

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# **CGRA-RISC: Simulation Infrastructure for Coupling CGRA Accelerator to RISC-V Processor**

**António Francisco Rente Ribeiro**

DISSERTATION



**FEUP** FACULDADE DE ENGENHARIA  
UNIVERSIDADE DO PORTO

Mestrado em Engenharia Informática e Computação

Supervisor: Dr. João Paulo de Castro Canas Ferreira

Second Supervisor: Dr. Nuno Miguel Cardanha Paulino

July 24, 2024



# **CGRA-RISC: Simulation Infrastructure for Coupling CGRA Accelerator to RISC-V Processor**

**António Francisco Rente Ribeiro**

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Dr. João Manuel Paiva Cardoso

External Examiner: Dr. Nuno Filipe Simões Santos Moraes da Silva Neves

Supervisor: Dr. Nuno Miguel Cardanha Paulino

July 24, 2024



# Resumo

O declínio da lei de Koomey, que estipula uma duplicação de performance por joule a cada 1.5 anos, acoplado a um aumento na procura por computação de alto desempenho e multi-domínio, nomeadamente Aprendizagem Profunda, tem vindo a potenciar dispositivos computacionais alternativos, referidos como aceleradores.

Incluído no conjunto destes dispositivos encontra-se o *Coarse-Grained Reconfigurable Array* (CGRA), uma arquitetura com uma elevada capacidade de paralelização e reconfiguração. Essencialmente, o CGRA inclui diversos Elementos de Processamento, capazes de paralelizar operações aritméticas e lógicas, integradas numa topologia de interconexão 2D, que direciona a transferência de dados entre elementos. CGRAs são mais reconfiguráveis que *Application-Specific Integrated Circuits* (ASICs) estáticos, e desempenham conjuntos de operações com uma maior eficiência energética que *Field-Programmable Gate Arrays* (FPGAs).

Dada a sua natureza, o processo de desenho, programação e verificação de CGRAs é uma tarefa árdua. Esta necessita de investigação contínua em ferramentas de compilação e programação que compõem a metodologia de *Design Space Exploration* (DSE). Ferramentas do estado da arte ajudam ao longo deste processo, a automatizar a geração de configurações e/ou gerar a especificação do *hardware*. Segue-se um passo de simulação, que avalia a exatidão da execução, considerando diversas métricas de *performance*.

Este trabalho tem por base um simulador de arquiteturas CGRA, escrito em Java, cujas capacidades foram alargadas, permitindo conectá-lo a um simulador RISC-V, que transfere dados e instruções com o acelerador. O principal objetivo é utilizar a plataforma de simulação integrada como ferramenta para DSE de CGRA, validar a comunicação entre RISC-V e CGRA, e por fim, abordar implementações em *hardware*.

A infraestrutura resultante foi avaliada de acordo com as suas capacidades para análise funcional do equipamento, avaliando a exatidão dos resultados. Demonstrou resultados promissores dada a sua capacidade de operar em simultâneo com um *Instruction Set Simulator* (ISS) do estado da arte, o X-HEEP. O sistema foi validado com sucesso na execução de dois kernels aritméticos, o produto escalar de vetores e um filtro *Finite Impulse Response* (FIR). Produziu consistentemente os resultados esperados e evidenciou uma robustez operacional ao longo de múltiplos tamanhos de malha de unidades e valores inseridos, atingindo valores de *performance* temporais satisfatórios.

**Keywords** Simulation, Design Space Exploration, RISC-V, Functional Verification



# Abstract

The downtrend of Koomey’s law, which accounts for the doubling of performance per joule every 1.5 years, coupled with an emergent demand for high-performance and multi-domain computing, such as Deep Learning (DL), has potentiated alternative computing devices, commonly called accelerators.

This set of devices includes the Coarse-Grained Reconfigurable Array (CGRA), a highly parallel and re-configurable architecture. Fundamentally, a CGRA contains different types of Processing Elements (PEs), capable of parallel arithmetic and logical operations. These are integrated into a 2D interconnection topology to direct the data transfers between elements. CGRAs stand in the middle of the accelerator spectrum, being more reconfigurable than static Application-specific Integrated Circuits (ASICs) and more energy efficient than Field-Programmable Gate Arrays (FPGAs).

However, their architecture makes designing, programming, and verifying CGRAs arduous, requiring further research in compilation and programming toolkits that compose its Design Space Exploration (DSE). Current state-of-the-art design frameworks help throughout this process by automating the generation of configurations or generating the hardware specification. A simulation step typically follows, which reports on the achieved correctness and evaluates different performance metrics.

This work builds upon an existing Java-based CGRA simulator, enhances its capabilities, and offers the option of pairing it with a RISC-V simulator that will exchange data and instructions with the accelerator, acting as a co-processor. The main objective is to use the integrated simulation platform as a tool for DSE of CGRA, validation of communication between RISC-V and CGRA, and consequent lowering to actual hardware implementations.

The simulation infrastructure was evaluated by its functional analysis capabilities, verifying the correctness of its results. It demonstrated promising results given its capability of operating simultaneously with a state-of-the-art RISC-V Instruction Set Simulator (ISS), X-HEEP. The system was successfully validated by executing two arithmetic kernels, the dot-product and Finite Impulse Response (FIR) filter. It consistently produced correct outputs and maintained reliable functionality across multiple mesh and input sizes while achieving acceptable timing performances.

**Keywords** Simulation, Design Space Exploration, RISC-V, Functional Verification



# Acknowledgements

First and foremost, I would like to express my deepest gratitude to my family— my father, mother, grandparents, and others— for their unwavering support and encouragement throughout my university career. I am eternally grateful for their love and sacrifices. A special mention goes to my grandfather, who is not here to witness this milestone. I would have loved for him to see me finishing this stage of my academic journey.

I would also like to thank my supervisor, Dr. João Paulo de Castro Canas Ferreira, and co-supervisor, Dr. Nuno Miguel Cardanha Paulino, for their invaluable guidance and continuous support during my research and project. Their insights were precious and significantly contributed to the quality of my work.

A heartfelt thanks to all the friends I have gathered over these five years at FEUP. The lunches, classes, countless long nights spent together working on projects, and group presentations have been some of the most memorable moments of this journey.

Finally, an honorary mention is given to the members of the SpeCS laboratory, Dr. João Carlos Viegas Martins Bispo, Dr. Luis Miguel Mendes Pimentel Alves de Sousa and all the students. Their support and collaboration in every meeting motivated me to achieve the outcomes showcased in this document.

António Ribeiro



*“I was an ordinary person who studied hard. There are no miracle people.  
It happens they get interested in this thing and they learn all this stuff, but they’re just people.”*

Richard Feynman



# Contents

|          |                                                                    |           |
|----------|--------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                                | <b>1</b>  |
| 1.1      | Context . . . . .                                                  | 1         |
| 1.2      | Motivation . . . . .                                               | 2         |
| 1.3      | Problem Statement . . . . .                                        | 3         |
| 1.4      | Objectives . . . . .                                               | 3         |
| 1.5      | Document Structure . . . . .                                       | 4         |
| <b>2</b> | <b>State of the art</b>                                            | <b>5</b>  |
| 2.1      | Background . . . . .                                               | 5         |
| 2.1.1    | RISC-V Instruction Set Architecture . . . . .                      | 5         |
| 2.1.2    | Coarse-Grained Reconfigurable Arrays . . . . .                     | 6         |
| 2.1.3    | Hardware/Software co-design . . . . .                              | 9         |
| 2.2      | Related Work . . . . .                                             | 11        |
| 2.2.1    | CGRA communication using micro-code instructions . . . . .         | 12        |
| 2.2.2    | CGRA generation platforms . . . . .                                | 14        |
| 2.3      | Summary and discussion . . . . .                                   | 24        |
| 2.3.1    | General considerations about frameworks . . . . .                  | 24        |
| 2.3.2    | Relevant features not present in most frameworks . . . . .         | 25        |
| <b>3</b> | <b>Proposed System Components</b>                                  | <b>27</b> |
| 3.1      | Type of execution and memory access abstraction . . . . .          | 28        |
| 3.2      | Choosing the RISC-V simulator . . . . .                            | 29        |
| 3.3      | Developing the micro-code specification . . . . .                  | 31        |
| 3.4      | Platform requisites . . . . .                                      | 31        |
| <b>4</b> | <b>Implementation of a Co-simulation System</b>                    | <b>33</b> |
| 4.1      | Micro-code specification . . . . .                                 | 33        |
| 4.1.1    | CGRA instruction definition . . . . .                              | 33        |
| 4.1.2    | CGRA Instruction Formats . . . . .                                 | 34        |
| 4.1.3    | Instructions . . . . .                                             | 35        |
| 4.2      | CGRASim simulator . . . . .                                        | 38        |
| 4.2.1    | CGRASim overview . . . . .                                         | 38        |
| 4.2.2    | YAML parser . . . . .                                              | 39        |
| 4.2.3    | Decoder and Controller structure . . . . .                         | 40        |
| 4.3      | Co-simulation with X-HEEP . . . . .                                | 42        |
| 4.4      | Helper Python scripts and mappers to create new programs . . . . . | 46        |

|          |                                                                |            |
|----------|----------------------------------------------------------------|------------|
| <b>5</b> | <b>Results and discussion</b>                                  | <b>49</b>  |
| 5.1      | Functional Analysis . . . . .                                  | 49         |
| 5.1.1    | Usability . . . . .                                            | 49         |
| 5.1.2    | Platform's limitations . . . . .                               | 51         |
| 5.1.3    | Testing new kernels . . . . .                                  | 52         |
| 5.2      | Experiments and benchmarks . . . . .                           | 53         |
| 5.2.1    | Outline of the two kernels used for platform testing . . . . . | 53         |
| 5.2.2    | Functional Verification . . . . .                              | 57         |
| 5.2.3    | Co-simulation experiments methodology . . . . .                | 57         |
| 5.2.4    | Timing and computing metrics for platform . . . . .            | 58         |
| 5.3      | Results and Discussion . . . . .                               | 60         |
| <b>6</b> | <b>Conclusion</b>                                              | <b>67</b>  |
| 6.1      | Closing remarks . . . . .                                      | 67         |
| 6.2      | Future work . . . . .                                          | 68         |
| 6.2.1    | Platform functionalities . . . . .                             | 68         |
| 6.2.2    | GUI interaction . . . . .                                      | 71         |
|          | <b>References</b>                                              | <b>73</b>  |
| <b>A</b> | <b>LSU element class</b>                                       | <b>81</b>  |
| <b>B</b> | <b><i>CGRAController</i> methods</b>                           | <b>83</b>  |
| <b>C</b> | <b>Three main components of the X-HEEP simulation</b>          | <b>85</b>  |
| C.1      | CGRAController SystemVerilog module . . . . .                  | 85         |
| C.2      | DPI function definition for the Co-simulation . . . . .        | 86         |
| C.3      | Executable file (main.c) . . . . .                             | 88         |
| <b>D</b> | <b>Micro-code generation with python</b>                       | <b>91</b>  |
| D.1      | parser.py script . . . . .                                     | 91         |
| D.2      | Example of input and output for the parser script . . . . .    | 93         |
| <b>E</b> | <b><i>UnaryIncrement</i> YAML architecture and class files</b> | <b>95</b>  |
| <b>F</b> | <b><i>FMTIInstruction</i> class</b>                            | <b>97</b>  |
| <b>G</b> | <b>Functional verification</b>                                 | <b>99</b>  |
| G.1      | Dot product test bench . . . . .                               | 99         |
| G.2      | FIR filter test bench . . . . .                                | 100        |
| G.3      | Co-simulation test bench . . . . .                             | 101        |
| <b>H</b> | <b>C implementation of the kernels</b>                         | <b>103</b> |
| H.1      | C version of the dot-product . . . . .                         | 103        |
| H.2      | C version of the FIR filter . . . . .                          | 104        |

# List of Figures

|      |                                                                                                                                                                                                                                                            |    |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1  | A illustration of a simple Coarse-Grained Reconfigurable Array ( <b>CGRA</b> ) showing: (a) the mesh topology, (b) the internal architecture of the Reconfigurable Cell, RC, and (c) an example of the configuration register (adapted from [68]). . . . . | 7  |
| 2.2  | Architecture comparison in regards to flexibility and performance . . . . .                                                                                                                                                                                | 8  |
| 2.3  | DT-CGRA and Eyeriss architecture diagrams . . . . .                                                                                                                                                                                                        | 8  |
| 2.4  | Architecture for Dynamically Reconfigurable Embedded Systems ( <b>ADRES</b> ) and HyCube . . . . .                                                                                                                                                         | 10 |
| 2.5  | Classical compilation flow for <b>CGRAs</b> , adapted from [58]. . . . .                                                                                                                                                                                   | 12 |
| 2.6  | RoCC instruction format, adapted from [34]. . . . .                                                                                                                                                                                                        | 13 |
| 2.7  | 32-bit word instruction format of the OpenEdgeCGRA architecture, adapted from [2] . . . . .                                                                                                                                                                | 13 |
| 2.8  | CGRA Compilation and Simulation Framework ( <b>CCF</b> ) instruction types, adapted from [21]. . . . .                                                                                                                                                     | 14 |
| 2.9  | eXtension for Integrated Accelerator Management ( <b>IXIAM</b> ) platform Instruction Set Architecture ( <b>ISA</b> ), adopted from [67] . . . . .                                                                                                         | 14 |
| 2.10 | CGRA-ME’s framework vision and initial development status, adapted from [17] . . . . .                                                                                                                                                                     | 16 |
| 2.11 | Hybrid system generation flow of the <b>CGRA-ME</b> with RISC-V support, adapted from [49] . . . . .                                                                                                                                                       | 17 |
| 2.12 | Overview of OpenCGRA Framework, adapted from [77] . . . . .                                                                                                                                                                                                | 17 |
| 2.13 | Overview of Morpher Framework, adapted from [82] . . . . .                                                                                                                                                                                                 | 18 |
| 2.14 | A High-Level Overview of <b>CCF</b> , adapted from [20] . . . . .                                                                                                                                                                                          | 19 |
| 2.15 | Pillars framework overview showing the main components, source [35] . . . . .                                                                                                                                                                              | 20 |
| 2.16 | Microarchitecture of Gemmini’s two-level spatial array, adapted from [31] . . . . .                                                                                                                                                                        | 22 |
| 2.17 | Gemmini hardware architectural template overview, adapted from [31] . . . . .                                                                                                                                                                              | 23 |
| 2.18 | <b>CGRA</b> co-simulation platforms developed by the Embedded Systems Laboratory ( <b>ESL</b> ), adapted from HEEPocrates [55] and HEEPsilon [2]. . . . .                                                                                                  | 24 |
| 3.1  | Proposed CGRA simulation platform . . . . .                                                                                                                                                                                                                | 27 |
| 4.1  | RISC-V base opcode map for RV32G and RV64G, adopted from [81] . . . . .                                                                                                                                                                                    | 34 |
| 4.2  | RISC-V base instruction formats, adopted from [81] . . . . .                                                                                                                                                                                               | 34 |
| 4.3  | Instruction formats included in the specification for the 25-bit leftover bits. . . . .                                                                                                                                                                    | 34 |
| 4.4  | Additional store live-in instruction for the proposed RISC-V extension. . . . .                                                                                                                                                                            | 36 |
| 4.5  | CGRASim simplified software diagram. . . . .                                                                                                                                                                                                               | 38 |
| 4.6  | YAML Ain’t Markup Language ( <b>YAML</b> ) parsing flow, using the <i>SnakeYAML</i> library . . . . .                                                                                                                                                      | 40 |
| 4.7  | Step process to execute an instruction. . . . .                                                                                                                                                                                                            | 41 |
| 4.8  | Diagram of the implemented co-simulation infrastructure. . . . .                                                                                                                                                                                           | 43 |
| 4.9  | Read Direct Programming Interface ( <b>DPI</b> ) call trace . . . . .                                                                                                                                                                                      | 45 |

|      |                                                                                                                                                                |    |
|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 4.10 | <i>Verilator</i> DPI callback sequence diagram. . . . .                                                                                                        | 46 |
| 4.11 | CGRAFlow's Graphical User Interface (GUI) dashboard. . . . .                                                                                                   | 47 |
| 5.1  | Communication protocol handling upon an erroneous instruction. . . . .                                                                                         | 51 |
| 5.2  | Dot product procedural graph. . . . .                                                                                                                          | 54 |
| 5.3  | Dot-product mapping in a $4 \times 4$ architecture . . . . .                                                                                                   | 54 |
| 5.4  | Finite Impulse Response (FIR) filter procedural graph. . . . .                                                                                                 | 55 |
| 5.5  | FIR filter mapping in a $5 \times 5$ architecture. . . . .                                                                                                     | 56 |
| 5.6  | FIR filter mapping in a $4 \times 4$ architecture. . . . .                                                                                                     | 56 |
| 5.7  | The output of the FIR filter with four input values. . . . .                                                                                                   | 57 |
| 5.8  | First verification strategy for the simulator . . . . .                                                                                                        | 57 |
| 5.9  | Second verification strategy, incorporating the co-simulation . . . . .                                                                                        | 58 |
| 5.10 | Variation of the simulation runtime depending on input size of arrays and different mesh configurations. Performed as the first verification scenario. . . . . | 60 |
| 5.11 | Number of useful operations per second of simulation runtime for different mesh sizes and number of input data elements. . . . .                               | 62 |
| 5.12 | Hypothetical time execution . . . . .                                                                                                                          | 63 |
| 5.13 | Timing of different run-time commands for a $12 \times 12$ mesh matrix . . . . .                                                                               | 65 |
| 5.14 | Timing of different run-time commands for the $4 \times 4$ and $5 \times 5$ meshes . . . . .                                                                   | 66 |
| 6.1  | Workflow for a co-simulation scenario with specialized compilation. . . . .                                                                                    | 69 |
| 6.2  | HEEPocrates architecture . . . . .                                                                                                                             | 70 |

# List of Tables

|     |                                                                                      |    |
|-----|--------------------------------------------------------------------------------------|----|
| 2.1 | Overview of different <b>CGRA</b> generation frameworks and their features . . . . . | 26 |
| 4.1 | Available instructions in the specification (ignoring the RISC-V prefix). . . . .    | 37 |
| 5.1 | $4 \times 4$ and $5 \times 5$ simulated runtime for 64 input elements. . . . .       | 61 |



# List of Listings

|    |                                                                                                                                |     |
|----|--------------------------------------------------------------------------------------------------------------------------------|-----|
| 1  | Step action in <code>GenericSpecsCGRA.java</code> . . . . .                                                                    | 39  |
| 2  | <b>YAML</b> architecture example . . . . .                                                                                     | 40  |
| 3  | JavaScript Object Notation ( <b>JSON</b> ) document that contains specification of the <b>CGRA</b> instruction format. . . . . | 41  |
| 4  | The Java code that implements instruction execution . . . . .                                                                  | 42  |
| 5  | The <i>always_ff</i> flip-flop block, that is executed on every positive edge of the clock . . . . .                           | 45  |
| 6  | The read <b>DPI</b> function definition. . . . .                                                                               | 45  |
| 7  | Input mnemonics for the parser . . . . .                                                                                       | 47  |
| 8  | The Load–Store Unit ( <b>LSU</b> ) Java class. . . . .                                                                         | 82  |
| 9  | <i>CGRAController</i> methods . . . . .                                                                                        | 84  |
| 10 | SystemVerilog module . . . . .                                                                                                 | 86  |
| 11 | DPI functions . . . . .                                                                                                        | 88  |
| 12 | main.c file content . . . . .                                                                                                  | 89  |
| 13 | Python Parser . . . . .                                                                                                        | 93  |
| 14 | Python Parser input file . . . . .                                                                                             | 94  |
| 15 | <b>YAML</b> architecture description that includes an <i>UnaryIncrement</i> unit. . . . .                                      | 95  |
| 16 | Unary Increment Processing Element ( <b>PE</b> ) . . . . .                                                                     | 96  |
| 17 | <i>FMTIInstruction</i> class . . . . .                                                                                         | 97  |
| 18 | Dot-Product CGRASim test-bench, for the first verification scenario. . . . .                                                   | 100 |
| 19 | <b>FIR</b> filter CGRASim test-bench, for the first verification scenario. . . . .                                             | 101 |
| 20 | <b>FIR</b> filter CGRASim test-bench, for the second verification scenario . . . . .                                           | 101 |
| 21 | Dot-product kernel in C . . . . .                                                                                              | 104 |
| 22 | <b>FIR</b> filter kernel in C . . . . .                                                                                        | 104 |



