of a dot product multiplication between a column and a row needs to be repeated 400 times. Consequently, even if a single cycle is added to each dot product, it will increase the total cycles by 400. We see, for example, that 400 cycles are spent to “Start CGRA”. The “Configure Input Ports” part happens for both updating rows and columns, which is why it takes $400 + 20 = 420$ cycles. The “Latency” section reflects the latency of the CGRA when computing the dot product. With a latency of 5 cycles, the total clock cycles increases by $400 \times 5 = 2000$ cycles. Compare and branch takes a different number of cycles depending on whether the condition is met, so the total number of cycles for “Branch” is not a multiple of 20. Comparing the 4×4 with the 8×8 CGRA, only the “Computation” portion of the runtime is affected. The other aspects of the runtime are constant cycle counts that are required for the processor and cannot be reduced. *MM* has large overheads for resetting the CGRAs, branching to decide if all columns/rows are finished, and so on. Hence, this application has a lower speedup ratio for the 8×8 CGRA relative to the 4×4 .

Table 5.3 presents a cycle decomposition for the *MM* application in software. The

Figure 5.1: Cycle count speedups of benchmarks compare to software on RISC-V with 1) three types of hybrid RISC-V+CGRA systems, 2) on a RISC-V processor with vector extensions.

inner-most loop is performed a total of $40 \times 20 \times 20 = 16000$ times. Each of these iterations performs a multiply and accumulate, leading to 32000 cycles for the “Computation”. “Load/Store” instructions take 48400 cycles, and the compare/branch requires 48428 cycles. Address updates and index updates require 32400 and 16420 cycles, respectively. Comparing Table 5.3 and Table 5.2, we can observe that CGRAs are able to accelerate a large portion of the inner-most loop for *MM* since the MAC operations are handled by the CGRA instead of the processor.

The large cycle-count speedups observed for the hybrid system vs. software alone arise from a number of factors:

- Spatial and pipeline parallelism:
 1. The CGRA computations are pipelined and all applications have an $II=1$, in the steady state producing outputs every clock cycle. For example, Taylor Expansion of e^x required 8 clock cycles for an output in software, while the CGRA can generate a result every clock cycle.

Table 5.2: Cycle decomposition of *MM* application on hybrid CGRA-based systems.

	4×4 CGRA	8×8 CGRA
Start CGRA	400	400
Latency	2000	2000
Computation	8000	4000
Configure Input Ports	420	420
Store output to memory	1200	1200
Address increment	840	840
Branch increment	440	440
Reset	800	800
Branch	2418	2418
Total	16518	12518

Table 5.3: Cycle decomposition of *MM* application in RISC-V software.

	Load/Store	Computation	Address Increment	Branch Increment	Branch	Other
Software	48400	32000	32400	16420	48428	1270

2. The CGRA can update addresses for memory interface ports in parallel with computations, but software requires separate instructions for this.

- Branches: CGRA computations are “branch free” because a counter is used to determine when the CGRA’s work is finished. Conversely, the processor needs to use compare-and-branch instructions to detect when a loop trip-count is reached, requiring several clock cycles.
- Memory access parallelism: the CGRAs are connected to multiple memory banks and memory interface ports can perform loads and stores at the same time. For example, for the FIR filter implemented on an 8×8 CGRA, six input data words are fetched from memory and two outputs are written to memory each clock cycle in steady state. In the processor, each load and store will require separate (non-parallel) memory accesses.

5.2.2 Area and Critical-Path Analysis

Synopsys Design Compiler is used to synthesize the RISC-V and the CGRAs to the NanGate 45nm standard-cell library [29]. Cadence Innovus is used to place and route the processor and CGRAs. We performed a layout extraction, and analyzed post-route performance using Synopsys PrimeTime. We synthesized, placed and routed two versions of the processor and CGRAs: area-optimized and timing-optimized. For the area-optimized, we supplied no performance constraint to synthesis and set a tight area constraint, causing it to select the smallest cells. For the timing-optimized, we supplied a tough-to-meet 1ns clock-period constraint to Design Compiler. The CGRA layout was floorplanned to keep PEs within rows/columns in the proper order in the physical layout. Area and critical path delay are evaluated separately for the RISC-V processor and the CGRAs. The post-layout results for area are presented Figure 5.2¹. The area-optimized RISC-V consumes 61.4K μm^2 . The timing-optimized RISC-V consumes 70.0K μm^2 . The 4×4 CGRA consumes about twice the area as the RISC-V. Unsurprisingly, the 8×8 CGRA consumes about 4 $\times$ the area of the smaller CGRA.