# Abbreviations and Acronyms

|                |                                                              |
|----------------|--------------------------------------------------------------|
| <b>AI</b>      | Artificial Intelligence                                      |
| <b>ADRES</b>   | Architecture for Dynamically Reconfigurable Embedded Systems |
| <b>AGU</b>     | Address Generation Unit                                      |
| <b>API</b>     | Application Programming Interface                            |
| <b>ASIC</b>    | Application-specific Integrated Circuit                      |
| <b>ALU</b>     | Arithmetic Logic Unit                                        |
| <b>BOOM</b>    | Berkeley Out-of-Order Machine                                |
| <b>CAD</b>     | Computer Aided Design                                        |
| <b>CCF</b>     | CGRA Compilation and Simulation Framework                    |
| <b>CGRA</b>    | Coarse-Grained Reconfigurable Array                          |
| <b>CNN</b>     | Convolutional Neural Networks                                |
| <b>CPU</b>     | Central Processor Unit                                       |
| <b>CV-X-IF</b> | Core-V eXtension interface                                   |
| <b>DFG</b>     | Data Flow Graph                                              |
| <b>DL</b>      | Deep Learning                                                |
| <b>DMA</b>     | Direct Memory Access                                         |
| <b>DNN</b>     | Deep Neural Network                                          |
| <b>DPI</b>     | Direct Programming Interface                                 |
| <b>DPR</b>     | Dynamic Partial Reconfiguration                              |
| <b>DSE</b>     | Design Space Exploration                                     |
| <b>DSP</b>     | Digital Signal Processing                                    |
| <b>EPFL</b>    | École Polytechnique Fédérale de Lausanne                     |
| <b>ESL</b>     | Embedded Systems Laboratory                                  |
| <b>FFT</b>     | Fast Fourier Transform                                       |

|              |                                                   |
|--------------|---------------------------------------------------|
| <b>FIFO</b>  | First In First Out                                |
| <b>FLOP</b>  | Floating point operations per second              |
| <b>FIR</b>   | Finite Impulse Response                           |
| <b>FP</b>    | Floating Point                                    |
| <b>FPGA</b>  | Field-Programmable Gate Array                     |
| <b>FU</b>    | Functional Unit                                   |
| <b>GNN</b>   | Graph Neural Network                              |
| <b>GPU</b>   | Graphical Processing Unit                         |
| <b>GUI</b>   | Graphical User Interface                          |
| <b>HAL</b>   | Hardware Abstraction Layer                        |
| <b>HDL</b>   | Hardware Description Language                     |
| <b>HW/SW</b> | Hardware/Software                                 |
| <b>IEEE</b>  | Institute of Electrical and Electronics Engineers |
| <b>II</b>    | Initiation Interval                               |
| <b>IDE</b>   | Integrated Development Environment                |
| <b>ILP</b>   | Integer Linear Programming                        |
| <b>IMC</b>   | In-Memory Computing                               |
| <b>I/O</b>   | Input Output                                      |
| <b>IoT</b>   | Internet of Things                                |
| <b>IP</b>    | Intellectual Property                             |
| <b>IPC</b>   | Inter-process Communication                       |
| <b>IR</b>    | Intermediate Representation                       |
| <b>ISA</b>   | Instruction Set Architecture                      |
| <b>ISS</b>   | Instruction Set Simulator                         |
| <b>IXIAM</b> | eXtension for Integrated Accelerator Management   |
| <b>JSON</b>  | JavaScript Object Notation                        |
| <b>JRE</b>   | Java Runtime Environment                          |
| <b>KNN</b>   | K-nearest neighbor                                |
| <b>LLVM</b>  | Low Level Virtual Machine                         |
| <b>LSU</b>   | Load–Store Unit                                   |

|              |                                                        |
|--------------|--------------------------------------------------------|
| <b>MIMD</b>  | Multiple Instructions, Multiple Data                   |
| <b>ML</b>    | Machine Learning                                       |
| <b>MLIR</b>  | Multi-Level Intermediate Representation                |
| <b>MMU</b>   | Memory Management Unit                                 |
| <b>MRRG</b>  | Modulo Routing Resource Graph                          |
| <b>MVC</b>   | Model View Controller                                  |
| <b>NoC</b>   | Network on a Chip                                      |
| <b>OBI</b>   | Open Bus Interface                                     |
| <b>ONNX</b>  | Open Neural Network Exchange                           |
| <b>OO</b>    | Object-oriented                                        |
| <b>OOP</b>   | Object-oriented programming                            |
| <b>OS</b>    | Operating System                                       |
| <b>PBT</b>   | Property-Based Testing                                 |
| <b>PE</b>    | Processing Element                                     |
| <b>PULP</b>  | Parallel Ultra Low Power                               |
| <b>RAM</b>   | Random Access Memory                                   |
| <b>RAMP</b>  | Resource-Aware Mapping                                 |
| <b>RISC</b>  | Reduced Instruction Set Computer                       |
| <b>RF</b>    | Register File                                          |
| <b>RoCC</b>  | Rocket Custom Co-processor                             |
| <b>RTOS</b>  | Real-time operating system                             |
| <b>RTL</b>   | Register Transfer Level                                |
| <b>SAT</b>   | Satisfiability                                         |
| <b>SE</b>    | Software Engineering                                   |
| <b>SIMD</b>  | Single Instruction, Multiple Data                      |
| <b>SoC</b>   | System on Chip                                         |
| <b>SPeCS</b> | Special-Purpose Computing Systems, Languages and Tools |
| <b>SRAM</b>  | Static Random-Access Memory                            |
| <b>TCP</b>   | Transmission Control Protocol                          |
| <b>TLB</b>   | Translation Lookaside Buffer                           |

**TLM** Transaction Level Modeling

**UART** Universal Asynchronous Receiver/Transmitter

**ULP** Ultra Low Power

**VHDL** VHSIC Hardware Description Language

**VLIW** Very-Long Instruction Word

**VLSI** Very Large Scale Integration

**XAIF** eXtensible Accelerator InterFace

**X-HEEP** eXtensible Heterogeneous Energy-Efficient Platform

**XML** Extensible Markup Language

**YAML** YAML Ain't Markup Language

# Chapter 1

## Introduction

### 1.1 Context

The limitations and downtrend perspective of Moores's and Dennard's law [12], coupled with the increasing demands for computing power, led to adopting more specialized choices to achieve different computing goals [84]. Although more radical approaches have been introduced (e.g., neuromorphic computing) to step away from the standard and limited von Neumann model, reconfigurable architectures present hardware components that are adaptable to different domains while delivering higher performance [50]. Instead of increasing the number of transistors, they can optimize the spatial configuration of silicon to better align with the computational requirements over time [68].

The Field-Programmable Gate Array (FPGA) is often one type of device associated with this group. It offers a versatile architecture composed of logic blocks that are routed through the interconnect and create complex combinational and sequential logic circuits. However, such capabilities come at a cost, including increased configuration and compilation times and lower clock frequency execution [68], motivating the research for other options.

As an alternative, the Coarse-Grained Reconfigurable Arrays (CGRAs) are one type of reconfigurable accelerator that organizes and connects different PE units to perform a specific set of operations, which are usually a repetitive mathematical or logical code section that can execute faster by exploiting parallelism between instructions and loop iterations. They present a good middle ground between the flexibility and computing capacity that current applications demand, with improved energy efficiency [58]. Compared to FPGAs, they offer higher clock frequencies and lessen the compilation and reconfiguration time [68]. Operating and modifying a CGRA depends on many factors, that need to be taken into account, such as their homogeneity, type of interconnect, context frame, partial reconfiguration capabilities, memory hierarchy, and host-processor communication [58].

With many configuration options, it becomes increasingly laborious to test each architecture and configuration set manually, and thus, frameworks that automatically generate and validate the

devices allow the designers to focus on the application rather than the configuration and architectural requirements for its execution. These frameworks explore the design space by employing multiple algorithms and using simulation as a tool to test the proposed approach, determining if the execution is within constraints and that it outputs the desired results. These tools need to include full configuration capabilities in an easy-to-manage way and, with the increasing demand for edge devices, offer a system-level view in which the **CGRA** is integrated. Co-simulation offers new ways of exploring the individual **CGRA** behavior and its interaction with a host-processor that performs the general computation and offloads the heavy work to the accelerator. Many of these frameworks include a compiler that takes a user-defined kernel and adapts it to the number of available processing elements.

Despite many existing efforts and contributions to developing such tools, there are many open issues on **CGRA** adoption and design [58], including difficulties in integrating them in different ecosystems, especially those that rely upon RISC-V. RISC-V is an **ISA** developed by UC Berkeley that provides a complete royalty-free instruction set adaptable to multiple scientific domains through extensions. Its open-source nature propelled its adoption in research and industry. The **ISA** includes feature sets specifically designed for embedded systems.

With these points in mind, we aim to create a co-simulation platform that enables users to test different **CGRA** configurations and couple it to a RISC-V simulator.

## 1.2 Motivation

The **CGRA** ecosystem still has many unexplored areas [58], requiring further research to take advantage of these powerful devices. The design space lacks established research platforms for benchmarking, compilation, and architecture templates [50].

Most existing **CGRA** generation and simulation frameworks originate from research papers and academic projects, culminating in similar frameworks with no clear standard, which is challenging to integrate and deploy into existing infrastructures. They often combine register transfer-level languages with unnecessary levels of detail, adding layers of complexity to the models, thus contributing to the steep learning curve. **CGRAs** tend to be domain-specific [68], where each implementation suits and can be retargetable to different types of requirements (e.g., energy consumption, speed of execution). This characteristic demands agile and accessible Design Space Exploration (**DSE**) instruments that allow users to test their created architectures or choose an ideal candidate for their performance and energy consumption requirements. A framework that takes in the architecture specification without the need to engage with lower-level hardware details and executes cycle-accurate simulations provides a user-friendly platform for testing and evaluating future deployments with minimal difficulty. To be functional-exact and couple the **CGRA** with a Central Processor Unit (**CPU**), the platform must implement interaction mechanisms to co-simulate a host and accelerator system. This mediation can be achieved with RISC-V instructions that command the co-processor.

A new RISC-V extension set (i.e., a micro-code-like configuration method) also streamlines the integration of the **CGRA** simulator with any RISC-V Instruction Set Simulator (**ISS**), and it enables the creation of a unified and generalized RISC-V compiler and **CGRA** mapper. This would ensure that both processes are intertwined in one step and that any **CGRA** interaction (loading instructions to the **CGRA**'s memory) before the execution starts should not be required.

### 1.3 Problem Statement

The following state-of-the-art overview suggests the current Coarse-Grained Reconfigurable Array (**CGRA**) co-design framework space has some limitations [58], such as the problematic optimal mapping and scheduling procedure, which is essentially an NP-Complete problem and necessitates heuristics to improve the speed of the placement, control-flow mappings, the lack of a programming model (without a straightforward programming interface, e.g., OpenMP), lack of floating-point support [68], difficulty in achieving the highest level of parallelism and use of **PEs** [50] and rare support for co-simulation. These points can deter its adoption by the general research community despite being essential to the **DSE** process that generates these devices. Among these difficulties is the ability to create co-simulation environments that integrate a **CGRA** and a RISC-V Instruction Set Simulator (**ISS**), providing a simulated System on Chip (**SoC**) platform. Additionally, many **CGRA** designs do not support Floating Point Processing Elements (**PEs**), an essential datatype for the Machine Learning domain. Thus, supporting this or similar datatypes, at least at the simulation level, is important to facilitate the adoption of these platforms for this type of workload. A unified hardware abstraction of a **CGRA**, considering the multiple configuration options they might display, is considered a problematic endeavor [50].

The research questions are as follows:

- How can we create a **CGRA** co-simulation infrastructure that can couple the accelerator to a RISC-V **ISS**, thus creating a co-processing environment that uses the processor to execute general computations and that, using a specific assembly implementation, triggers the offload of parallelizable operations and data to the **CGRA**?
- How can we build this platform to be not only modular and extensible but also configurable, such that users can easily test different **CGRA** designs, verify their operational correctness, iterating until they reach an optimal design?

### 1.4 Objectives

The dissertation will focus on designing and implementing a co-simulation platform that couples a general processor unit, a RISC-V processor, and a configurable **CGRA** accelerator.

In essence, the proposed objectives are as follows:

- Design a set of micro-instructions that will control the behavior of the simulated **CGRA**.

- Define a structured file format to specify the architectural parameters of a specific **CGRA** instance.
- Test the **CGRA** with pre-defined routines configuring it to execute simple computations, validated post-execution.
- Review existing RISC-V-based simulation platforms or designs and select one to fulfill the co-simulation.
- Implement the RISC-V & **CGRA** co-simulation, using the aforementioned instructions to communicate between the simulated processor and accelerator.
- Validate the co-simulation with different routines that will require different patterns of co-processing. Use them to compare results with a general processor-based execution.

These objectives will help attest to the capabilities of the proposed tool as an efficient simulation platform that enables an agile design of Hardware/Software (**HW/SW**) systems based on **CGRAs**, a faster simulation execution, and future integration with RISC-V compilation, given the instructions-based control of the **CGRA**.

As the document describes throughout the different chapters, these objectives were achieved with varying levels of satisfaction, which can be considered the contributions of this work.

## 1.5 Document Structure

The document is organized into six chapters. Chapter 1 provides the introduction, including motivation, problem statement and objectives. Chapter 2 includes an explanation of several relevant concepts, an overview of previous research works about the topic, and finally, a summary that illustrates many of these points in a table format. Chapter 3 describes our reasoning for determining the different proposed components, the platform's requisites and other considerations about the nature of the work. Chapter 4 details the implementation of the system, including aspects such as the micro-code specification, the software architecture of the application and the co-simulation elements. Chapter 5 covers the evaluation of the system, considering its functional analysis behavior in different types of algorithms, as well as a simulation timing analysis. Chapter 6 presents the concluding remarks on the research, findings discussed throughout the work, and potential directions for future developments.

## Chapter 2

# State of the art

This chapter overviews state-of-the-art Coarse-Grained Reconfigurable Array (**CGRA**) design frameworks. We introduce essential foundational knowledge and then compare different framework alternatives, presented in an accompanying table.

### 2.1 Background

This section explores fundamental concepts and relevant technologies to the research that follows. It delves into the details of the RISC-V **ISA**, **CGRAs** and their applications and the different methodologies used in **HW/SW** co-design.

#### 2.1.1 RISC-V Instruction Set Architecture

RISC-V is a royalty-free, open-source Instruction Set Architecture (**ISA**) developed at UC Berkeley between 2010 and 2015 and managed by RISC-V International. It enhances the Reduced Instruction Set Computer (**RISC**) design and differs from previous specifications that rely on intellectual property restrictions and have expensive licensing agreements [6]. In recent years, it has gained increased adoption from established and newer companies, such as Google and SiFive, founding members of the organization.

The base integer instruction set supports integer operations, 32-bit, and 64-bit memory addresses but allows for extensibility through standard extensions (e.g., the F extension to provide support for the Institute of Electrical and Electronics Engineers (**IEEE**) 754-2008 single-precision floating-point format). Other non-standard extensions increase functionality and adapt the architecture to specific domains [18]. RISC-V extensions can help Ultra Low Power (**ULP**) devices minimize power consumption while performing signal processing operations on the received wireless network [4] or integrate a single execution convolution operator to accelerate Deep Neural Networks (**DNNs**) [80].

The open-source nature of the **ISA** has contributed greatly to its adoption and study in the research community, which has made significant progress in the security mechanisms [54] and incorporating the architecture in the Machine Learning (**ML**) ecosystem [40]. The Internet of

Things (IoT) field has embraced RISC-V for two main reasons: its flexibility in supporting extensions for efficient power utilization [9] and its open-source licensing, which contributes to lower unit costs for devices built with RISC-V cores [40].

There have also been advancements in providing standard core designs, development languages and frameworks for heterogeneous systems, as demonstrated by the Gemmini accelerator generator [31], which couples a lower-power Rocket core [7] to a systolic array, using non-standard RISC-V custom instructions.

The simulation and emulation ecosystem is continuously growing. The standard QEMU emulator supports the RISC-V ISA, and some variants of this project have been released to fulfill specific demands, such as Unicorn [64], that focuses on the processor's security research, and Qbox [26], QEMU-SC <sup>1</sup>, and others that include SystemC capabilities to incorporate external simulated devices.

Additionally, there are many simulators, including Spike <sup>2</sup>, a functional simulator for the RISC-V ISA, that is built in C++ and integrated into the Chipyard ecosystem, RISC-V Virtual Prototype [38], that supports Transaction Level Modeling (TLM) with SystemC and interfaces to include other device models with co-simulation and Gem-5 [11], a system-level and micro-architecture simulator that supports multiple ISA and could be built from source to support an event-driven loop that uses TLM and SystemC [61]. However, none of the mentioned solutions are cycle-accurate since they rely on high-level abstractions and event-driven behaviors to describe the processor. For more accurate timing, synthesis-based FPGA simulators are commonly used, such as Firesim [41].

Other alternatives emphasize the extensibility of the ISA. The eXtensible Heterogeneous Energy-Efficient Platform (X-HEEP) [56] includes three open-source, single-core RISC-V architectures (cv32e40p, cv32e2, and cv32e40x), defined in SystemVerilog, that can be synthesized into an FPGA. It extends them with a configurable system bus, 32kB SRAM memories, different interrupt types, power management units, I/O, and other open-source Intellectual Property (IPs) that enable booting an Real-time operating system (RTOS). The system bus offers two types of operating policies (parallel or blocking master-slave connections) and is designed to interface with heterogeneous processing elements (peripherals and accelerators). It uses Verilator and Questasim for Register Transfer Level (RTL) simulation that allows benchmarking of each architecture for a given application, and at the moment, is used in an implementation that extends the design with a CGRA for ULP signal processing [2]. It also provides extensive documentation and project templates to create and integrate new hardware designs or customize the existing system.

### 2.1.2 Coarse-Grained Reconfigurable Arrays

A Coarse-Grained Reconfigurable Array (CGRA) is a special-purpose computation architecture composed by Processing Element (PE) units, which can perform arithmetic and logical operations and are interconnected by data paths in a mesh-type topology, as seen in Figure 2.1, first

<sup>1</sup><https://github.com/nvgreensocs/qemu-sc>

<sup>2</sup><https://github.com/riscv-software-src/riscv-isa-sim>

demonstrated in the 2000s [60, 73].

Regarding execution behavior, data flow is carefully orchestrated through different processing units to transform and output results cycle-by-cycle, corresponding to the Multiple Instructions, Multiple Data (MIMD) paradigm [50]. Each unit contributes with its computation, engaging in a coordinated behavior to aggregate output. This type of processing yields high concurrency and energy efficiency when applied to repetitive operations or data-intensive loop kernels, from matrix multiplications to Fast Fourier Transforms (FFTs). CGRAs are both spatially and temporally configurable. CGRAs are not recommended with kernels that contain complex or irregular control flow [50].

Figure 2.2 compares the competing architectures to CGRAs, within two critical metrics, the ease of programming and devices' performance. Both graphs reveal that the CGRA has one of the best energy-efficiency present in the group, only surpassed by Application-specific Integrated Circuits (ASICs). Figure 2.2b conveys the same scenario in terms of throughput performance, measured in operations per second, demonstrating the opportunities this architecture offers when applied to computationally demanding areas. It is possible to observe in Figure 2.2a, that CGRAs strike a balance between an ASIC's static and a general CPU programmability, that is, the level of control a user can employ to suit a specific task and the availability of interfaces for that end. However, CGRAs stand on the sub-group that offers lower programmability, compared to the CPU ecosystems that support high-level languages, or even the Graphical Processing Unit (GPU), that despite having a lower energy-efficiency, possesses suitable interfaces and libraries [44] for its use.

The runtime reconfiguration of the device broadens its use cases when deployed, given that the architecture easily accommodates more complex behaviors with specific PE activations and routing.

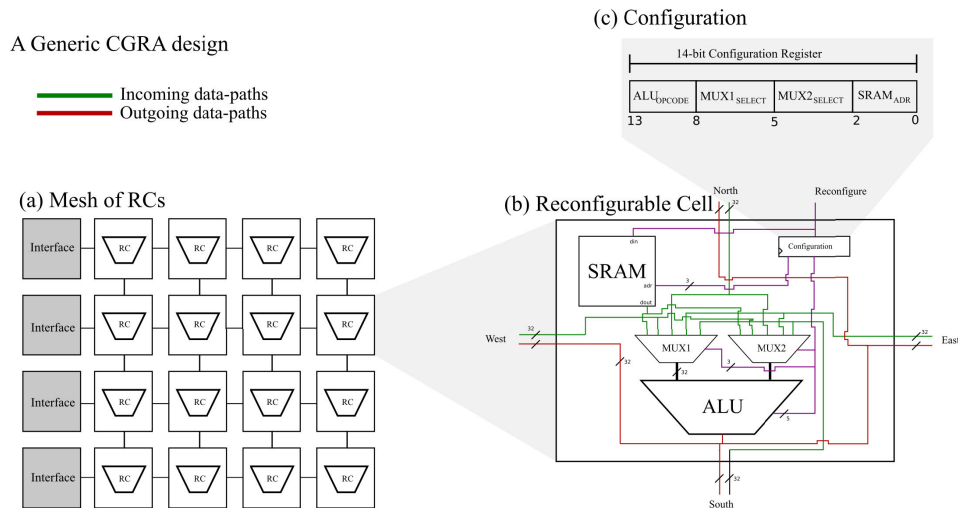


Figure 2.1: A illustration of a simple CGRA showing: (a) the mesh topology, (b) the internal architecture of the Reconfigurable Cell, RC, and (c) an example of the configuration register (adapted from [68]).

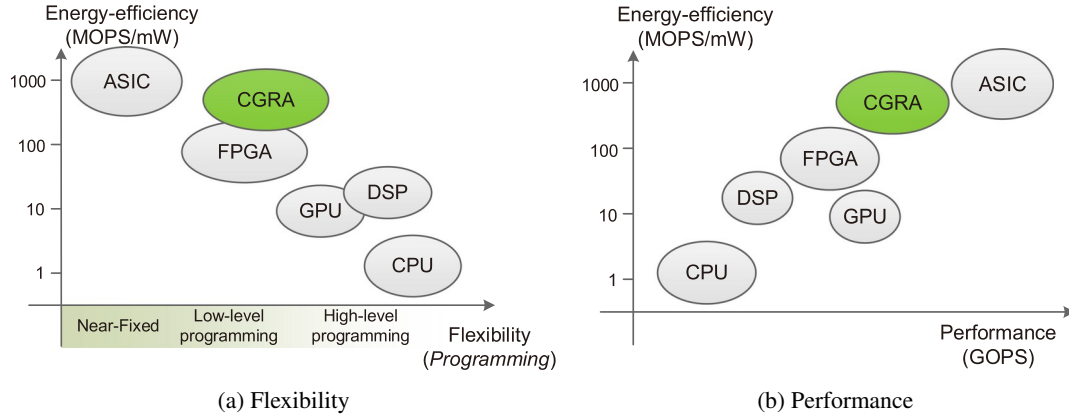


Figure 2.2: Architecture comparison in terms of flexibility, performance, and energy efficiency (from [51])

The granularity of operations (i.e., at the word or sub-word level) allows for a different trade-off in resource usage and synthesis/mapping time relative to **FPGAs** since arithmetic and logic operations are more directly mapped to these coarse units [68]. Most design toolkits provide configuration options, including the type and operation of **PE**, to support unique data representations (for instance, the 8-bit binary8, [68]), the utilization of different memory strategies, such as the use of Register Files (**RFs**) for caching, and the connection of the **PE**'s ports.

A prominent use case for **CGRAs** is the field of Artificial Intelligence (**AI**), which includes training in Deep Neural Networks and Convolutional Neural Networks. The parallelizable nature of the arithmetic operations employed in these models favors the utilization of interconnected **PEs**, individually configured to support data streams and transfer them to other units, providing high-throughput results. These characteristics can be observed in the DT-CGRA, [30] and Eyeriss [15], whose architectures are seen in Figure 2.3's diagrams. Additionally, alternative smaller data types, such as lower-precision floating-points (e.g., the brainfloat 16 format [33]), are favored by Neural Network models [62] and **CGRA** devices [68].

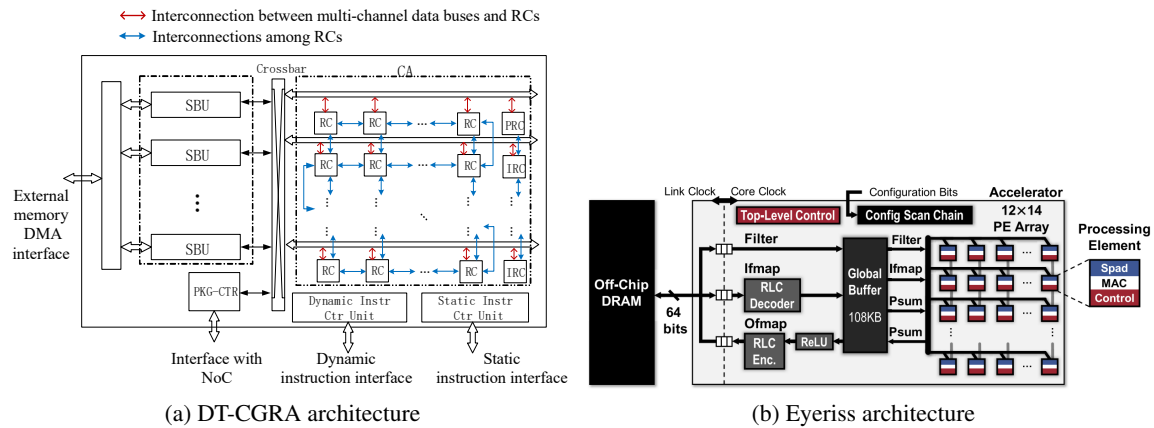


Figure 2.3: DT-CGRA and Eyeriss architecture diagrams, adapted from [30] and [15] respectively.

**CGRAs** have many other applications where lower power consumption is a vital aspect of the system [1], namely in **IoT** infrastructures, as they can be coupled as a specialized processing unit to a standard conventional processor. That is the case with the MCEA architecture [46], used in different edge applications.

In the medical field, a **CGRA** unit has been incorporated in HEAL-wear, where it analyzes biological signals [28]. Low reconfiguration overhead and low energy consumption are two relevant factors in its utilization. The authors claim that leveraging the **CGRA** reconfiguration with Single Instruction, Multiple Data (**SIMD**) execution brings a 1030% speed up and up to 37.2% energy savings, compared to a **ULP** multi-core, without **CGRA** acceleration. **CGRAs** have also been used in REACT [66], an algorithm for real-time epileptic seizure detection, with a 60% speed up compared to a conventional single-processor Digital Signal Processing (**DSP**).

The difficulty in coordinating the two-dimensional array of **PE** resources, as well as effectively exploring the parallelism in the source application, are listed as one of the main challenges regarding the hardware ecosystem [50], impacting the productivity of the design process. **DSE** tools aim to assist in navigating and optimizing these procedures.

Although each application and domain requires specific **CGRA** configurations, there are **CGRA** architectures that are seen as standards. Two of those cases are the HyCUBE [42] and the Architecture for Dynamically Reconfigurable Embedded Systems (**ADRES**) [59]. HyCube has a 2D mesh topology with a statically scheduled network where Functional Units (**FUs**) can contact distant **FUs** in a single cycle (different from a nearest-neighbor interconnect), coupled with a crossbar switch with clock-less repeaters for control signals and can send data to multiple **PEs** in the same cycle. The authors claim this design archives a 1.5x performance-per-watt improvement compared to standard Network on a Chip (**NoC**). **ADRES** joins a coarse-grained array that receives the intensive loop segment of the kernel with a Very-Long Instruction Word (**VLIW**) processor that increases parallelism of the other code portions, impacting performance and power efficiency. These share a **RF** and cache memory.

### 2.1.3 Hardware/Software co-design

A typical design process of hardware and software systems progresses through three consecutive steps: modeling, creating a model for the software and hardware components from the conceptualization of requirements; validation, which tests and evaluates the system to ensure it follows the requirements; implementation, the physical placement of the designed components, through synthesis and compilation [25].

Classical design techniques for digital circuitry have numerous challenges that hinder designers' productivity and the final product's quality. Separating the design of hardware and software can lead to a longer time-to-market, the occurrence of errors in the later parts of the design chain, underdesigning (not meeting the requirements), or overdesigning (being too strict and increasing the manufacturing costs). Additionally, the emergence of heterogeneous **SoC** devices with different application-specific units, peripherals, and memory storage, accompanied by the necessary

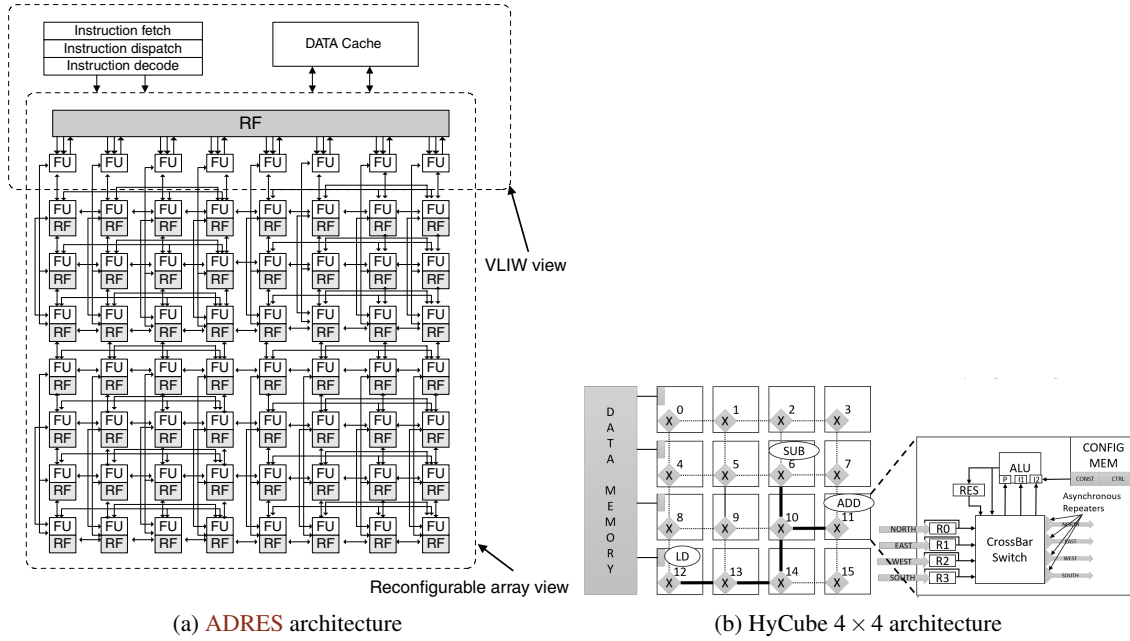


Figure 2.4: ADRES and HyCube’s architecture diagrams, adapted from [59] and [42] respectively.

communication networks (e.g., NoC, buses) generates more sophisticated designs that increase the difficulty of the design task when taken into consideration [78].

Hardware/Software co-design is a methodology in which hardware and software development are concurrently intertwined to reach system-level objectives, encouraged by new Computer Aided Design (CAD) tools and the expansion of integrated circuits. It is distributed throughout different areas (e.g., software compilation, Very Large Scale Integration (VLSI) circuit design), with applications in the three different stages of the design pipeline. Systems are assigned a value, usually performance-related, and characterized by the application domain (operated as an individual unit or embedded in a more extensive system), the degree of programmability (which actors have programming access and at what abstraction level it is performed), and hardware implementation characteristics [25].

One of the primary challenges of hardware/software co-design is hardware/software partitioning. It encompasses the task of identifying the segments of the program that provide better performance if offloaded to the hardware and how to compile them efficiently. Most of the systems that benefit from this process work off the co-processing paradigm, which divides the work between a centralized unit and an accelerator, with different levels of parallelism, either with direct synchronous processor control or parallel execution [25].

For this reason, current co-design strategies involve mapping procedures, which include allocation (selection of system resources and their connectivity), binding (assignment of operations to resources), and scheduling (taking the set of tasks, using their relations to assign a start time to execute or perform an action, with the intent of decreasing latency) [78]. In this work’s context, CGRA mapping establishes the configuration and behavior of the device by embedding the Data Flow Graph operations in each resource, usually described as a Modulo Routing Resource

Graph (**MRRG**), determining the schedule (timesteps), assignment of operations to specific **PEs**, and routing (selecting data paths between units). The goal is to generate the lowest possible Initiation Interval (**II**) between kernel iterations within the **CGRA** constraints [17]. It is typical for the Data Flow Graph (**DFG**) to be extracted from a loop kernel written in a high-level programming language. We can observe how intricate the pipeline of these operations is and how they intertwine in Figure 2.5.

With the increasing space of possible configurations and mapping permutations, co-design requires Design Space Exploration (**DSE**) to ensure the produced components are not under or overdesigned. The objective of **DSE** is to find the optimal design within the set of feasible solutions, that is, the ones that meet specific criteria (e.g., cost, performance, power consumption) [78]. This procedure must efficiently provide useful estimates to be integrated with engineers' workflow. For this reason, it can employ different types of exploration algorithms, either heuristic-based (e.g., simulated annealing) or proof-based techniques (e.g., Integer Linear Programming (**ILP**), Satisfiability (**SAT**) solvers).

**DSE** algorithms are commonly oriented in a multi-objective exploration that constrains solutions to multiple objectives. These algorithms evaluate each solution iteration through metrics that provide precise performance and cost outcomes according to the defined objectives. Simulation techniques can perform that evaluation, considering that they can estimate timing and power usage, which are essential goals we want to achieve while refining our design and stepping to the next iteration.

As mentioned in Section 1.2, current simulation techniques use a variation of **RTL** abstractions, which provide correct simulation results but are too time-consuming, given the detailed depth of the process. In co-simulation, the processor is elevated to a higher abstraction level than the target hardware that executes the offloaded software. [29]. Co-simulation abstracts the processor's execution to its functional behavior, excluding internal timing, and concentrates on the other components of the system and the necessary communication interfaces [78]. This technique simplifies the combination of multiple models. However, this task proves difficult when trying to guarantee the simulation accuracy. Dealing with the abstraction layers (e.g., adapted bus and memory access) without sacrificing the accuracy of the simulation demands different complementary types of simulation strategies [24], namely, a bus functional model, the instruction set model, and the cycle-accurate model. In this case, the priority is given to the accurate number of processor clock cycles throughout execution instead of the communication timing.

## 2.2 Related Work

This section reviews existing literature and prior research relevant to our work, including **CGRA** micro-code specifications and **CGRA** generation platforms. The former details how other platforms implement **CPU** to **CGRA** communication through instructions, and the latter delves into generation platforms that, through architecture and kernel inputs, provide automated generation and simulation of **CGRA** executions.

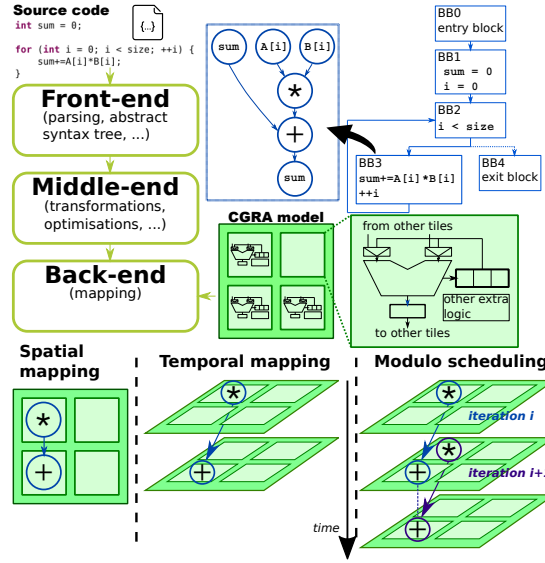


Figure 2.5: Classical compilation flow for **CGRAs**, adapted from [58].

### 2.2.1 CGRA communication using micro-code instructions

Many platforms utilize micro-code instructions to manage communication exchanges and control accelerators within their architectures.

The Rocket Custom Co-processor (**RoCC**) interface, included in the Gemmini [31] **SoC**, contemplates a bi-directional communication protocol between an accelerator and a RISC-V co-processor (Berkeley Out-of-Order Machine (**BOOM**) [14] or Rocket-core). This protocol incorporates handshakes for validating the requests and determining the device’s status, such as busy or ready to execute. Figure 2.6 represents the instruction format used in [34], which explored the matrix multiplication abilities of the accelerator. The **RoCC** Command packet delivers information to the accelerator to reproduce a given action, encapsulating a 32-bit instruction field (which includes funct, register, and opcode operands) with two 64-bit registers. A **RoCC** response sends data back to a specified register. The emphasis on Deep Learning (**DL**) requires distinctive compilation alternatives and the use of existing tools for automatic code generation (e.g., using the Apache TVM compiler [83]).

The OpenEdgeCGRA [2] architecture includes a 32-bit instruction set that has most of the bits encoding each **PE** location and Arithmetic Logic Unit (**ALU**) multiplexers routing, as seen in Figure 2.7. This architecture observed in Figure 2.18b integrates a kernel context memory pre-loaded with the instructions and loads them to the **PE**’s private memory according to the kernel specification. It also contains a configuration memory that specifies the start position of the context memory’s instructions and how many will be placed in the **PE**s.

The **CCF** framework [20] defines a 32-bit **CGRA ISA** for **PE** configuration and data communication between the host and the accelerator during the prolog, kernel, and epilog stages. It includes two instructions in the binaries generated from the annotated source kernels: the P and R types

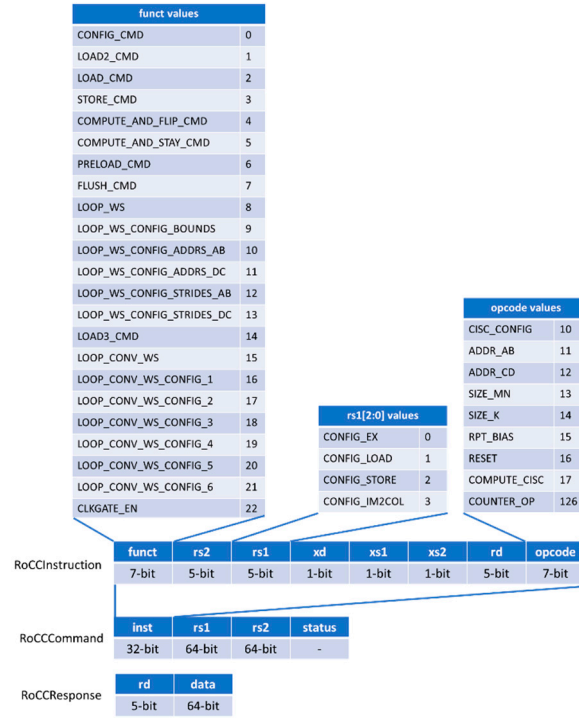


Figure 2.6: RoCC instruction format, adapted from [34].

| Field | muxAsel | muxBsel | aluOp | rfSel | rfWe | muxFsel | imm  |
|-------|---------|---------|-------|-------|------|---------|------|
| Bits  | 31:28   | 27:24   | 23:18 | 17:16 | 15   | 14:12   | 11:0 |

Figure 2.7: 32-bit word instruction format of the OpenEdgeCGRA architecture, adapted from [2]

(seen in Figure 2.8). The compiler stores the instructions as global variables that can be accessed by the **CGRA** since it is tightly coupled at the L2 cache level. According to its manual [21], most operations are mapped as an R-type instruction, such as addition, subtraction, multiplication, logical AND, and compare operators. The P-type defines more specific operations, such as predicated execution, address generation, and pre-loading values into the **CGRA**'s array. Each instruction delineates a virtualized opcode (depending on the operation), different kinds of operands for input sources (neighboring multiplexers, register files and memory access from the address or data bus, immediate value), and control bits to interpret the correct operands.

There are more comprehensive extensions aimed for **SoC** platforms that need to coordinate multiple accelerators at the time. That is the case of **IXIAM** extension [67], which accounts for synchronization, reservation, and data transference between the **CPU** and other accelerators (e.g., **CGRA**). To accomplish these objectives, the **ISA** includes instructions that trigger mechanisms for polling the devices' states and mutual exclusion of their availability, denoted in Figure 2.9b. To attend to heterogeneous environments, it can interface with different memory types (e.g., register-based, shared-memory) with a specific location format (Figure 2.9a) that distinguishes types and

|        |    |       |       |       |       |       |    |    |    |           |
|--------|----|-------|-------|-------|-------|-------|----|----|----|-----------|
| 31:28  | 27 | 26:24 | 23:21 | 20:19 | 18:17 | 16:15 | 14 | 13 | 12 | 11:0      |
| Opcode | P  | LMux  | RMux  | R1    | R2    | RW    | WE | AB | DB | Immediate |

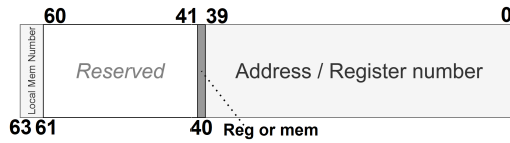
(a) R-type

|        |    |       |       |       |       |       |       |           |
|--------|----|-------|-------|-------|-------|-------|-------|-----------|
| 31:28  | 27 | 26:24 | 23:21 | 20:19 | 18:17 | 16:15 | 14:12 | 11:0      |
| Opcode | P  | LMux  | RMux  | R1    | R2    | RP    | PMux  | Immediate |

(b) P-type

Figure 2.8: CCF instruction types, adapted from [21].

specifies the correct unit inside an accelerator. The system’s complexity requires more bits to encode this information and auxiliary control registers to store process and accelerator identification codes.



(a) Request generated by different instructions

| Instruction | Input                          | Output | CPU-reg | CSR    |
|-------------|--------------------------------|--------|---------|--------|
| RESERVE     | accId                          |        |         | procId |
| CHECK       | accId                          | ret    |         | procId |
| TGL         | size, src_cpu,<br>dest_acc     |        | accId   | procId |
| TGS         | size, src_acc,<br>dest_cpu     |        | accId   | procId |
| TL          | size, src_acc,<br>dest_acc     |        | accId   | procId |
| TRL         | size, src_reg_cpu,<br>dest_acc |        | accId   | procId |
| TRS         | size, src_acc                  | ret    | accId   | procId |
| EXEC        | accId, opId                    |        |         | procId |
| ISBUSY      | accId                          | ret    |         | procId |
| RELEASE     | accId                          |        |         | procId |

(b) Instruction table for the ISA

Figure 2.9: IXIAM platform ISA, adopted from [67]

## 2.2.2 CGRA generation platforms

The following paragraphs illustrate different state-of-the-art frameworks researchers use to design **CGRA**s, typically organized into three main stages, requiring two input types. Most commonly, the inputs are an architectural overview of the available resources, modeled as a Modulo Routing Resource Graph (**MRRG**) and a code segment, usually a loop, in a high-level language. Then, a compiler (typically Low Level Virtual Machine (**LLVM**)) extracts a **DFG** and its Intermediate Representation (**IR**). As mentioned in Section 2.1.3, the resource and the data graph are combined in the Mapping stage with multiple algorithm alternatives (e.g., simulated annealing, **ILP**, Resource-Aware Mapping (**RAMP**) [19]), specifying which operations go into each **PE** and where to route their ports’ outputs to achieve maximum performance. The final stage includes a simulation, serving two purposes: verifying if the output design’s execution matches the expected results and providing execution estimations. Some frameworks, such as OpenCGRA, interface with other tools to generate the hardware.