Table 5.4: Timing analysis. Timing results for RISC-V and CGRAs are post-layout; timing results for the vector units are post-synthesis.

Architecture	RISC-V		CGRA 4×4		CGRA 8×8		4-lane Vec.~Proc.(Syn)		8-lane Vec.~Proc.(Syn)	
	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time
CP (<i>ns</i>)	5.95	3.86	4.13 (Baseline)	2.22 (Baseline)	4.33 (Baseline)	2.41 (Baseline)	6.46	2.05	6.51	2.07

The post-layout results for critical path delay are presented in Table 5.4. The area-optimized RISC-V has a critical path delay of $\sim 6\text{ns}$. The timing-optimized RISC-V has a delay of $\sim 3.9\text{ns}$. The CGRA critical-path delay values in Table 5.4 represent floor values, beginning at a processing element, passing through the interconnect to an immediate neighbour processing element, passing through that neighbor’s ALU, and then reaching the local register file. Thus, the critical-path delay values in the table reflect

¹The area results do not include the RAMs, which can be sized according to the application needs.

Figure 5.2: Area analysis. Results for RISC-V and CGRAs are post-layout; results for the vector units are post-synthesis.

an application mapping scenario where all communicating PEs are placed immediately adjacent to one another. Such an ideal placement is not always possible, however, and the ADRES CGRA permits longer connections between PEs to be formed by combinationally *bypassing* one or more PEs (see bypass path in Figure 2.3). The delay costs associated with using one of such bypass connections is shown in Table 5.5. By combining the floor critical path delay data in Table 5.4, with the number of bypasses used for each application in its CGRA mapping (reported by the CGRA-ME mapper), and the bypass delay data in Table 5.5, one can compute the final critical path for each application mapped onto the CGRA.

The critical path data for each application is shown in Table 5.6. For each application, the number of bypasses on the longest path is presented. The last row of the table shows the final computed critical path delay. The CGRA critical-path delays are close to the RISC-V critical-path delay. As such, we believe it a reasonable design decision that the RISC-V and CGRA use the same clock signal. The final computed critical path delays

Table 5.5: Bypass delays (ns)

	Delay (ns)	
CGRA Size	Area-optimized	Timing-optimized
4×4	0.38	0.26
8×8	0.46	0.30

Table 5.6: Bypass counts and critical path delays on ADRES CGRA.

Application	MM				KNN				FIR				POLY			
CGRA Size	4×4		8×8		4×4		8×8		4×4		8×8		4×4		8×8	
Bypass Number	2		3		3		3		1		3		1		3	
Optimization	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time	Area	Time
Final CP (ns)	4.89	2.74	5.71	3.31	5.27	3.00	5.71	3.31	4.51	2.48	5.71	3.31	4.51	2.48	5.71	3.31

for CGRAs are lower than the RISC-V processor, so the critical path delay for the hybrid system is the same as the delay of the software implementation. This makes the speedups given in Table 5.1 also valid for wall-clock time.

To evaluate area and critical path delay for the memory, we experimented with an open-source framework called OpenRAM [11]. It is a memory compiler written in Python that can create the layout, netlists, timing and power models, placement and routing models, and other views necessary to use SRAMs in ASIC design. It also uses the NanGate 45nm standard-cell library. We generated small memories successfully that were operable in the 500-1000MHz range (i.e., considerably faster than the CGRA and RISC-V). This leads us to believe the clock period will not be the system bottleneck. However, the memory compiler does not yet support generation of large memories (it cannot generate a 4MB memory), so a detailed memory evaluation is left to future work.

5.2.3 Comparison with a RISC-V Vector Processor

Cycle Latency Speedup

An alternative approach to acceleration is through the use of a vector processor, which uses the single-instruction-multiple-data (SIMD) paradigm. We use the open-source RISC-V² [23] vector processor to execute the same 4 applications and assess performance

and area relative to hybrid CGRA-based systems. RISC-V² is a RISC-V processor with vector extensions, supporting instructions like vector-add `vadd` and vector-multiply `vmul`. We removed the floating point and divider units from the RISC-V² to permit a more fair comparison with the CGRA capabilities. The processor is configurable with different numbers of lanes, ranging from 2 to 16. Each lane contains one execution unit. We opted for 4-lane and 8-lane versions for this study, owing to them having similar area to the two CGRA sizes used (discussed below).