These frameworks share many similarities but also some distinctive features; for instance, CGRA-ME has the possibility of using an Extensible Markup Language (XML)-based language for the CGRA's architecture, and OpenCGRA includes Property-Based Testing (PBT).

Upon further analysis, we find that most frameworks include essential modeling components that are modifiable in connectivity and bit-width. They generally include ALU units with arithmetic operations (e.g., multipliers), logic units (e.g., multiplexers), and LSU. Memory-wise, we find Register File (RF) or other memory banks.

Some frameworks allow for heterogeneity at several levels, namely within individual PEs and over the general CGRA structure. For example, the OpenCGRA [77] framework uses the term when specifying PEs that contain a chain of simpler operations executed in a single cycle. CGRA-ME [17] defines heterogeneity when describing a compute fabric that connects PEs with different operation capabilities (e.g., ALUs connected to multiply-only units).

Each listed framework includes case studies that demonstrate each their viability by generating a simple CGRA design and executing different types of kernels that validate the proposed approach. Most opt for a  $4 \times 4$  design with simple PEs. Still, some include variants of standard architectures, such as the HyCUBE in Morpher, and the ADRES in Pillars, evaluated across embedded benchmarks and often compared to other frameworks' results.

**CGRA-ME [17]** Implements a full-stack framework for CGRA design, including mapping procedures and Verilog RTL generation for simulation purposes and physical implementation, seen in Figure 2.10. It uses an XML-based language (CGRA-ADL), used as the CGRA architecture model, which includes "primitive" tags (multiplexers, registers, RFs, ALUs), modifiable in terms of bit-width and input that can compose into custom user-defined components. The framework resorts to the LLVM frontend, which gathers the initial DFG from a source kernel. In the case study, the authors demonstrate CGRA-ME's capabilities with the generation of block architectures ( $4 \times 4$  to  $6 \times 6$ ), two interconnect styles, different types of PE, and their evaluation across several operation tests, such as multiplication and Taylor series. A more recent publication [5] details the current evolution of the initial framework, which includes an application-agnostic RTL generation that can provide cycle-accurate functional verification with the ModelSim simulator environment, for example.

An improved release of the original system (CGRA-ME 2.0 [69]) now offers a C++ Application Programming Interface (API) for design creation, ALUs capable of Floating Point (FP) operations through [23], different mapping algorithms (e.g., ILP, simulated annealing) and control divergence with PE predication (selecting routes that execute based on a predicate condition).

Additionally, [49] creates a framework for the automatic generation of hybrid systems that encompass a RISC-V processor (based on the Parallel Ultra Low Power (PULP) [70] project) loosely coupled with CGRA accelerators generated by CGRA-ME. The RISC-V processor communicates with the CGRA through memory-mapped I/Os and shared memory. The framework's mapper outputs from the DFG a configuration bitstream as a Verilog file fed into the CGRA model. The processor accepts hex files compiled with the RISC-V tool-chain. The processor and the control unit

manage the start and end of the execution (with the *Enable* and *Reset* ports). The *hybrid.h* header file is provided to help users write a hand-written assembly that is necessary to embed into the core's executable. According to the documentation<sup>3</sup>, there are two modes that the host-processor can alternate for operating the **CGRA**. The counter implementation assumes the core writes in a control unit register the number of cycles required to complete the kernel. When the value reaches zero, this unit is signaled, setting a memory-mapped "Done" register to 1 so the processor may proceed. The other approach is implemented in executions that require the **CGRA** to load values from the shared memory, where the host processor places input data. Here, a pre-defined *sentinel* constant (e.g., -1) is compared to each value the control unit reads and flags the program's termination through the output port. The architecture description file must coordinate the ports used for this operation. The main compilation framework generates other Verilog **RTL** files (observed in Figure 2.11) from the original architecture description: the co-simulation interface (that contains Input Output (**I/O**) registers, and the control unit), the dual-port memory bank and a top-level model that instantiates and connects all modules, used for **RTL** simulation. In a case study using the **ADRES** CGRA on a set of four benchmarks (i.e., **FIR** filter, matrix multiplication, K-nearest neighbor (**KNN**), polynomial evaluation), speedups of 11-19x were observed compared to RISC-V software alone for a 4x4 CGRA+RISC-V and 14-37x for an  $8 \times 8$  CGRA+RISC-V. Additionally, a speedup of approximately 2x was observed for the RISC-V+CGRA systems compared to a RISC-V with a vector unit.

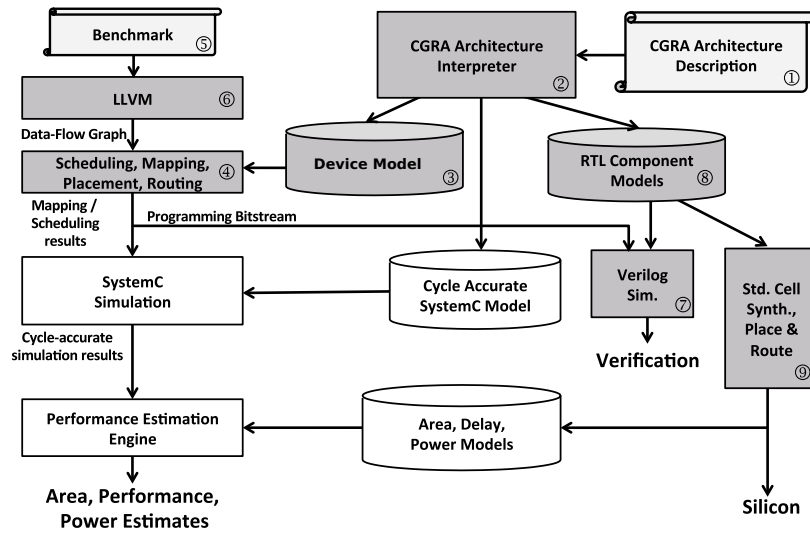


Figure 2.10: CGRA-ME's framework vision and initial development status, adapted from [17]

**OpenCGRA [77]** Leverages the Python language and ecosystem to construct a full-stack open-source pipeline that includes modeling, testing, and evaluating of **CGRA** devices. It is divided

<sup>3</sup>[https://cgra-me.ece.utoronto.ca/docs/features/hybrid\\_system/](https://cgra-me.ece.utoronto.ca/docs/features/hybrid_system/)

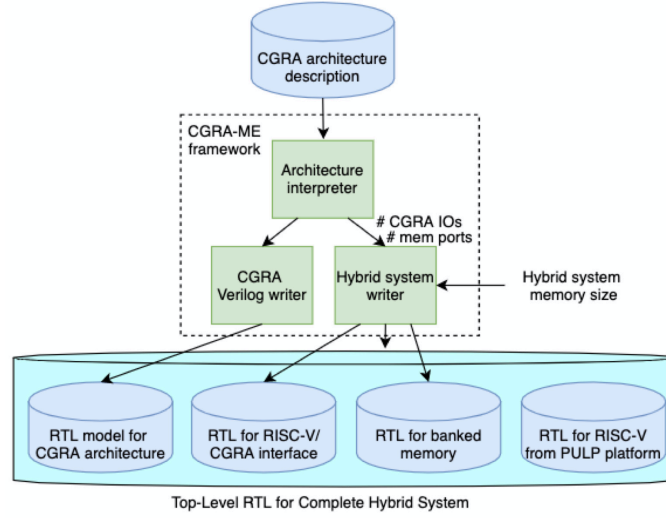


Figure 2.11: Hybrid system generation flow of the **CGRA-ME** with RISC-V support, adapted from [49]

into three stages: modeling, testing, and evaluation, as seen in Figure 2.12. It integrates open-source libraries such as the PyMTL hardware modeling and simulation framework [53], which can perform cycle-accurate simulation, hypothesis for **PBT** of each component, and mflowgen [13], a build-system generator that interfaces with tools from Synopsys and Cadence for synthesis, routing, and performance evaluation. The **LLVM** frontend provides the **DFG** and executes the mapping algorithm that uses a proximity cost heuristic and evaluates which heterogeneous **PEs** better suit a set of tasks. These extrapolated units need to be manually introduced by the users in the Python source code that describes the architecture. It considers three abstraction levels for modeling and simulation: functional, cycle-level, and **RTL**. As a general-purpose language, Python accommodates different levels of expertise. The case study describes the creation of a  $4 \times 4$  heterogeneous **CGRA**, tested across different embedded kernels (UTDSP, CBench, and Mibench).

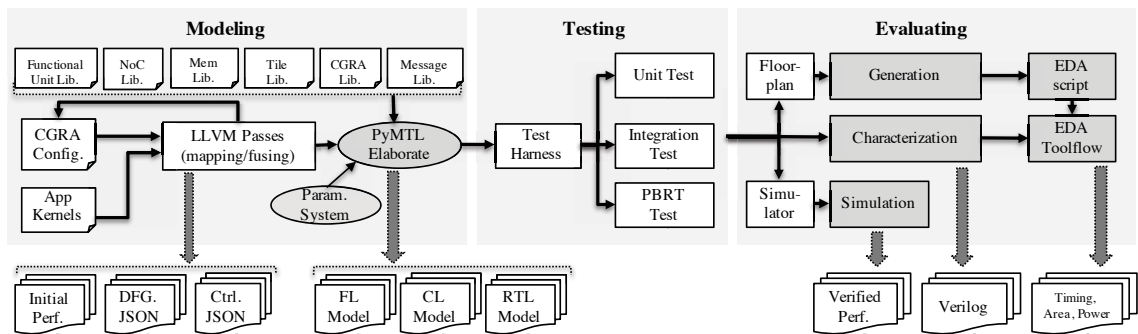


Figure 2.12: Overview of OpenCGRA Framework, adapted from [77]

**Morpher** [82] Describes an open-source full-stack **CGRA** compilation and cycle-accurate simulation framework. It supports complex kernels with control divergence and recurrence edges.

Users can provide an architecture as an **MRRG**, in which components (or modules) describe functional units, **RFs**, and memory units that can be coupled to form more complex modules. One of Morpher’s key aspects is the initial **LLVM** pass, which, in addition to the **DFG** extraction, locates live-in and live-out variables used to automatically verify the **CGRA** layout in the simulation, compared to a general-purpose execution (steps 3 and 6 in Figure 2.13). The mapping algorithm aims to maximize the parallelism and the throughput of the final configuration, starting by sorting the **DFG** in topological order, mapping the operations to the units, and routing the connections with Dijkstra’s shortest path algorithm, an Adaptive-heuristic, Simulated Annealing or a Graph Neural Network (**GNN**). The case study centered around a  $4 \times 4$  **CGRA** based on the HyCube architecture [43], tested with **ML** kernels, which demonstrated an improvement over CGRA-ME’s execution time by an average factor of 13.3. Unshaded portions in Figure 2.13 are not yet implemented.

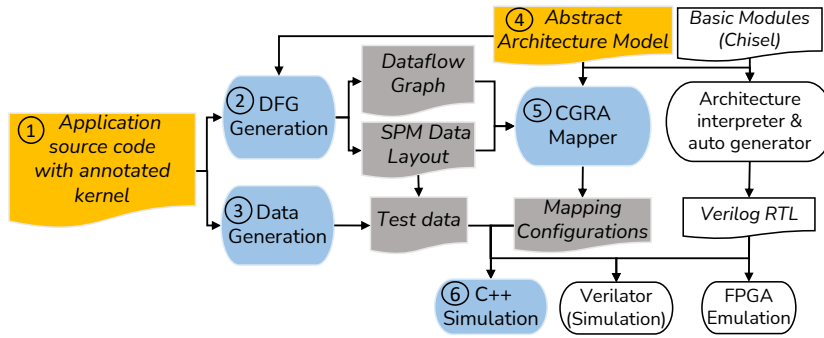


Figure 2.13: Overview of Morpher Framework, adapted from [82]

**CCF** [20] The **CGRA** CGRA Compilation and Simulation Framework (**CCF**) is an open-source toolkit that tackles **CGRA** generation in a heterogeneous **SoC**. It can handle branch divergence, sub-words, and pointer access in the embedded loop. The initial compiler-based parsing (using **LLVM** [48]) takes in a pragma-annotated kernel and collects the **DFG**. While the annotated code is encapsulated as a function call in the central **CPU**, it creates machine-type code for the **CGRA**, detailed in Section 2.2.1. The compiler executes the liveness analysis on the **IR**. It locates the necessary live-in and live-out variables that must be communicated to the processor (either through shared communication channels or Direct Memory Access (**DMA**)/memcpy exchange). The framework integrates the Gem5 simulator in system emulation mode to evaluate the final binary, allowing co-simulation with an external processor core, such as the ARMv7a mentioned in the work. The mapping algorithm uses RAMP (Resource-Aware Mapping for **CGRAs**, [19]), prioritizing minimizing the **II** and resource utilization. It iteratively explores routing scenarios (**PEs**, Register, Distributed **RFs**, and memory), modifies and schedules the **DFG**, and selects the best possible scenario for the final mapping configuration. It uses failure analysis to map previously unmapped **DFG** operations and dependencies into the **CGRA**. This method easily adapts **DFG**

characteristics and behavior onto the system's resources. Figure 2.14 describes the compiler's parallel steps to create the general processor code and the **CGRA** instructions with the aforementioned transformations until it reaches a final executable file.

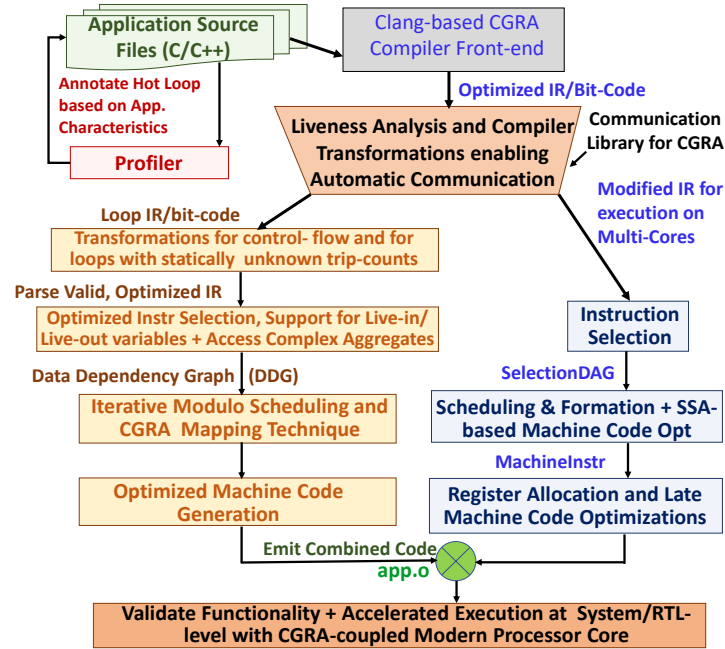


Figure 2.14: A High-Level Overview of **CCF**, adapted from [20]

**Pillars [35]** is an open-source full-stack framework that combines modeling, **RTL** generation, and a cycle-accurate simulation. The architecture description uses Chisel [8], a scala-based language allowing for high-level hardware design. This language specifies the fabric's blocks, subdivided into sub-blocks and elements (multiplexers, constant units, **ALUs**, **LSUs**, **RFs**), their data widths, and respective connections. The pipeline flow is aligned with other frameworks, as Figure 2.15 demonstrates: a **DFG** file and architecture file are fed into the pipeline, mapped, and scheduled, using **ILP** to map nodes to resources and topological search for synchronization, producing an execution model. In this case, the hardware description is interpreted and has its hierarchy flattened for the **MRRG** and a Chisel Top Design. The framework automatically generates auxiliary components (configuration and schedule controllers, synchronizers) that better suit the execution and dynamic reconfiguration of the device, allowing users to focus on high-level details. In step 13 of Figure 2.15, the Verilator backend takes the Chisel-generated **RTL**, creating a C/C++ model that used for the cycle-accurate simulation. Three stages compose the simulation: pre-process (load input data to the **LSUs** and top-level module reads configuration contexts), activation (simulation starts and auxiliary modules trigger), and post-process (extract data from the **LSUs**). Design verification is accomplished by verifying the top module outputs (during activation) or checking the data in the post-process phase. The authors present a case study that tests four variants of the **ADRES** architecture with torus connectivity and 32-bit data width, which are

physically implemented into an **FPGA**. The results report the frequency and area of the **FPGA** implementations and the percentage of successful mappings within 7200 seconds and ten random seeds, using metrics from [16]. However, the process failed to support multiplication operations in the most straightforward design due to insufficient multiplier units and connections.

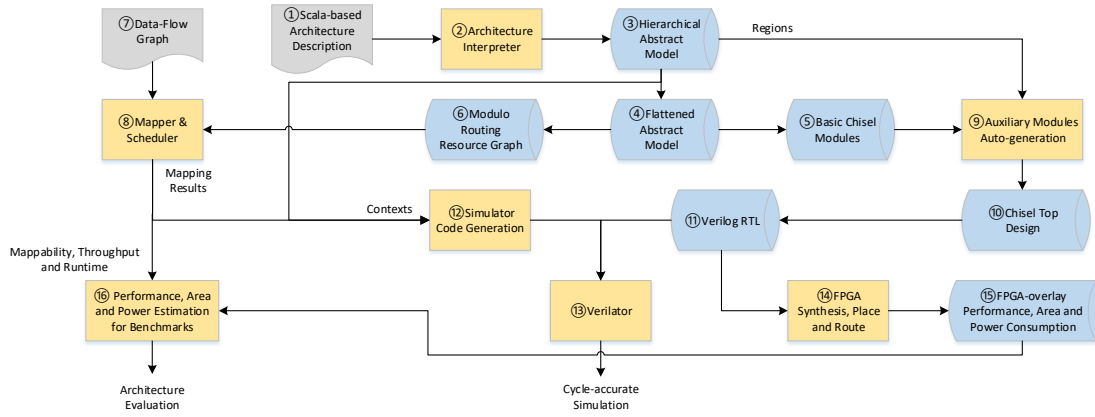


Figure 2.15: Pillars framework overview showing the main components, source [35]

**ML-CGRA** [32] is a compilation framework built with the Multi-Level Intermediate Representation (**MLIR**) infrastructure that provides performant **ML** models on **CGRAs**, compatible with **ML** platforms such as Open Neural Network Exchange (**ONNX**), PyTorch, TensorFlow. According to the authors, the generic compilation passes most frameworks employ to generate the **IR**, and the **DFG** loses useful high-level information about the kernel’s operation, adding overhead in operations dependent on tensor size and sequential data access. In contrast, **ML-CGRA** seeks to keep and optimize that information, providing an efficient direct offloading of the operations through higher-level code structures. The framework **CGRA** architecture supports **FUs**, and **RFs**. It includes a  $10 \times 11$  interconnect crossbar, control memory, and specialized units for *mul-add* and *max* operations, frequent in **ML** applications. The final design is set to deploy with a host platform, using **DMA** for communication. The fabric topology is a king mesh with eight routing connections, meaning each **PE** is connected to horizontal, vertical, and diagonal neighbors). **ML-CGRA** presents the *cgra-dialect*, with first-order representations and abstractions of **ML** operations, that the **MLIR** ecosystem uses to produce the Intermediate Representation. The framework implements **CGRA-Opt**, which includes three optimization techniques done in transformation passes: operation fusion minimizes data flow, merging tensor operations element-wise; loop tiling, which arranges tensors based on the size of the **CGRA**’s buffers and utilizes double-buffering input storage (minimizing the communication delay of the buffers); pattern matching recognizes chains of operations in the kernel that the **CGRA** template can directly accelerate, namely *mul-relu-add* or *matmul*, which can configure the device for these systolic array-based matrix multiplication; finally, offloading converts the final operation set into a function call for the **CGRA**. **ML-CGRA**

integrates simulation techniques from other frameworks (e.g., OpenCGRA) interfacing with a runtime library, CGRA-Runner, taking the function call and generating cycle-level and cycle-accurate RTL simulation. The authors took a Deep Learning model in the Linalg MLIR dialect and mapped it onto different CGRA configurations. The  $4 \times 4$  architecture was set as the baseline and evaluated in four experiments with progressive optimizations. The results demonstrate that implementing the entire ML-CGRA pipeline achieves a 215% speed-up over the baseline. The  $8 \times 8$  CGRA obtained a 90.17% and 123.61% speed-up (with and without double-buffering, respectively) compared to  $4 \times 4$  alternatives, validating the framework's scalability options.

SNAFU [32] emphasizes Ultra Low-Power CGRA Applications on edge devices, maximizing energy efficiency and the device's flexibility at the cost of area and performance. The authors identify common issues with other systems: standard programmable devices are inefficient, and ASICs cannot adapt to future implementations. SNAFU accepts a library of PEs and a CGRA topology description. Although only two arithmetic PEs are available, users can include a customized FU by adapting it to a generic SNAFU PE interface. The scheduling of the operations to PEs is implemented through an ILP algorithm. The framework also produces RTL code that standard CAD tools use to synthesize the architecture in a top-down approach from the high-level CGRA architecture to the placed-and-routed ULP design. In contrast, a bottom-up approach is more time-intensive, as individual units are separately synthesized and later combined, which is challenging considering the loop characteristics introduced by the multi-hop NoC. SNAFU extracts the DFG from the provided kernel code and creates the bitstream for the CGRA, which is issued by the co-processor upon starting the execution. These instructions include three control signals (communicated through the NoC) that order the CGRA to load a new fabric/bitstream, store a scalar value, and start execution. The RISC-V core is extended with the E, M I, and C extensions that enable the control signals. It employs several techniques to reduce energy spending. SNAFU's CGRA architecture is centered around vector-dataflow execution, which decreases the cost of fetching, decoding, and controlling operations and achieves higher parallelization and efficiency. Mapping supports, at most, a single operation into each PE, statically establishing routes to direct the data flow. SNAFU contains FUs with variable latency and, therefore, cannot implement static assignment and scheduling. Instead, it employs dynamic scheduling with an asynchronous cycle-independent dataflow and an in-order input stream. The NoC is bufferless and multi-hop, and no data duplication is employed since all intermediate values are buffered in the producer PE. The framework does not provide an integrated simulator, but the RTL code is compatible with many other external tools, such as Verilator and Cadence Xcelium RTL simulator. SNAFU-ARCH implements a  $6 \times 6$  mesh topology CGRA accelerator (based on SNAFU) with a scalar RISC-V core and 256 KB of on-chip Static Random-Access Memory (SRAM) memory. Compared to a RISC-V scalar core execution, this design achieves a 9.9 speed-up factor and 81% less energy consumption across ten embedded kernels with large input (e.g., sparse and dense 2D convolution, radix sort, FFT).

**Gemmini [31]** Gemmini (Berkeley’s Spatial Array Generator) utilizes the open-source Chipyard ecosystem, including the aforementioned Rocket core and Chisel hardware description language, composing a full-stack framework. The platform’s main focus is enhancing the performance of **DNN** computation tasks (e.g., inference), which represent a design challenge and constantly evolve with state-of-the-art research. Gemmini explores multiple internal architectural templates with different memory hierarchies and network topologies, which are flexible and adaptable to frequent **DNN** kernels (e.g., ReLU, Matrix Multiplication). The accelerator models a spatial architecture, connecting tiles comprising multiple **PEs** (with multiply-accumulate units) and memory registers, portrayed in Figure 2.16.

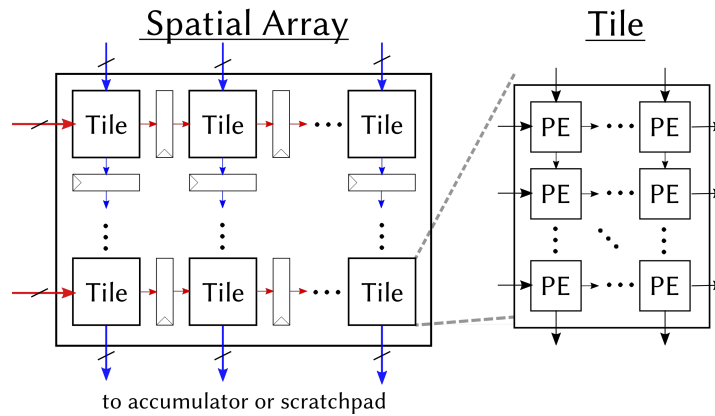


Figure 2.16: Microarchitecture of Gemmini’s two-level spatial array, adapted from [31]

The input is read from a scratchpad memory unit, and each **PE** includes local accumulators for intermediate results. It offers virtual memory support, hiding manual address translations and offering new possibilities to the end-user. It supports floating point data types, multiple reconfigurable dataflows at runtime, vector and systolic array architectures, and different types of **DNN** operators. Although it is not explained in detail, the mapping process uses heuristics to maximize data allocation in the system’s scratchpad. The framework determines the tile’s size, which controls how much data moves along the memory hierarchy in each cycle. Interface-wise, Gemmini offers its users a software stack that is capable of automatically reading **DNN** models through **ONNX** files, an open-source format, generating binaries, and mapping the respective operations to the accelerator. Additionally, it includes C/C++ **APIs** for low-level control of the accelerator. These options allow designers to explore different alternatives that better suit the domain, increasing their productivity. The framework emphasizes the integration and analysis of the accelerator into a **SoC** platform (as exemplified in Figure 2.17), considering that it is often the case of deployment in both cloud computing and edge devices.

In this case, devices can interface with other Gemmini-based accelerators and multiple RISC-V host **CPUs** using specific communication protocols and parallel data management. The software stack in the generated **SoC** includes a Linux-based operating system, which facilitates the debugging of realistic scenarios in a simulated deployment (e.g., context switches and page table

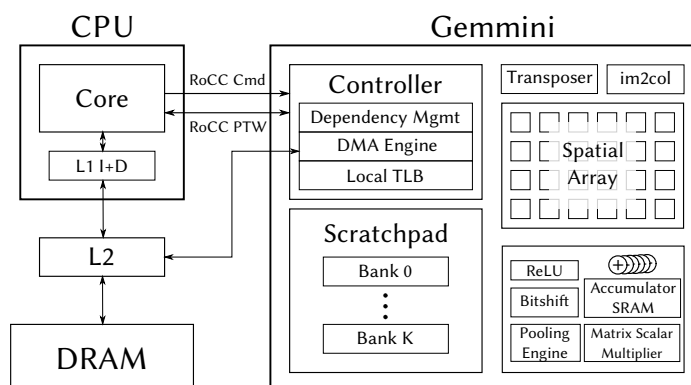


Figure 2.17: Gemmini hardware architectural template overview, adapted from [31]

evictions). The Chipyard ecosystem provides the FireSim **FPGA**-accelerated simulator to evaluate each design with a cycle-accurate simulation. In [31], Gemini is simulated and evaluated under different configurations coupled with two RISC-V cores (Rocket and **BOOM** [14]) across five **DNN** models (ResNet50 [37], AlexNet [47], SqueezeNet v1.1 [39], MobileNetV2 [71], and BERT [27]), with and without a specialized im2col convolution block. Results demonstrate that this unit performs better offloading to the accelerator, minimizing the computation on the **CPU**, and improving the overall execution. The im2col configuration with RocketCPU performs at 22.8 FPS at 1 GHz in the ResNet50 inference, which is a 2.670 speed-up over the implementation without the offloading block and a 1.130 over the **BOOM** [14] **CPU**. Results are similar to other platform accelerators (e.g., NVDLA) and out-compete with **CPU**-only executions, such as a 144 speed-up increase of the Rocket **CPU** BERT model implementation.

Two case studies highlight the flexibility of the framework in configuring system-wide behaviors. The first one displays the configuration of different virtual address translation patterns that tune the **SoC** and decrease the number of hit misses. Researchers customized the Translation Lookaside Buffer (**TLB**) to operate on two levels: a private accelerator **TLB** and the L2 cache level. In addition, they divided the **TLB** hits for page read and write operations into separate registers, which helped decrease **TLB** hit latency for successive page accesses. The other study tackled the resource partition challenge of the **DNN** domain, specifically the ResNet50, which balances convolutions with high data turnover and arithmetic intensity turnover, and residual additions, which take advantage of cache locality. Gemmini enables the exploration of different configurations, concluding that a scratchpad increases the performance by 10% in a single-core case, and extra L2 memory provides an 8% increase in dual-core.

**HEEPsilon** [2] & **HEEPocrates** [55] The Embedded Systems Laboratory (**ESL**) of the École Polytechnique Fédérale de Lausanne (**EPFL**) has two projects dedicated to **CGRA** platforms: **HEEPsilon** and **HEEPocrates**. Both extend the **X-HEEP** [56] platform, implementing new hardware modules in its structure, specializing in **ULP**. **HEEPsilon** deploys the open-source and lower-power OpenEdgeCGRA [2] design (shown in Figure 2.18b) with the X-HEEP micro-controller,

which, as previously mentioned in Section 2.1.1 enables several options for simulation and testing. It incorporates the SAT-MapIt [79] modulo scheduling compiler, which, from a kernel in a high-level language (i.e., C), outputs a valid mapping of the operations for the group of PE. The CGRA comprises a  $4 \times 4$  mesh with torus connectivity, a synchronizer (that schedules the operations in each PE, when possible), and a 2Kib context memory (that stores the instructions for run-time load). Memory transfers are supported in two methods: direct, which allows access to the memory positions for read and write operations, and indirect, with the necessary address as part of the instruction. This project also institutes a 32-bit instruction format representing a kernel with basic arithmetic and logical operations, with conditional branch support. The final ASIC design (completed as a 65nm low-power LVT TSMC technology node) occupies  $0.40\text{mm}^2$  and operates with a 250MHz clock cycle. The HEEPocrates platform follows a similar design, illustrated in Figure 2.18a, with a specialized emphasis on a ULP edge device for healthcare applications (e.g., using a Convolutional Neural Networks (CNN) for seizure detection [36]). It achieves maximum frequencies of 170MHz to 470MHz, with power consumption as low as 270  $\mu\text{W}$ .

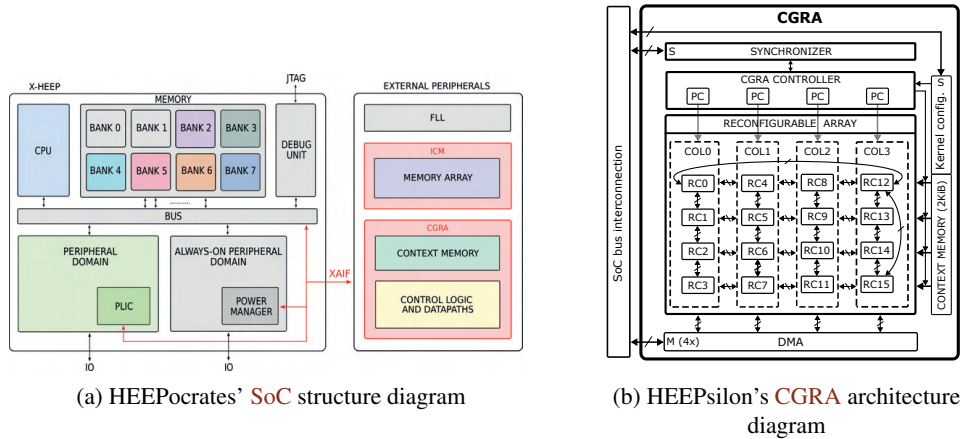


Figure 2.18: CGRA co-simulation platforms developed by the ESL, adopted from HEEPocrates [55] and HEEPpsilon [2].

## 2.3 Summary and discussion

### 2.3.1 General considerations about frameworks

The following Table 2.1 summarizes the current state-of-the-art frameworks available for CGRA generation. Information that is not clearly mentioned in the respective publications or not found in each repository description was left as N/A.

Most tools provide a complete generation pipeline that accepts an MRRG, extracts a DFG from source code, and then combines different mapping and scheduling approaches to design a model, usually in RTL. Each distinguishes itself from other alternatives by offering specific functionalities (e.g., PBT). The simulation stage is generally offloaded to specific tools that accept

Verilog or other high-level descriptions (e.g., SystemC). Not all frameworks support the same set of features, such as complete architecture configuration and floating-point operations, and most do not include any support for co-processing with a host platform.

### 2.3.2 Relevant features not present in most frameworks

Considering motivation expressed in Section 1.2, upon analyzing the broad spectrum of available frameworks, we can conclude:

- Most frameworks do not offer a co-simulation infrastructure, considering the **CGRA** as an isolated device. There are a few exceptions, such as Gemmini, that resorts to the Chipyard ecosystem, and others that occur from academic research that is not officially integrated into the framework (e.g., the thesis [49] that creates a RISC-V and **CGRA** hybrid system).
- Some require using lower-level hardware description languages (e.g., HEEPocrates [55]) to interpret and modify existing infrastructure.
- When co-simulation exists, compilation for the **CPU** and mapping/scheduling processes are seen as two separate parts, where the user or the host-processor needs to interact with the **CGRA** auxiliary components, such as memory or configuration controllers, storing instructions before starting the main computation. One of the limitations of current **CGRA** devices is the reliance on static compilation and setup of **PEs** and interconnections, making it challenging to schedule configurations dynamically, whose sequencing and synchronous communication have been established [51].

Throughout the document, we explore several approaches and implementations that tackle these shortcomings differently.

Table 2.1: Overview of different **CGRA** generation frameworks and their features, partially adapted from [82]. ✓ marks a included feature, ✗ otherwise, and N/A not specified by the paper.

| Feature                       | CGRA-ME | Pillars | OpenCGRA | CCF  | Morpher | Gemmini | ML-CGRA | SNAFU | ESL platforms |
|-------------------------------|---------|---------|----------|------|---------|---------|---------|-------|---------------|
| Explicit Control Divergence   | ✓       | ✗       | ✓        | ✓    | ✓       | N/A     | N/A     | N/A   | ✓             |
| Supports Recurrence Edges     | ✓       | ✗       | ✓        | ✓    | ✓       | N/A     | N/A     | N/A   | ✓             |
| Domain specific compilation   | ✗       | ✗       | ✗        | ✗    | ✗       | ✗       | ✓       | ✗     | ✗             |
| User-defined architectures    | ✓       | ✓       | ✓        | ✗    | ✓       | ✗       | ✗       | ✓     | ✗             |
| Multiple memory organizations | ✗       | ✗       | ✓        | ✗    | ✓       | ✓       | N/A     | N/A   | ✗             |
| Architecture adaptive mapping | ✓       | ✓       | ✗        | ✗    | ✓       | ✓       | ✗       | ✓     | ✓             |
| Data layout aware mapping     | ✗       | ✗       | ✗        | ✗    | ✓       | ✓       | ✓       | ✓     | ✗             |
| Heterogeneous PEs             | ✓       | ✓       | ✓        | ✓    | ✓       | ✓       | ✓       | ✗     | ✓             |
| Float-Point PEs               | ✓       | ✗       | ✗        | N/A  | N/A     | ✓       | ✓       | ✗     | ✓             |
| Cycle accurate simulation     | ✗       | ✓       | ✓        | N/A  | ✓       | ✓       | ✓       | ✗     | ✓             |
| Test data generation          | ✗       | ✗       | ✗        | ✓    | ✓       | ✗       | ✗       | ✗     | ✗             |
| Validation against test data  | ✗       | ✓       | ✗        | ✓    | ✓       | ✗       | ✗       | ✓     | ✗             |
| Focus on <b>ULP</b>           | ✗       | ✗       | ✗        | ✗    | ✗       | ✗       | ✗       | ✓     | ✓             |
| Co-processing Integration     | ✓       | ✗       | ✗        | ✓    | ✗       | ✓       | ✓       | ✓     | ✓             |
| Publication Date              | 2017    | 2020    | 2020     | 2018 | 2022    | 2021    | 2023    | 2021  | 2023/2024     |

## Chapter 3

# Proposed System Components

As stated in Section 1.3, there is a need to build a modular and extensible co-simulation environment that integrates a **CGRA** and a RISC-V Instruction Set Simulator (**ISS**), so that users can test the operation correctness of their **CGRA** designs. Our solution is a complete co-simulation infrastructure with a cycle-accurate **CGRA** simulator, denoted as the blue and green components in an encompassing platform shown in Figure 3.1. The remainder of this chapter outlines the design rationale, initial exploration of existing tools and frameworks, and decisions taken towards the rest of the implementation for all system components.

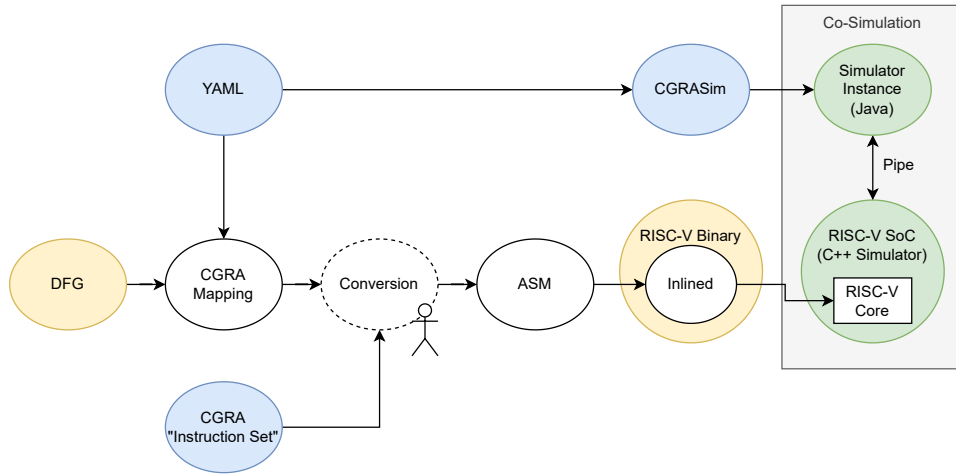


Figure 3.1: Proposed CGRA simulation platform. All relevant components discussed in the implementation appear in blue. The green elements represent components of the co-simulation. Input sources without direct user modification are colored in yellow, and the white shapes define an intermediate object in the compilation flow.

The main **CGRA** simulator is built using the high-level Java programming language, which, through a hierarchical Object-oriented programming (**OOP**) design, provides templates and interfaces for complete customization and extensibility of **CGRA** components.

The hardware architecture configuration of each simulated **CGRA** instance is defined by **YAML**

input files, which describe the structure that generates the internal simulation model (e.g., the number of rows and columns, the type of PE and the available inputs). **YAML** is a well-known format in Software Engineering (**SE**) and has been previously utilized for hardware modeling [74]. It delivers a simple and readable notation (based on keys and indentations) rooted in the concept that data models can be structured with lists, hashes, and scalar values. For the purpose of this work, it will model different characteristics and possible relationships between the elements of the **CGRA**.

The **CGRA** is operated with an assembly-type micro-code that can be fully delivered by the processor or partially from the compiler. Figure 3.1 displays a broadened view of how a given kernel and **CGRA** mapping eventually instantiates the co-simulation environment. Although it is not the focus of this work, it is expected that many of the compilation steps (i.e., partitioning a source kernel, extracting the information from its **DFG**, mapping and scheduling operations, replacing the necessary **CGRA** micro-instructions in the binary) will be fully automated, once the infrastructure of this project expands with added functionalities.

Information regarding the context at each cycle is stored in the device's memory and guides the accelerator's execution behavior. This micro-code is RISC-V compatible so that it can integrate general-purpose RISC-V compilation and communicate with external components. Both systems are considered loosely coupled, meaning the accelerator does not have the means to directly access the internal registers or **ALUs** of the host **CPU** but resorts to other modules capable of such operations. Instead, the interface based on custom instructions leaves it to the specific RISC-V core implementation, which supports the instructions, to control whatever underlying hardware exists in the system in order to exchange data and commands to the **CGRA**.

There are considerable decisions that affect the project's development and, thus, need to be considered, namely the type of execution model, the specific RISC-V simulator, and the micro-code specification.

### 3.1 Type of execution and memory access abstraction

The co-processing execution can be implemented in two separate approaches. One includes having a blocking processor execution while waiting for the offloaded results, and the other an asynchronous execution, in which an external controller and interrupts could alert the processor upon the **CGRA** finishing its assigned task. We have adopted a blocking mechanism approach to simplify development, focusing on the accelerator implementation.

Another important issue to manage regarding the co-processing is the memory access abstraction, or in other words, how memory is represented in the simulation and what role the RISC-V core assumes in this aspect. Several alternatives can be considered. The first, and presumably simplest, determines that the RISC-V processor offloads all relevant data to the parallelized algorithm into a storage bank located inside the **CGRA**, similar to the scratchpad included in the Gemini's [31] **SoC** architecture. The **PEs** can be configured to access the stored data, even allowing some data-streaming capabilities. Other options undergo some way of accessing the memory hierarchy of the RISC-V core/**SoC**. One approach includes the processor supplying the **CGRA** with

the base addresses of input and output data at the start of the execution, which are stored either on a scratchpad memory bank, directly to the respective **LSUs** or a single Address Generation Unit (**AGU**) that then propagates the data to the respective **PEs**, on a cycle-by-cycle basis. This option would require access to some level of **SoC**'s memory hierarchy. Another option would be to encapsulate the data using the communication protocol between the **ISS** and **CGRA**, using a command-like structure to guide the scalar values into the appropriate **PEs**. This approach would require an internal **CGRA** controller and an active/synchronous execution between both elements, which could add additional difficulty to the co-processing task.

Each solution has implications in the manner in which the algorithms are programmed and executed, the simulator's envisioned possibilities (i.e., the features of **LSU** elements), and the support offered by the chosen RISC-V simulator. Some options can be combined to provide added functionalities. To generalize and facilitate future integration in other co-simulation environments, it was decided that a hybrid approach is the most complete implementation, facilitating scenarios where we need to verify the simulator in isolation or with a host processor. In this approach, there is an abstracted Random Access Memory (**RAM**) (having all relevant data pre-loaded) within reach of the **CGRA** instance (accessed by each **LSU**). A scratchpad memory is also coupled for temporary storage and live-ins/live-outs. Coupling the simulator with the **ISS** makes it possible to rethink this implementation, such as **LSU** access to a certain shared memory controlled by the host machine (in the **SoC**) or other components of the memory hierarchy.

## 3.2 Choosing the RISC-V simulator

In Section 2.1.1, several RISC-V **ISSs** were mentioned that align with the desired objective of this work. With a few exceptions (i.e., **X-HEEP**), none are cycle-accurate, although the **CGRA** simulator is. This limitation was predicted, considering that the communication aspect of the system already invalidates this characteristic.

The aforementioned emulator variants from QEMU that support RISC-V (such as Qbox [26]) are not actively supported and require all system components to be implemented in SystemC. Therefore, they are not apt for the integration of our proposed approach.