The four applications were implemented to run on the vector processor using the compiler provided by the RISC-V² project. Complete Verilog RTL for RISC-V² is available, allowing us to perform cycle-accurate simulation for verification and to extract cycle latencies. We modified the memory model used in RISC-V² to match with the hybrid CGRA systems above, namely, single-cycle loads/stores from/to memory². With the memory-model change, we can make an apples-to-apples comparison between application execution on the RISC-V² and the other systems considered. Applications execute partially on the scalar RISC-V, which invokes the vector-accelerator for compute-intensive portions of applications using vector instructions.

Cycle-latency results for the RISC-V² are shown in the right-most two columns of Table 5.1. The speedups vs. the RISC-V alone are appreciable: $6.3\times$ and $12.3\times$ for the 4-lane and 8-lane vector processors, respectively. Speedups exceed lane counts as the vector processor eliminates many compare/branch instructions in the scalar RISC-V software. Note however, these average speedups are smaller than those afforded by using a CGRA for acceleration. The vector processor is able to fetch and process multiple data items according to the # of lanes, but it still requires several cycles to compute results. One instruction executes at a time and the compute units are arranged in a 1D configuration. In contrast, in the CGRA, the DFG specifying the computations is embedded into a 2D array of computational units, permitting pipeline parallelism, where

²The RISC-V² originally used an L1 cache.

different types of computations are overlapped in time.

Area and Critical-Path Analysis

Figure 5.2 and Table 5.4 gives the area and speed performance of the vector unit (not including the RISC-V). Regarding area, the 4×4 CGRA consumes less area than the 4-lane vector processor, but the 8×8 CGRA is larger than the 8-lane processor because the 8-lane vector processor is double the size of the 4-lane vector processor, whereas 8×8 CGRA is $\sim 4\times$ the area of the smaller CGRA. We investigated why the area of the 4×4 CGRA having 16 ALUs is smaller than the 4-lane vector processor having just 4 ALUs. In the area-optimized 4-lane vector unit, only 29% of the area is consumed by the execution units containing its ALUs. 54% of area is consumed by the issue unit, with the majority of this consumed by the vector register files – one per lane. 11% of the area is consumed by the memory unit, and the remaining 6% of area in other circuits. The vector unit’s area is not dominated by its ALUs, but rather, by the area of its other internal structures.

Table 5.4 also reports the critical-path delay for the RISC-V² vector unit. The delays reported are post-synthesis not post-layout, and therefore likely to be underestimates. While the source code for the vector unit went through synthesis without issues, when we pushed the technology-mapped gate-level netlists through place/route, we encountered timing problems related to very high fanout nets. Nevertheless, the post-synthesis delays are similar to the CGRA and RISC-V delays.

Overall, the performance results demonstrate that our framework, while generic and automatic, can generate RISC-V+CGRA hybrid systems having performance competitive with alternative acceleration approaches.

Table 5.7: Cycle latency of benchmarks on hybrid system with 4×4 HyCUBE CGRA.

	MM	KNN	FIR	POLY
Hybrid 4×4 HyCUBE	13718	682	111	71
Speedup vs. Hybrid 4×4 ADRES	1.20	1.90	1.87	0.96
Speedup vs. Hybrid 8×8 ADRES	0.91	0.98	0.97	0.39

5.2.4 RISC-V+HYCUBE CGRA Hybrid System vs. RISC-V Software Alone

Finally, to illustrate the flexibility of our RISC-V+CGRA hybrid system generation framework, we “swapped out” the ADRES CGRA architecture with the HyCUBE CGRA architecture [13]. In our HyCUBE architecture, I/Os are present on the top/bottom; memory ports are present on the left/right. Relative to ADRES with the same # of PE rows/columns, HyCUBE has double the number of I/Os and memory ports.

With double number of I/Os and memory ports and improved routability, applications whose mappings are limited by the number of ports and I/Os can be mapped to smaller size of HyCUBE CGRA compared to ADRES. Table 5.7 shows cycle latency results for a hybrid system with a 4×4 HyCUBE CGRA and compares with the ADRES-based hybrid systems described above. For three benchmarks, *MM*, *KNN* and *FIR*, the DFG used for the 8×8 ADRES-based hybrid system was able to be mapped to the 4×4 HyCUBE. For these benchmarks, the 4×4 HyCUBE system has similar cycle counts to the 8×8 hybrid system, with the small differences due to different numbers of pipeline stages in the DFG’s CGRA mapping for HyCUBE. For *FIR*, the 4×4 HyCUBE could not accommodate the larger DFG used for the 8×8 ADRES – the mapper could not find a routing in the smaller CGRA. For this case, the 4×4 ADRES and HyCUBE have similar latency.