A clear alternative would be the Spike simulator<sup>1</sup>, an open source RISC-V **ISS** maintained by RISC-V International, referred to as a "golden" reference in the simulator space. It is written in C++, modeling multiple hardware components as classes and internal circuitry through Object-oriented (**OO**) relationships. It includes the **CPU** model, a Memory Management Unit (**MMU**), and different levels of memory hierarchy while delivering multiple debugging tools, support for RV32IMAFDQCV native extensions, and complete documented extensibility that enables programmers to create new RISC-V extensions that, in the context of this work, would regulate the co-simulation process. Additionally, the simulator is integrated into the Chipyard [3] ecosystem, designed to generate and validate **SoC** platforms that can include external **RoCC** accelerators coupled to a Rocket or **BOOM** [14] core (using a specific communication protocol). However, given

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

that Chipyard is highly dependent on the Chisel Hardware Description Language (HDL), it is unclear how adaptable it can be to the current Java-based simulator. The most appropriate solution would be to use a Transmission Control Protocol (TCP) port or file descriptor to act as the communication channel between the CGRA and the RISC-V.

Other options resort to Transaction Level Modeling (TLM) with the SystemC library. SystemC is an open-source standard containing a set of C++ classes for functional hardware modeling [76], based on an event-driven model driven by a master thread that serves as an interrupt controller. It abstracts the pure logic descriptions of typical RTL models, written in lower level HDL, such as Verilog and VHSIC Hardware Description Language (VHDL). The library expresses the system's architecture in a high-level form, enabling faster development and verification processes despite sacrificing the timing accuracy of RTL. These characteristics make it a standard component of HW/SW co-design. TLM is an abstraction level that models system components as modules and encapsulates their communications in an abstracted data channel [57], usually a system bus that exists on the actual hardware. The modules work as concurrent processes that determine their behavior (through a set of states) and have access to TLM interfaces that enable data transmission in a master-slave transaction. The responsibility of the system's synchronization is assigned to at least two modules that create a deterministic simulation. This standard offers interfaces defined in System-C and isolates the duty of communication and computation into separate domains, easing the change of a given element or a re-arrangement of the system.

The previously mentioned Gem-5 [11] simulator supports the RISC-V ISA. It includes a modification that adapts the internal event-driven loop with a TLM interface to communicate with external devices [61]. Following the setup steps proved unviable since we could not generate an executable to start the simulator for the RISC-V build option. Another ISS that arises as a viable alternative is the RISC-V Virtual Prototype [38], as it accommodates a TLM interface and includes examples on how to use it with simple external peripherals. The drawback is that the documentation is relatively scarce, and there is no clear indication of how a possible RISC-V extension or modification to include the CGRA control code could integrate with its system.

X-HEEP [56] (previously cited) is also a well-established SoC simulator platform, present in the HEEPpsilon and HEEPocrates platforms. Given its recent creation, these projects demonstrate its capabilities and reliability in validating and completing a full-fledged ASIC design, with co-simulation and using up-to-date frameworks. It also provides vast documentation that instructs users on using the project's template and file structure to build new hardware modules and connect them to the processor. It offers Verilator as one of the simulation possibilities, which can be modified to comply with our goal using C++ instead of HDL variations.

Upon deliberation, it was decided that X-HEEP would be the preferred option for continuing this project, especially considering its very recent (2024) use in the HEEPocrates platform, where X-HEEP's eXtensible Accelerator InterFace (XAIF) is used to communicate custom instructions with a CGRA, akin to what we envisioned for this work. This interface can be configured to integrate multiple accelerators with different requirements, such as programmability, bandwidth constraints, and interrupt events. The supported cv32e40x core incorporates the Core-V eXtension

interface (**CV-X-IF**), which extends the **CPU** with custom extensions.

### 3.3 Developing the micro-code specification

The **CGRA** executes according to a group of instructions, similar to a processor's micro-code, planned to be developed in this work's context. It provides the mechanisms for the simulated processor to control the **CGRA** and perform the fabric setup to execute the required algorithm. Ideally, these instructions should be integrated into a RISC-V processor's compilation flow as long as the compiler and assembler support the extension. This could enable the use of the micro-code as an interface for other hardware generation frameworks that could use the simulator to verify their designs. The idealized co-processing behavior is intended as follows: the Instruction Set Simulator executes its general code, and when it finds a **CGRA**-specific instruction in the executable, it executes it by transmitting a given command or data to the accelerator (for the code segment that needs to be accelerated).

We implemented different layers of functionality into the micro-code: essential tasks for the configuration of the **CGRA** (e.g., establishing cycle contexts for the fabric, designating routes between **PEs**, loading constants from the **CPU** to the **CGRA**), execution behavior commands, and a query extension that will allow the co-processor to obtain information about the device it is coupled with (e.g. matrix dimensions) or for communication debugging purposes. More complex features such as Dynamic Partial Reconfiguration (**DPR**) (as seen in [45]) and data streaming are also considered for future work.

Researching how other platforms communicated and commanded the accelerator with their own developed micro-code Section 2.2.1 helped determine common patterns and functionality.

The OpenEdgeCGRA [2] architecture, although it includes a single instruction format, gives utmost priority to **PE** and **ALU** selection to route data and ensure the correct algorithm execution. In case of necessity, an address for indirect loads and stores can be encoded in the instruction. The **IXIAM** extension [67] includes a larger variety of instruction capabilities due to the wide range of accelerators it may include in its platform. This case demonstrates the usefulness of polling-type and mutual exclusion ownership instructions in a multi-accelerator **SoC**.

Adapting and creating a specification that suits our objectives with such unpredictable and constrained requirements (e.g., 32-bit size, multiple operands) required a careful analysis of the drawbacks of the different studied approaches. For example, specifying two **PEs** in a given instruction for routing can be achieved by an absolute or relative location definition. Each representation has drawbacks regarding the minimum bits required to provide relevant information and may even include edge cases. Similarly, branching instructions depend on the architecture of the **CGRA**.

### 3.4 Platform requisites

Taking on the objectives in Section 1.4, we devised the following user requirements that guided the project's development. The platform must:

- Accept a structured specification of **CGRA** parameters.
- Be controllable at runtime by processing incoming instruction words arriving from a host core.
- Be algorithmically correct for a given configuration specified by the instructions words and architecture description (producing the desired outcome).
- Support inter-operation with an external Operating System (**OS**) process sourcing the instructions.
- Produce cycle count metrics after a completed simulation run.

## Chapter 4

# Implementation of a Co-simulation System

The subsequent chapter delineates the two hardware components required for the co-simulation system. Firstly, we will introduce the **CGRA** simulator, along with the instruction set for its control, which was formalized as an RISC-V-like instruction set. This characteristic allows for compatibility with the second hardware component, i.e., the RISC-V **SoC**, which was based on a state-of-the-art platform (**X-HEEP**), modified to implement the communication between both components. Using the developed instructions, a RISC-V application can directly control and modify the **CGRA** device.

### 4.1 Micro-code specification

One of the main objectives of this work is to provide a new instruction specification that enables user-level control of the **CGRA** configuration, execution, and small query operations to retrieve information on the device. Given that most **CGRAs** operate in edge devices (e.g., **IoT**), they are often coupled with 32-bit architecture micro-controllers. As earlier noted in Section 2.1.1, **IoT** is also moving towards the RISC-V ecosystem [72].

#### 4.1.1 CGRA instruction definition

Our instruction set is constructed based on the custom instruction format of the RISC-V **ISA**, easing its adoption. This set of instructions is integrated into the RISC-V standard as a brownfield extension, which implies that it fits and does not overlap with existing encodings. Figure 4.1 presents the base format opcode table that indicates which available encodings can be used in the least significant bits of the instruction. Most of the space is already occupied, although the standard offers the *custom0* and *custom1* placements, which are employed in instruction-set extensions. These specific opcodes are avoided in current and future standard extensions.

Our format will use the *custom0* opcode with a "00010" start (little-endian notation) and two additional bits ("00"), given that the general 32-bit format for a RISC-V instruction includes a fully

|           |        |          |          |          |        |          |                |         |
|-----------|--------|----------|----------|----------|--------|----------|----------------|---------|
| inst[4:2] | 000    | 001      | 010      | 011      | 100    | 101      | 110            | 111     |
| inst[6:5] |        |          |          |          |        |          |                | (> 32b) |
| 00        | LOAD   | LOAD-FP  | custom-0 | MISC-MEM | OP-IMM | AUIPC    | OP-IMM-32      | 48b     |
| 01        | STORE  | STORE-FP | custom-1 | AMO      | OP     | LUI      | OP-32          | 64b     |
| 10        | MADD   | MSUB     | NMSUB    | NMADD    | OP-FP  | reserved | custom-2/rv128 | 48b     |
| 11        | BRANCH | JALR     | reserved | JAL      | SYSTEM | reserved | custom-3/rv128 | ≥ 80b   |

Figure 4.1: RISC-V base opcode map for RV32G and RV64G, adopted from [81]

|                       |    |    |    |     |    |     |        |             |    |        |   |        |   |        |
|-----------------------|----|----|----|-----|----|-----|--------|-------------|----|--------|---|--------|---|--------|
| 31                    | 27 | 26 | 25 | 24  | 20 | 19  | 15     | 14          | 12 | 11     | 7 | 6      | 0 |        |
| funct7                |    |    |    | rs2 |    | rs1 | funct3 | rd          |    | opcode |   | R-type |   |        |
| imm[11:0]             |    |    |    |     |    | rs1 | funct3 | rd          |    | opcode |   | I-type |   |        |
| imm[11:5]             |    |    |    | rs2 |    | rs1 | funct3 | imm[4:0]    |    | opcode |   | S-type |   |        |
| imm[12 10:5]          |    |    |    | rs2 |    | rs1 | funct3 | imm[4:1 11] |    | opcode |   | B-type |   |        |
| imm[31:12]            |    |    |    |     |    |     |        |             |    | rd     |   | opcode |   | U-type |
| imm[20 10:1 11 19:12] |    |    |    |     |    |     |        |             |    | rd     |   | opcode |   | J-type |

Figure 4.2: RISC-V base instruction formats, adopted from [81]

occupied 7-bit opcode (indicated in Figure 4.2). Therefore, each instruction starts with "0001000" in its least significant bits.

While some applications use a memory map operation behavior (e.g., OpenEdgeCGRA [2]), with the operation and their operands stored in the host device's memory and retrieved by the CGRA through a mmap/address-based process, we intend to create a packet-based communication system in which all the operations' information is included in the instruction, similar to a packet in a network system. The host device reads the starting opcode of an instruction in the binary ("0001000") and forwards the other 25 bits through specialized hardware (i.e., bus) to the CGRA, acting accordingly. Some trade-offs (such as the decreased amount of PE indexing) exist since 25 bits cannot offer much information space.

#### 4.1.2 CGRA Instruction Formats

In Figure 4.3, we can analyze the different instruction formats (for the most significant 25 bits) designed for the specification. The lower three bits designate the format, followed by four bits that indicate the instruction opcode (linear operations numbering). Each format is theoretically more suited for a designated action.

|              |    |    |          |    |    |     |      |
|--------------|----|----|----------|----|----|-----|------|
| 31           | 22 | 21 | 18       | 13 | 9  | 6   |      |
| operandb     |    |    | operanda |    | op | 000 | FMT1 |
| operand      |    |    |          |    | op | 001 | FMT2 |
| X (not used) |    |    |          |    | op | 010 | FMT3 |
| operandb     |    |    | operanda |    | op | 011 | FMT4 |
| imm[0:9]     |    |    | operanda |    | op | 100 | FMT5 |
| imm[10:31]   |    |    |          |    | op | 101 | FMT6 |

Figure 4.3: Instruction formats included in the specification for the 25-bit leftover bits.

- FMT1 is a general two-operand instruction.
- FMT2 has 18 bits dedicated to a single operand (e.g. set the execution mode).
- FMT3 ignores the 18 most significant bits for an operation that does not require an operand.
- FMT4 is designed to incorporate a smaller operand (e.g. the **ALU** control mode) and a larger one (e.g. **PE** index).
- Finally, FMT5 and FMT6 work in tandem to transfer a 32-bit constant to the live-in memory of the **CGRA**.

### 4.1.3 Instructions

A **CGRA** has two main stages of operation: the configuration of each cycle (i.e., the context) and the execution. This division is seen in Table 4.1, with most instructions operating to configure the **CGRA** before even starting its execution. In most devices, the configuration is stored in an internal memory, which applies the appropriate context at each step.

We start the context configuration stage of the device with the "Start Setup" instruction, which also determines one of the three context application behaviors: a fixed context for every cycle, an alternated ring-style application (iteration % context\_size), and a regular linear numbering. The "Start Context" instruction allows it to operate in the specified context, and a "Store context" (with a mandatory matching identification) closes it. In this step, the user can route different **PEs** to propagate information, connect **PEs** to scratchpad registers to retrieve or store results, and set an operation on a control-bit operated **PE**, such as setting the bit 1 in the **ALU** (add operation). Instructions that reference a specific **PE** are order dependent, requiring special attention to the internal architecture since they occupy the input and output ports of the **PE** from left to right as the instruction arrives (e.g., if the left input in a **LSU** is the address and the right is the value, the first route instruction connects to the address port and the second to the value port).

With "Start Execution", the user can start the **CGRA** according to three different modes: a manual step-by-step execution (requiring the host processor to send multiple "Step" instructions), an iteration-based execution (N iterations included as one of the operands) and finally a condition-based execution, that terminates the process when a pre-determined "flagged" **PE**, outputs True on a given condition (e.g., used to emulate a loop conditional statement). Before executing, we can store live-ins in the Scratchpad memory and retrieve the live-outs afterward. The specification also encompasses options to poll the **CGRA** and halt the execution, as seen in other accelerator applications, such as [67].

Currently, there are caveats regarding the two instructions presented in the table. The "Retrieve live-out" instruction would ideally require the **CGRA** to access one auxiliary register or internal memory to store the live-out properly. For now, upon receiving the instruction to retrieve a specific live-out, the **CGRA** forwards that value through the channel, making it the host's responsibility to use its hardware to store the value in the most appropriate location. This process could incorporate some encapsulation of the value (two-packet retrieval) instead of simply returning data without

|        |                          |           |    |           |        |
|--------|--------------------------|-----------|----|-----------|--------|
| funct7 | Address Reg              | Value Reg | 01 | rd        | 001001 |
| funct7 | Directly encoded Address |           | 01 | Value Reg | 001001 |

Figure 4.4: Additional store live-in instruction for the proposed RISC-V extension.

identifying the instruction that triggered it, adding robustness to the protocol in more complex execution scenarios (e.g., asynchronous communication with multiple live-outs that may arrive out-of-order).

The "Store live-in" operations resemble other pseudo-instructions for immediate values (e.g., *addi*) that require two steps to load the upper and lower parts of a 32-bit value, and therefore additional decoding logic (and intricate mechanisms, such as pipeline stalls). The host RISC-V processor must interpret, decode, and internally split the operation. In this case, this instruction has to be previously modeled as a single 32-bit instruction (ideally aligned with the rest of the specification since it has to be fully interpreted as such by the core) and associate the **CPU** registers that contain the value to be sent and the **CGRA**'s Scratchpad address. Since the R-type instruction in Figure 4.2 can accommodate two registers (one for the scratchpad address and the value), the extension may include an additional instruction for this purpose, with a different 7-bit value in its opcode, as in a variation of the *custom0* opcode or the two other additional bits (e.g., "01"). Supposing we are working with a customizable architecture and/or can alter its decoding process or customize a RISC-V extension format, it is possible to unify specific fields to directly encode the scratchpad address value while keeping a comparable overall instruction signature, as seen in Figure 4.4.

| Stage     | Name                                           | Behaviour                                                            | Type      | Instruction                          |
|-----------|------------------------------------------------|----------------------------------------------------------------------|-----------|--------------------------------------|
| Context   | Start setup                                    | Initiates setup of <b>CGRA</b>                                       | FMT2      | <ctx_mode> 0000 001                  |
|           | Start context                                  | Initiates configuration of context                                   | FMT2      | <idx> 0001 001                       |
|           | Route <b>PE</b> s                              | Setup connection between 2 <b>PE</b> s                               | FMT1      | <from_i> <to_i>0010 000              |
|           | Link <b>PE</b> to Scratchpad <b>RF</b> (STORE) | Connect a <b>PE</b> to a <b>RF</b> address                           | FMT1      | <addr> <from_i>0011 000              |
|           | Link Scratchpad <b>RF</b> to <b>PE</b> (LOAD)  | Connect a Scratchpad <b>RF</b> to a <b>PE</b>                        | FMT1      | <to_i> <addr>0100 000                |
|           | Set Control                                    | Specifies control bit of a <b>PE</b> (e.g. <b>LSU</b> , <b>ALU</b> ) | FMT4      | <PE_index> <CTRL> 0101 010           |
| Execution | Store Context                                  | Stores current configured context                                    | FMT2      | <idx> 0110 000                       |
|           | Start execution                                | Starts execution in a predetermined mode                             | FMT1      | <no_iter (opt)> <exec_mode> 0111 000 |
|           | Step                                           | Manually steps to next cycle                                         | FMT3      | 000000000000000000000000 1000 010    |
|           | Force stop                                     | Halts <b>CGRA</b> execution                                          | FMT3      | 000000000000000000000000 1001 010    |
|           | Store live-in                                  | Stores a 32 bit value into a Scratchpad address                      | FMT5/FMT6 | <imm(10)> <addr> 1010 100            |
|           | Retrieve live-out                              | Retrieves live-out from the Scratchpad onto the host                 | FMT2      | <imm(22)> 101                        |
| Query     | Select <b>PE</b> for Halt                      | Conditional halt based on <b>PE</b> boolean                          | FMT4      | <scratchpad_reg> 1011 000            |
|           | Verify current execution                       | Enables a poll boolean response of the <b>CGRA</b> state             | FMT3      | <PE> <X> 1100 010                    |
|           | No of <b>PE</b> s                              | Inquires for the number of units in the mesh                         | FMT3      | 000000000000000000000000 1101 101    |
| Query     | Device ID                                      | Inquires for the device's identifier                                 | FMT3      | 000000000000000000000000 1111 010    |

Table 4.1: Available instructions in the specification (ignoring the RISC-V prefix).

## 4.2 CGRASim simulator

To implement the instructions in the platform, we added valuable components to CGRASim that enhance its functionality. This section starts with a brief overview of the existing codebase, the new components (YAML parser, Decoder, memory access, execution control, and interface with the RISC-V simulator system), and how the current **X-HEEP** co-simulation works, ending with a description of scripts that facilitate the compilation of new programs.

### 4.2.1 CGRASim overview

The CGRASim platform is a **CGRA** simulator built in Java that allows users to validate their architecture and algorithms in this type of device. It takes advantage of the language's **OOP** abilities to create successive abstractions that enable increased customization. From a top-down perspective, the platform models a **CGRA** with three main components: the mesh, the interconnect, and a scratchpad (live-in/out storage). It can also be coupled with an external memory, accessed by the **LSUs**. The interconnect is responsible for applying and verifying the different contexts during runtime (e.g., routing connections) while propagating the values. The mesh aggregates the **PE** elements, operating them as a whole. These are divided into different types of binary and unary operating structures (e.g., adders, **LSUs**, **ALUs**, multipliers) that operate on abstracted data types (e.g., **PEBoolean**, **PEInteger**), storing results in ports that propagate values. Figure 4.5 represents a simplified diagram that details this implementation.

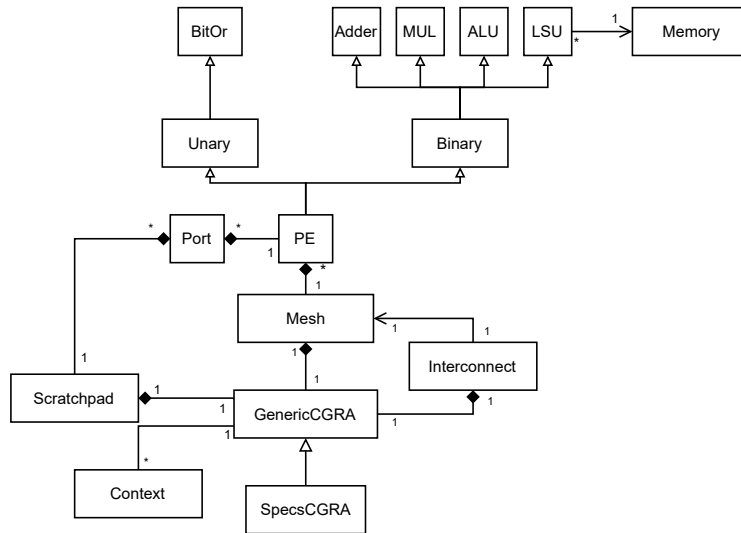


Figure 4.5: CGRASim simplified software diagram.

The primary functionality of a **CGRA** can be comprehended by implementing a step action in **GenericSpecsCGRA** as seen in Listing 1. During an execution cycle, the interconnect first propagates the previously computed or stored results to the next connected ports, and then the mesh executes each individual **PE** with the collected operands and pre-determined operation. This

procedure is repeated multiple times throughout the execution. Each **LSU** (seen in Appendix A) is assigned a *MemoryComm* call-back utility to abstract memory access operations in the execute stage of the step operation. As of this moment, it is configured as a 1-cycle latency operation.

```

1  /**
2   * Evaluates the equivalent of one clock cycle, propagating previous output values
3   *   ↪ to target PEs and computing new results for every active PE
4   */
5  public boolean step() {
6      if (!this.interconnect.propagate()) {
7          return false;
8      }
9
10     if (!this.mesh.execute()) {
11         return false;
12     }
13
14     return true;
15 }
16

```

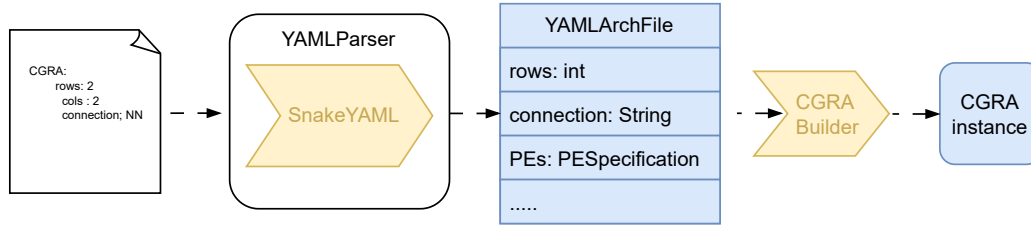
Listing 1: Step action in GenericSpecsCGRA.java

## 4.2.2 YAML parser

In this chapter’s opening, we proposed the addition of a **YAML** parser that equips the **CGRA** simulator to read an architecture specification from a document that complies with this format and instantiates the corresponding model. Despite Java not natively supporting **YAML** as an interpretable format, the *SnakeYAML*<sup>1</sup> library equips the program to read and de-serialize the whole data structure into a regular class instance. This flow is illustrated in Figure 4.6 Firstly, the *SnakeYAML* library uses Java’s reflection mechanism to de-serialize the file into a *YAMLArchFile* class with the rows, columns, memory size, connection attributes, and an array of *PESpecification* classes for the nested **PE** structures. This representation is then passed into a **CGRA** builder method that instantiates the class with the correct architecture, following a mapping of each name to the respective constructor. When creating a new type of **PE**, the user can add another entry (i.e., name to constructor pair) and, from that point on, use it in the **YAML** description file. The program makes the appropriate verifications, ensuring the specification does not incur any errors or incoherence (e.g., the rows and columns product needs to match the number of processing element structures, and no two **PEs** are defined for the same position).

Shown in Listing 2 is a two-by-two architecture example that the program can parse and instantiate. The **PE** specification may include the type of operation, its position in the mesh, and other attributes. In the future, other extensions, such as specifying the data type per **PE**, may be defined as an additional field in the **YAML**. Support for flexible datatypes is facilitated in CGRASim since **PEs** receives any class that implements *PEData*, meaning new data types can be defined as long as they implement the respective arithmetic methods that determine how operations are performed on that data type.

<sup>1</sup><https://mvnrepository.com/artifact/org.yaml/snakeyaml>

Figure 4.6: **YAML** parsing flow, using the *SnakeYAML* library

```

1 CGRA:
2   rows: 2
3   columns: 2
4   connectionType: NN
5   registerFileSize: 100
6
7   processingElements:
8     - { type: "ALU", x: 0, y: 0, latency: 1, memorySize: 1, RFenabled: True}
9     - { type: "ALU", x: 0, y: 1, latency: 1, memorySize: 1, RFenabled: True}
10    - { type: "ALU", x: 1, y: 0, latency: 1, memorySize: 1, RFenabled: True}
11    - { type: "ALU", x: 1, y: 1, latency: 1, memorySize: 1, RFenabled: True}

```

Listing 2: **YAML** architecture example

### 4.2.3 Decoder and Controller structure

To interpret the instructions sent by the RISC-V processor, the CGRASim platform must include the appropriate infrastructure to correctly interpret and employ such commands in the **CGRA** model. The step-by-step process of how an instruction is processed by the platform is illustrated in Figure 4.7. Succinctly:

1. The hexadecimal instruction is supplied to the *Decoder*.
2. The *Decoder* uses a parser that outputs each field in an array.
3. This array is converted into a *DecodedInstruction*, that contains the action to be executed by the controller.
4. The controller executes said action, modifying the **CGRA** model accordingly by calling the Java methods that de facto drive the simulation.

#### 4.2.3.1 Decoder Model

The Decoder model tackles interpreting the instruction sent to the **CGRA**, transforming a hexadecimal number into a more comprehensive structure (*DecodedInstruction*) that contains the necessary information to execute the command.

In order to parse each instruction, the decoder uses an existing in-house parser library capable of consuming binary instructions (in binary string or integer format) and decoding them into the respective fields and values. The binary parser receives a **JSON** file specifying the instruction formats to perform this interpretation. Since the CGRASim uses this library, extending the Decoder

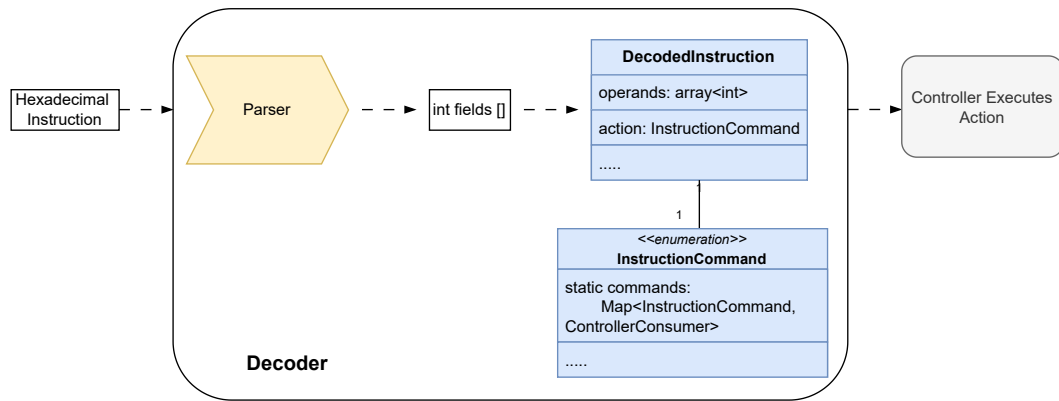


Figure 4.7: Step process to execute an instruction.

to support future versions of the **CGRA** instruction set is expedited. The document includes two main components, as seen in Listing 3:

- A list ("fields") of the named operands used in the different formats, with their index matching the placement in the output structure. This ensures consistency in how operands are referenced and utilized in the code-generation process.
- A list of instruction formats that define the different (underscore-separated) tokens consumed by the lexer, enabling a pattern-matching algorithm. Each operand/field is represented by its name and number of occupying bits.

```

1  {
2    "fields": [
3      "opcode",
4      "operanda",
5      "operandb",
6      "imm"
7    ],
8    "formats": [
9      "x(7)_operandb(9)_operanda(9)_opcode(4)_000",
10     "x(7)_operanda(18)_opcode(4)_001",
11     "x(25)_opcode(4)_010",
12     "x(7)_operandb(14)_operanda(4)_opcode(4)_011",
13     "x(7)_imm(10)_operanda(8)_opcode(4)_100",
14     "x(7)_imm(22)_101"
15   ]
16 }
17

```

Listing 3: **JSON** document that contains specification of the **CGRA** instruction format.

Following the parsing step, a *DecodedInstruction* instance is created, which, among other attributes, includes an *InstructionCommand* enumeration that is able to map the opcode to an executable command. This *InstructionCommand* enumeration is one of the few classes that require modifications if the instruction specification undergoes any alteration. It includes map entries linking the opcode numbering to Java Consumer interfaces (*ControllerConsumer*) that execute the corresponding actions.

#### 4.2.3.2 Controller class

To enhance the accessibility and manageability of the project, the **CGRA** instruction simulation process has been designed using the Model View Controller (**MVC**) pattern. In this architecture, the **CGRA** serves as the Model, while the *CGRAController* is responsible for operating and modifying its state. It encompasses a variety of methods that can actuate and alter the state of the CGRA model (essentially, all the instructions mentioned in Table 4.1), ensuring dynamic and flexible control. Listing 4 illustrates the five necessary steps to process an instruction, detailing the procedure from when the instruction enters CGRASim as a hexadecimal integer to its operation within the controller. During the context configuration stage, each configuration detail is initially created in a *ProvisoryContext*, which employs simpler structures that are subsequently transformed into a fully-fledged context upon starting the execution. In future iterations of CGRASim, if more complex scenarios are envisioned (e.g., having an instruction that reverses the last context configuration operation), these structures can be modified or reversed more quickly than possible with the final *Context*.

The controller is also responsible for managing the execution of the **CGRA** model according to the different instructed modes (enunciated in Section 4.1.3), with three main methods for controlling the simulated accelerator. Appendix B shows two methods of the *CGRAController* class, involved in context application and the execution of a kernel that depends on the halting condition of a **PE**.

While this behavior could be integrated directly into the **CGRA** model without the **MVC** design pattern, we find a similar decoupling of responsibilities (controller that actuates/coordinates the mesh of **PEs**) in actual hardware implementations (e.g., Figure 2.18b or the **CGRA-ME/RISC-V** hybrid system [23]). Architecturally, it improves the modifiability of the simulator's structure and opens new avenues for the functionality and role of the controller in the co-simulation.

```

1 Optional<DecodedInstruction> decodedInstr = decoder.decode(instruction);
2
3 DecodedInstruction instrObj = decodedInstr.get();
4 ControllerConsumer cgraControllerAction = instrObj.getControllerFunction(); //
  ↳ action to be executed
5 ArrayList<Integer> actionArguments = instrObj.getOperands(); // stored operands
6 controller.executeAction(cgraControllerAction, actionArguments); // execute action

```

Listing 4: The Java code that implements instruction execution

### 4.3 Co-simulation with X-HEEP

Before describing the integration of **X-HEEP** in the co-simulation infrastructure, it is essential to understand the implemented architecture that is considered and the role of the **SoC** simulator, illustrated in Figure 4.8. In this scenario, a host 32-bit RISC-V core, simulated through **X-HEEP**, commands the accelerator with instructions decoded in the binary (with the proposed RISC-V extension). The **CGRA**, simulated through CGRASim, uses an internal decoding mechanism to

interpret the instruction and configure or start the execution. Both elements have access to a shared memory that contains all data relevant to the kernel, and the **CGRA** has both a context memory that stores configurations for each cycle and a Scratchpad for temporary or fixed values (e.g., pointers) throughout the execution.

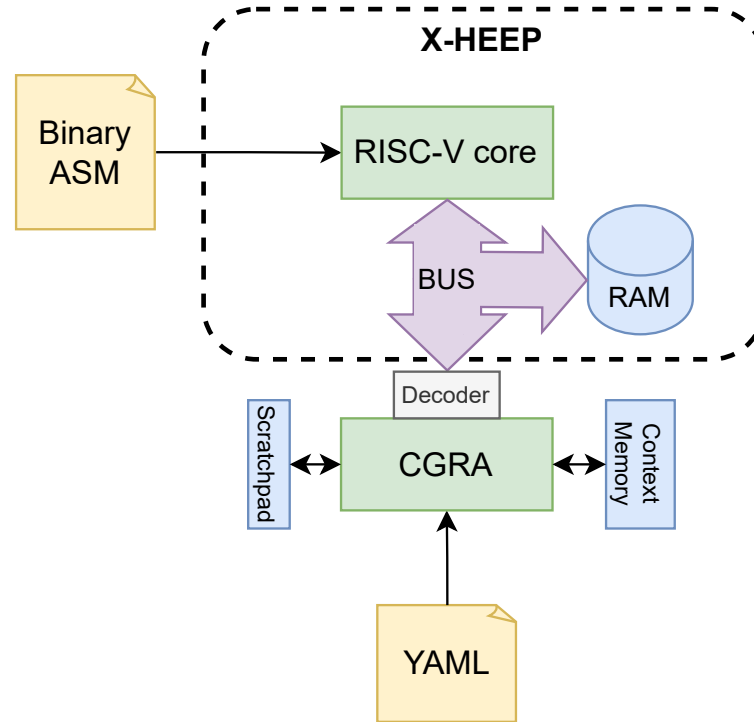


Figure 4.8: Diagram of the implemented co-simulation infrastructure.

As explained in this section, several compromises were made to achieve the final implementation, namely having the shared memory only accessible to the **CGRA** (given the current constraints and difficulty of including the memory in **X-HEEP**) and the bus system, which is available in the **SoC**, but is not capable of communicating with an external process.

As noted in Section 3.2, the **X-HEEP** platform was chosen as one of the co-simulation components. It uses by default the CV32E20 Ibex 32-bit RISC-V core (originally part of **PULP** [70]) and includes a variety of **I/O** and peripheral subsystems, suitable for embedded applications. No cache memory systems are present in the **CPU**. The official documentation for **X-HEEP** is extensive, and the project’s structures provide a template for adding new elements.

It specifies a bus system with the Open Bus Interface (**OBI**) [10], along with **EXT\_M/EXT\_S** ports for external communication and different types of peripheral access. It includes more complex protocols that use the existing hardware system, such as **XAIF**. However, **X-HEEP** is not designed to support interactions with other processes running on the **OS** level and is constrained

to hardware modules in the ecosystem to be modeled with *Verilator* or *Questasim*. These are constructed in *SystemVerilog* (or *SystemC*), integrated into the simulation, and with C interfaces accessed in user-defined programs.

To circumvent this problem, for a proof-of-concept objective, we opted to use the *Verilator* test bench as a mediator layer between the **CPU** model running in the simulation and the CGRASim process running in the **OS**. Similar to what is already achieved by the simulator's Universal Asynchronous Receiver/Transmitter (**UART**) that communicates with an external pseudo-terminal, we resort to *SystemVerilog*'s **DPI** [75] procedures that embed user-defined C functions into the Verilog designs, enabling the execution of C code defined else-where. The required setup to have this system running in **X-HEEP** was as follows:

1. Define new peripheral addresses for the **CGRA** interface in the *testharness.sv* and *testharness\_pkg.sv* files. Add new entries in the *x-heep-tb-utils.core* file, part of the build process.
2. Analogous to the **UART** module, create a new hardware component under the *ip\_examples* directory designating a "mock" **CGRA** device, with the sole purpose of overriding a write and read action on its peripheral memory address. The full *SystemVerilog* code appears in Listing 10. In it, use *initial begin* and *initial end* blocks for actions during the instantiation and closure (e.g., socket close). Use an *always\_ff* flip-flop block (seen in listing 5), executed upon a positive edge on the clock signal *clk\_i* or a negative edge on the reset signal *rst\_ni*. We can determine if the module is writing or reading from the address using a *reg\_req\_t* input port (*reg\_req\_i.write*) and verify if the data is valid. An output port of the same type retrieves an integer, the *rvalue* for a given assign expression in the code.
3. Create the necessary **DPI** procedures with the external communication programs (e.g., First In First Out (**FIFO**) channel, shared file, sockets, Inter-process Communication (**IPC**)) in the *dpi* directory under the main module directory. Create one function per method expected to appear in execution, as is the read in Listing 6. Use these functions in the **CGRA** *SystemVerilog* code. The implementation used for this project is seen in Listing 11.
4. Select a new *main.c* application, using Listing 12 as boilerplate, that would be turned into an executable and run on the RISC-V core.

The read operation call trace over the three primary files mentioned in the list's second, third, and fourth steps is illustrated in Figure 4.9. The complete flow of operations for sending an instruction from the **CPU** to CGRASim is represented by the sequence diagram in Figure 4.10. The primary executable file, shown in Listing 12, executes a simple array write or read (to the designated peripheral address), which triggers a **DPI** procedure elevated to the level of the *Verilator* test-bench. Then, a Linux Unix Domain socket channel, created when instantiating the *SystemVerilog* module, can exchange data (instructions and live-outs) between the two platforms, given that the Java process is running an **I/O** loop that awaits new instructions. A live-out is stored in the

<sup>1</sup><https://github.com/lowRISC/ibex>

```

1  always_ff @(posedge clk_i or negedge rst_ni) begin
2      if (rst_ni) begin
3          if (reg_req_i.valid && reg_req_i.write) begin
4              cgraitfdpi_write(ctx, reg_req_i.wdata);
5          end
6
7          if (reg_req_i.valid && ~reg_req_i.write) begin
8              reg_rsp_o.rdata = cgraitfdpi_read(ctx);
9          end
10     end
11 end
12

```

Listing 5: The *always\_ff* flip-flop block, that is executed on every positive edge of the clock

```

1  int cgraitfdpi_read(void *ctx_void) {
2      struct cgraitfdpi_ctx *ctx = (struct cgraitfdpi_ctx *) ctx_void;
3      int data;
4      int ret = recv(ctx->sock, &data, sizeof(data), MSG_WAITALL);
5
6      if (ret == -1) {
7          perror("recv");
8          cgraitfdpi_close(ctx_void);
9          exit(1);
10     }
11     return data;
12 }

```

Listing 6: The read **DPI** function definition.

output register (*reg\_rsp\_o.rdata*), which stores the return data from the read operation, allowing the running application to store it as a regular program variable.

Currently, it is possible to exchange data with a bi-directional communication capability (e.g., retrieve a live-out), but future work plans to expand the capabilities of this interface. This implementation assumes direct synchronous access in both devices, stalling and expecting the complete data packet from the socket stream. As demonstrated by the existing **UART** module in the platform, this process may represent a simplified version of how an actual hardware system would behave in a deployment environment, with considerations for asynchronous capabilities, includ-

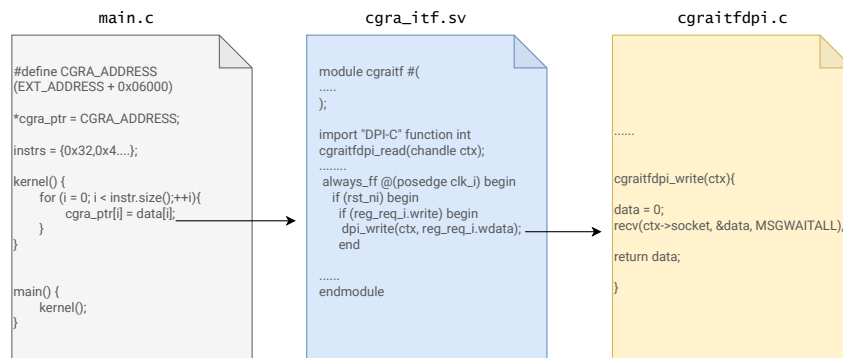


Figure 4.9: A trace over the three main operations that are executed when the main executable triggers a read.

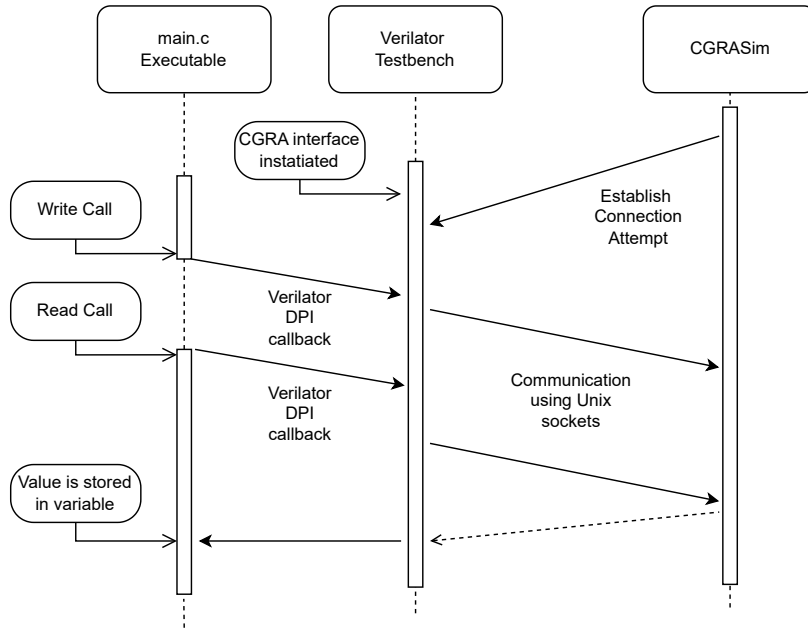


Figure 4.10: Verilator DPI callback sequence diagram.

ing access synchronization and/or a polling mechanism, and baud-rate constraints for symbol/bit communication. With this in mind, it is possible to modify the current interface *SystemVerilog* module to incorporate these elaborate processes. Additionally, the co-simulation could take advantage of the *EXT\_S* port in the bus system and detour to use the **DMA** peripheral<sup>2</sup> that includes several transaction-based mechanisms to communicate data between peripherals and **RAM** memories, with reduced **CPU** interaction. The **DMA** possesses a Hardware Abstraction Layer (**HAL**) interface that eases the process of configuring transactions and provides memory-safety checks in user applications. The appropriate timing constraints would enable a more realistic simulation of the **SoC**, where the main memory is located outside the **CGRA**'s reach (additional overhead for hardware mechanisms), at the expense of increasing total simulation time. Other more advanced communication options, such as the **XAIF**, are available to incorporate ports for more complex aspects of the communication (e.g., memory access, interrupt, power controls [56]). This aspect would enable a possible user-defined modification of the bandwidth capabilities of the **SoC** considering the **CGRA** simulation. One of **X-HEEP** projects, **HEEPocrates** [55], adopts **XAIF** for connecting the RISC-V to a **CGRA** and In-Memory Computing (**IMC**) modules.

#### 4.4 Helper Python scripts and mappers to create new programs

To execute kernels on the **CGRA** through the co-simulation and the micro-code, we evaluated several methods that could constitute a pipeline for the generation of valid **CGRA** mappings from

<sup>2</sup><https://x-heep.readthedocs.io/en/latest/Peripherals/DMA.html>

input code, that could help validate the system.