The results illustrate that application performance will depend significantly on the selected CGRA architecture, and that the proposed automated RISC-V+CGRA hybrid system generation framework can be applied for researching which architectures are best suited to a given basket of applications.

5.3 Summary

In this chapter, we described an experimental study comparing RISC-V software alone, with RISC-V+CGRA hybrid systems, and also with a RISC-V vector processor. Cycle latency speedups of $11\text{-}37\times$ were observed for the ADRES CGRA-based systems vs. software alone, depending on the size of the CGRA and the application. The significant speedups arise from the spatial and pipeline parallelism of the CGRA, memory-access parallelism, and branch-free operation. The vector processor, on the other hand, offered speedups ranging from $4\text{-}14\times$. All hardware blocks were implemented in 45nm standard cell ASIC technology to assess area and critical path delay. The post-layout performance of the CGRAs was similar to the RISC-V processor, in both area-optimized and timing-optimized implementations. To demonstrate the flexibility of the framework for automated RISC-V+CGRA system generation, the ADRES CGRA was exchanged with a HyCUBE CGRA – a significantly different CGRA architecture – and the performance was compared.

Chapter 6

Conclusion and Future Work

In this thesis, we described an open-source framework for the generation of RISC-V + CGRA hybrid systems, combining an established RISC-V pipelined processor implementation with CGRA accelerators as generated by the CGRA-ME framework. The processor and CGRA communicate through memory-mapped access to the CGRA’s I/O ports, and through shared memory. Hybrid system generation is automatic and generic – it is not tied to any specific CGRA architecture. Any CGRA that can be modelled by the CGRA-ME framework can be incorporated into a hybrid system. In a case study using the ADRES CGRA on a set of four benchmark applications, speedups of $11\text{-}19\times$ are observed vs. RISC-V software alone for a 4×4 CGRA+RISC-V, and $14\text{-}37\times$ for an 8×8 CGRA+RISC-V. Speedups of $\sim 2\times$ are observed for the RISC-V+CGRA systems vs. a RISC-V with a vector unit. We also showed the flexibility of the framework through use of an alternative CGRA architecture, HyCUBE. We believe the framework will enable a variety of new research on RISC-V+CGRA systems and associated programming methodologies.

Future directions are many. First, it is important to improve the programming model to allow high-level languages to be used instead of requiring the use of assembly. Work is currently underway to develop an MLIR [16]-based front-end compiler for the generated

hybrid systems. MLIR is a successor framework to LLVM that allows the program to be represented and optimized at different abstraction levels, including those appropriate for hardware implementation.

A second future direction surrounds memory. We would like to conduct a detailed analysis of memory area and performance, for different memory sizes. It would also be interesting to explore alternative memory architectures for hybrid systems, including those with a cache or multiple levels of cache.

A third direction is to analyze the power efficiency of hybrid CGRA systems. While this work looked at area and performance, prior work on accelerators has demonstrated significant improvements to energy efficiency.

Appendix A

Application DFGs

Figure A.1: DFG for FIR on 4×4 CGRA with counter control

Figure A.2: DFG for FIR on 8×8 CGRA with counter control

Figure A.3: DFG for KNN on 4×4 CGRA with counter control

Figure A.4: DFG for KNN on 8×8 CGRA with counter control

Figure A.5: DFG for MM on 4×4 CGRA with counter control

Figure A.6: DFG for MM on 8×8 CGRA with counter control Figure A.7: DFG for POLY on 4×4 CGRA with counter control

Figure A.8: DFG for POLY on 8×8 CGRA with counter control Figure A.9: DFG for FIR on 4×4 CGRA with data control

Figure A.10: DFG for FIR on 8×8 CGRA with data control

Bibliography

- [1] PolarFire SoC. <https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga>. Microchip Corp.
- [2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
- [3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.
- [4] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson. CGRA-ME: A unified framework for CGRA modelling and exploration. In *IEEE ASAP*, pages 184–189, 2017.
- [5] S. Alexander Chin, Kuang Ping Niu, Matthew Walker, Shizhang Yin, Alexander Mertens, Jongeun Lee, and Jason H. Anderson. Architecture exploration of standard-cell and FPGA-overlay CGRAs using the open-source CGRA-ME framework. In *ACM ISPD*, page 48–55, 2018.

- [6] S. Das, K. J. M. Martin, P. Coussy, and D. Rossi. A heterogeneous cluster with reconfigurable accelerator for energy efficient near-sensor data analytics. In *IEEE ISCAS*, 2018.
- [7] L. Fiolhais, F. Gonçalves, R. P. Duarte, M. Véstias, and J. T. de Sousa. Low energy heterogeneous computing with multiple RISC-V and CGRA cores. In *IEEE ISCAS*, pages 1–5, 2019.
- [8] 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 TVLSI*, 25(10):2700–2713, 2017.
- [9] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *ACM/IEEE ISCA*, pages 28–39, 1999.
- [10] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nandathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.
- [11] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6, 2016.
- [12] Wilfried Haensch. Scaling is over — what now? In *Annual Device Research Conference (DRC)*, pages 1–2, 2017.
- [13] M. Karunaratne, A. K. Mohite, T. Mitra, and L. Peh. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *ACM/IEEE DAC*, 2017.

- [14] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim. ULP-SRP: Ultra low power samsung reconfigurable processor for biomedical applications. In *IEEE FPT*, pages 329–334, 2012.
- [15] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *IEEE CGO*, pages 75–86, 2004.
- [16] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore’s law. *CoRR*, abs/2002.11054, 2020.
- [17] Xiaoyi Ling, Takahiro Notsu, and Jason Anderson. An open-source framework for the generation of RISC-V processor + CGRA accelerator systems. In *Euromicro Conference on Digital System Design*, 2021.
- [18] Feng Liu, Heejin Ahn, Stephen R. Beard, Taewook Oh, and David I. August. Dynaspam: Dynamic spatial architecture mapping using out of order instruction schedules. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 541–553, 2015.
- [19] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.*, 52(6), October 2019.
- [20] João D. Lopes and José T. de Sousa. Versat, a minimal coarse-grain reconfigurable array. In Inês Dutra, Rui Camacho, Jorge Barbosa, and Osni Marques, editors, *High Performance Computing for Computational Science – VECPAR 2016*, pages 174–187, Cham, 2017. Springer International Publishing.

- [21] Eric Matthews and Lesley Shannon. TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features. In Marco D. Santambrogio, Diana Göhringer, Dirk Stroobandt, Nele Mentens, and Jari Nurmi, editors, *FPL*.
- [22] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *FPL*, pages 61–70, 2003.
- [23] K. Patsidis, C. Nicopoulos, G. C. Sirakoulis, and G. Dimitrakopoulos. RISC-V²: A scalable RISC-V vector processor. In *IEEE ISCAS*, pages 1–5, 2020.
- [24] RISC-V Foundation. *The RISC-V Instruction Set Manual*, 2019.
- [25] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B. Kent, Peter Jamieson, and Jason Anderson. The VTR project: Architecture and CAD for FPGAs from Verilog to Routing. In *ACM FPGA*, page 77–86, 2012.
- [26] D. Rossi, F. Conti, M. Eggiman, S. Mach, A. D. Mauro, M. Guermendi, G. Tagliavini, A. Pullini, I. Loi, J. Chen, E. Flamand, and L. Benini. A 1.3TOPS/W @ 32GOPS fully integrated 10-core SoC for IoT end-nodes with 1.7 μ W cognitive wake-up from MRAM-based state-retentive sleep mode. In *IEEE ISSCC*, pages 60–62, 2021.
- [27] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini. PULP: A parallel ultra low power platform for next generation IoT applications. In *IEEE Hot Chips*, 2015.
- [28] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and

- computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.
- [29] J. E. Stine and et al. FreePDK: An open-source variation-aware design kit. In *IEEE MSE*, pages 173–174, 2007.
- [30] Brian Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Managing short-lived and long-lived values in coarse-grained reconfigurable arrays. In *2010 International Conference on Field Programmable Logic and Applications*, pages 380–387, 2010.
- [31] M. J. P. Walker and J. H. Anderson. Generic connectivity-based CGRA mapping via integer linear programming. In *IEEE FCCM*, pages 65–73, 2019.