Creating examples that aid in developing the project proved difficult since no compiler is tuned to the current micro-code specification. To reduce the burden of creating new programs to run, we developed a simple helper script that converts human-readable comma-separated mnemonics (seen in Listing 7) for each instruction stored in a text file to hexadecimal code that can be directly fed into the CGRASim’s decoder for execution. The main script that composes this pipeline and an example input file with the hexadecimal translation is in Appendix D.

```
1 startContext(2) # generates -> 0x109
2 routeTwoPEs(1,5) # generates -> 0x50090
3 setControlBits(5, 1) # generates -> 0x50ab
4 setControlBits(2, 1) # generates -> 0x20ab
```

Listing 7: A small portion of a text-based input file that can be interpreted by the developed parser to create instructions.

One of the significant tasks of **CGRA** compilation is the mapping process, which, as discussed earlier, can be a complex and challenging chore typically performed by specialized algorithms. Currently, the platform lacks an automated compilation process, which is out of scope for the development of this work, making the manual execution of this step necessary—a significant hurdle in the development workflow. Open-source mappers can be employed to facilitate the execution of more complex algorithms. These tools optimize the mapping process, making it more efficient and effective. These take advantage of **ILP**, simulated annealing, and SAT-based techniques.

CGRA Flow [58] is integrated into the openCGRA [77] ecosystem and likewise uses **LLVM** to extract a **DFG** for the operations. The user can opt for its docker container, which offers the user a **GUI** (seen in Figure 4.11) with a visual dashboard for choosing the kernel, mapping algorithm (heuristic/optimal placement), **PE** capabilities and a step-by-step through each cycle.

The complete functionality of this mapper was not utilized in this research since the generated mapping does not enclose a compilation step that generates the configuration’s micro-code defined within our specification. Nevertheless, its visual output of the **DFG** and **PE** mapping provided a rough outline of how to map algorithms to the **CGRA**.

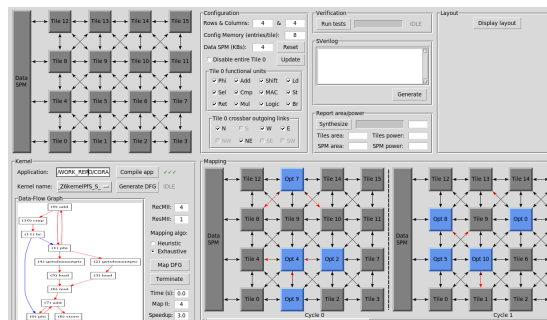


Figure 4.11: CGRAFlow’s **GUI** dashboard.



## Chapter 5

# Results and discussion

This chapter presents the results obtained from the functional analysis and verification of the platform. Firstly, it qualitatively evaluates the system’s operational capabilities, including usability and robustness. Secondly, it provides two test-case simulations that provide timing results, which are discussed to evaluate performance and scalability.

### 5.1 Functional Analysis

#### 5.1.1 Usability

Overall, the platform is relatively intuitive to use. Both components require a prior setup of the required dependencies. **X-HEEP** provides a step-by-step guide <sup>1</sup> in the official documentation, which details the installation of the RISC-V toolchain and Verilator, integrating them within the ecosystem. Most of the build system is done via makefiles and bash commands. CGRASim requires a dependency manager (i.e., Ivy) to complete the installation and some libraries created by the members of the Special-Purpose Computing Systems, Languages and Tools (**SPeCS**) <sup>2</sup> laboratory. The README file in CGRASim’s official repository includes manual installation and setup information.

In terms of running the system, **X-HEEP** is the one that requires the most steps to boot up the simulation (i.e., activating the python environment, creating the Verilator and **CPU** model, creating the executable from the user-defined code that is fed into the core and running the *Vtestharness* file from a pre-determined build directory). In our current system fork, the user can resort to the *run\_cgra.sh* bash file, which shortens many of these steps. The CGRASim simulator can run as a typical Java process. It is possible to use the *CoSimulationWithProcess* class by modifying its primary function and global parameters with the according architecture configuration file name, number of inputs, and socket address, from which the system connects and provides the **I/O** loop that retrieves, decodes and executes instructions. For the **X-HEEP** point of view, the user can embed the instructions within Listing 12’s code.

---

<sup>1</sup>[https://x-heap.readthedocs.io/en/latest/How\\_to/GettingStarted.html](https://x-heap.readthedocs.io/en/latest/How_to/GettingStarted.html)

<sup>2</sup><https://specs.fe.up.pt/>

Customizing the underlying structure or capabilities is achievable, building upon what was previously available in the code base, that is, the hierarchy of different PEs and their abstractions (shown in Figure 4.5). This structure facilitates the inheritance of properties and the creation of new units with similar functionality. For example, the subtraction PE differs from the addition PE in regards to the arithmetic operation, while the rest of the behavior (i.e., inputs) does not. Therefore, they can inherit from *BinaryProcessingElement*, stipulating their unique operation. The implementations in Section 5.2.1 use a unary increment unit to increase the iterator value. We created the new class *UnaryIncrement* in Listing 16, which derived from *UnaryProcessingElement*, with the appropriate operation for execution and added a new entry in the YAML parser map that connects PE names to their constructor. This setup enabled the utilization of this PE in the architecture file under the name "INC", in Listing 15. The code base is structured to facilitate access to components, divided into the parser, CGRA structure, platform, and simulation. Modifying the operability of the CGRA model (e.g., multiple reads per Scratchpad address) presumes in-depth knowledge of the sequence of steps during execution. Some actions demand alterations in the controller part of the code-base and the CGRA model, given that both need to be coherent in what they allow.

A user wanting to run simulation experiments with the current micro-code instructions must create new test benches with the correct instructions, memory inputs, and architecture files. Assuming the objective is to change the specific format, it requires the user to change the JSON that includes the formats and the operands' placement (listed in Listing 3, and add/modify the corresponding *DecodedInstruction* class that interprets the output of the initial parser for each type of format. The *DecodedInstruction* class child that interprets an FMT1 instruction can be analyzed in Listing 17. To add a new instruction with the supplied format, it is possible to create a new controller function and add/replace a new entry to map a new opcode to the correct action, as enunciated in Section 4.2.3.1.

Several if statement clauses have been placed throughout the code before executing operations and modifying the CGRA. In the event the clause fails, it throws an exception with the appropriate message to inform the user of an error or incoherence. These are found on operations such as the parsing of the YAML architecture file (if it does not comply with the proposed standard), in the instruction decoding stage (in case it does not match the specification, e.g., wrong operand) in the context configuration stage if any of the previously defined contexts is not acceptable for the final context memory (e.g., routing two PEs in discordance with the interconnect pattern) or during execution if any error that interrupts the step's iteration. For informational and functional purposes, the platform includes several classes of exceptions, namely *CriticalError* and *MinorError*, specifying the level of relevance of the error that has just occurred. This exception is propagated to the decoding and executing loop, where the user can select different ways of dealing with distinct errors, such as alerting but continuing the execution when a minor exception occurs and ultimately halting the program if a critical exception arises. Each added function should include the proper amount of if statements that verify its behavior in its body, triggering the appropriate exceptions in the event of an error.

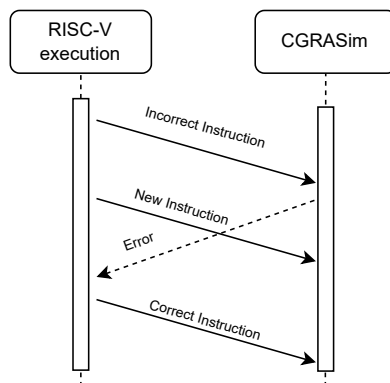


Figure 5.1: Communication protocol handling upon an erroneous instruction.

As mentioned in Section 4.3, the current implementation of communication is performed through UNIX domain sockets (IPC). Since the *Verilator* DPI call starts the socket, the Java process makes multiple attempts (e.g., by default, 20) at connecting to the simulated core at specified intervals until it exits. On the other hand, *X-HEEP* has a timeout setup (10 seconds) both on the socket (for read operations) and the initial socket accept (awaiting to receive CGRASim’s connect).

### 5.1.2 Platform’s limitations

Currently, no GUI is available to ease the simulator’s use, although it is considered for future work, incorporating previous research [22]. There is an extensive installation process for dependencies before utilization.

Section 4.3 details some of the limitations of the co-simulation infrastructure. Most of these limitations will be addressed in future work.

Firstly, the execution model only supports synchronous communication, where the CPU blocks pending any response from the CGRA, restricting more dynamic protocols and even some actions provided by the instructions in the proposed extensions. The halt and polling status instructions cannot adequately fulfill their actions since, during the execution, the CGRA cannot diverge from its operations. This functionality would require a separate thread with Controller interaction that would decouple the CGRA computation from the communication and decoder components.

The RAM is currently situated in the CGRASim (accessed by its LSUs), which does not demonstrate a fair and accurate simulation, given that the user needs to populate it manually and as it is not reachable to the CPU. Floating or fixed-point support has not been integrated into the platform or communication protocol. All operated values are integers.

Earlier mentioned in Section 5.1.1, changing the underlying micro-code specification and what it produces in the CGRA model assumes some knowledge about the inner workings of the CGRA simulator (i.e., the decoder and controller package). It demands internal modifications of the code

base, which moves the user from utilization to a code developer standpoint. This could deter users from customizing and increasing functionalities to the device, and, as of now, no work is planned to correct this issue. However, adding a GUI with configuration capabilities (e.g., create your PE) could be the key to ease its adoption.

Given the preliminary nature of the implementation, in which the RISC-V executable sequentially iterates over the instructions to send without interruption, the rest of the protocol assumes a correct communication behavior without any data errors, meaning if an instruction is wrong, the Java process and the Verilator simulation eventually exit. Given that an exception may be thrown in CGRASim in the event of a packet mismatch (e.g., the second packet of a two-packet instruction is lost in the communication, or a new context is started without having ended the previous one), it is possible to stipulate a backup operation, assuming a more dynamic execution (without a pre-arranged iterated instruction array) and run-time instruction generation. In this scenario (left for future work), the Java simulation sends an error code to the RISC-V core (with a more robust interface hardware module), relaying information to correct the error and communicate an instruction that follows or overlaps the previous one, illustrated by the Figure 5.1 sequence diagram. This network-based mechanism in the protocol requires additional examination since it has to incorporate asynchronous communication and state management functionality. In this case, the core may continue to send new instructions before receiving and acknowledging there was an error, where the protocol can contemplate discarding further commands or storing the current configuration, signaling the failed step, in order to repeat it. This process would also facilitate the communication once the shared-memory component is moved to the X-HEEP SoC, and LSUs have to halt and use an open channel to load and store values.

### 5.1.3 Testing new kernels

Testing and simulating the execution of a new kernel can be completed in the following steps:

1. The user designs the routing and mapping, writing a new instruction file with the mnemonics described in Section 4.4, and with the pseudo-assembler generates the instructions.
2. Choose the correct architecture and write the appropriate YAML file, placing it in the resource folder of the CGRASim project.
3. Customize the main.c program (Listing 12) converted to an executable by replacing the *instructions* array content with the new kernel. Additionally, if the core expects any live-out variables, set the *N\_LIVEOUTS* to the corresponding number and input the instruction in the *liveout\_instr* array.
4. Begin the execution of the Verilator simulation.
5. Begin the execution of the CGRA simulation as the *CoSimulationWithProcess* main function. Certain changes in parameters and operations may be required beforehand (e.g., create the correct memory layout, select the correct architecture file and the number of live-outs).

Two methods for pre-loading the memory of the **FIR** filter and dot-product already exist in the platform.

## 5.2 Experiments and benchmarks

### 5.2.1 Outline of the two kernels used for platform testing

This subsection describes the kernels used to evaluate our final work. Although some state-of-the-art mappers were tested to ascertain the viability of an optimal placement (with modulo scheduling) for each operation in the **PEs**, ensuring the minimum required steps to complete the total iterations, it was not possible to fully use them since some of these platforms were architecture-dependent. Further, there is no direct transcription from their configuration/output to our micro-code specification. Therefore, each kernel mapping was synthetically handmade with the intuition of being simple and testable, which required a thorough analysis of the kernels' behavior.

#### 5.2.1.1 Dot-Product

A dot product is an algebraic operation that takes in two number sequences of the same length and outputs a single value, primarily used for Euclidean geometry and angle operations between vectors. In mathematical notation, it is represented by:

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix} = a_1b_1 + a_2b_2 + a_3b_3 + \cdots + a_nb_n = \sum_{i=1}^n a_ib_i$$

This kernel is commonly portrayed in software programs as a for loop that iterates over the length of both arrays and multiplies the values into an accumulator. It can be succinctly decomposed into simpler operations by the procedural graph in Figure 5.2. In these experiments, for a  $4 \times 4$  mesh with nearest-neighbor connectivity, it was mapped according to Figure 5.3, which included an Increment unary unit, two **LSUs**, and all remaining **PEs** were **ALU** due to their interchangeability in regards to operation. The live-ins included the iterator and the accumulator, both started at 0, the memory access pointers for both arrays, and a loop termination value. An **ALU** unit was designated as a termination-ending **PE**, where the equality operation halts execution. The example assumes some platform constraints, which may be altered in future work (e.g., only one access to a scratchpad address per cycle, **LSU**s load latency is only one cycle).

#### 5.2.1.2 FIR filter

A **FIR** filter is a computation in **DSP** that computes a weighted average of the recent  $N$  input samples with a series of filter coefficients [85]. It can serve as a bandpass or stop filter, which

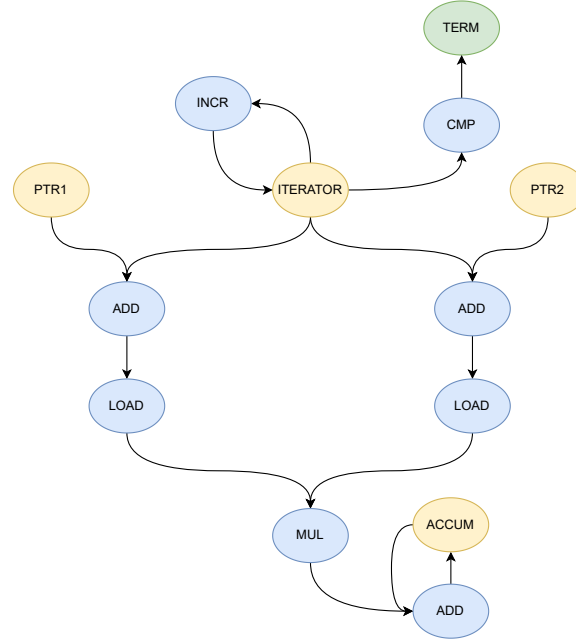


Figure 5.2: Dot product procedural graph. Yellow circles represent stored or input values, blue represents the operations, and green represents the termination event.

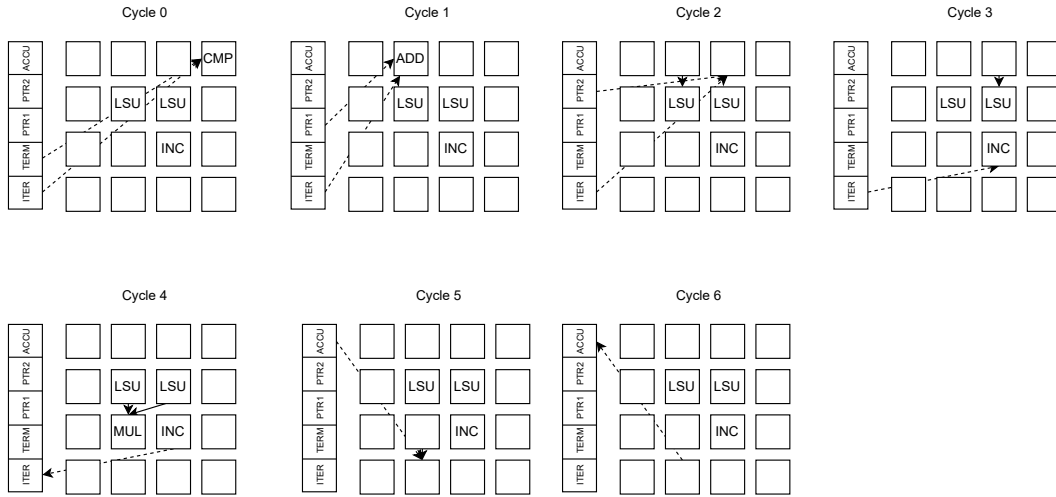


Figure 5.3: Dot-product mapping in a  $4 \times 4$  architecture

modifies input frequencies according to specified limits. Applications include audio-processing and image processing in the medical field [63]. In mathematical notation, a simpler FIR operation may be represented as:

$$\mathbf{y}[n] = \mathbf{h} \cdot \mathbf{x}[n] = \sum_{k=0}^{M-1} h[k] \cdot x[n-k] \quad (5.1)$$

Where:

- $\mathbf{y}[n]$  is the output at  $t = n$ .
- $\mathbf{h}$  is the vector of filter coefficients.
- $\mathbf{x}[n]$  is the vector of input signal values at  $t = n$ .
- $M$  is the order of the filter (number of terms in the operation).

The filter includes many parameters that can be modified to suit any application, such as the order that affects the number of coefficients and values in the tap window. The test kernel implements a filter with order 3, meaning that each accumulator sum includes multiplying three coefficients by the last two values and the new input. In this case, Figure 5.4 illustrates the operations that execute this kernel and how they inter-depend on each other. For a  $5 \times 5$ , each iteration was mapped in six CGRA cycles, as seen in Figure 5.5, with the following live-in variables: an iterator, its maximum value, pointers to the input, output arrays, and coefficients, as well as temporary registers to store the last two recently read elements. A  $4 \times 4$  mapping was provided for the same kernel, with ten cycles for each iteration. This represents a CGRA mesh with a reduced number of PEs and supposedly a smaller footprint since fewer LSUs are included. However, given that the mapping has to be more conservative with the number of utilized units and the orchestration of the propagation of values, the kernel requires more cycles to finish each loop iteration.

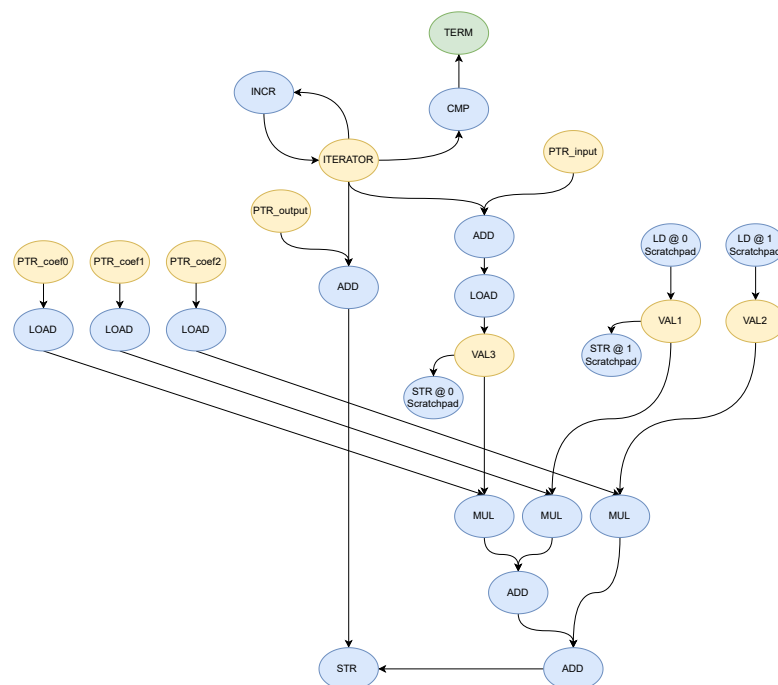
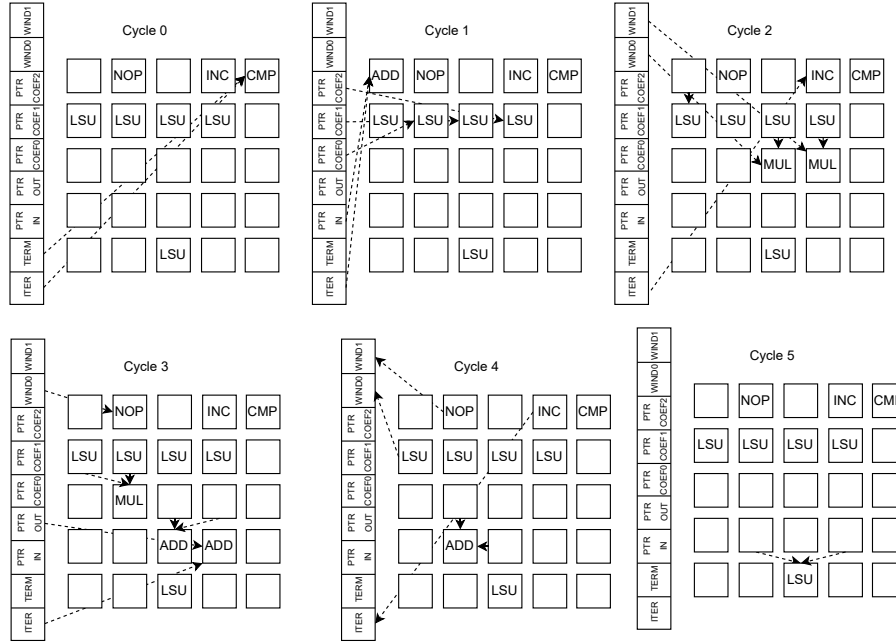
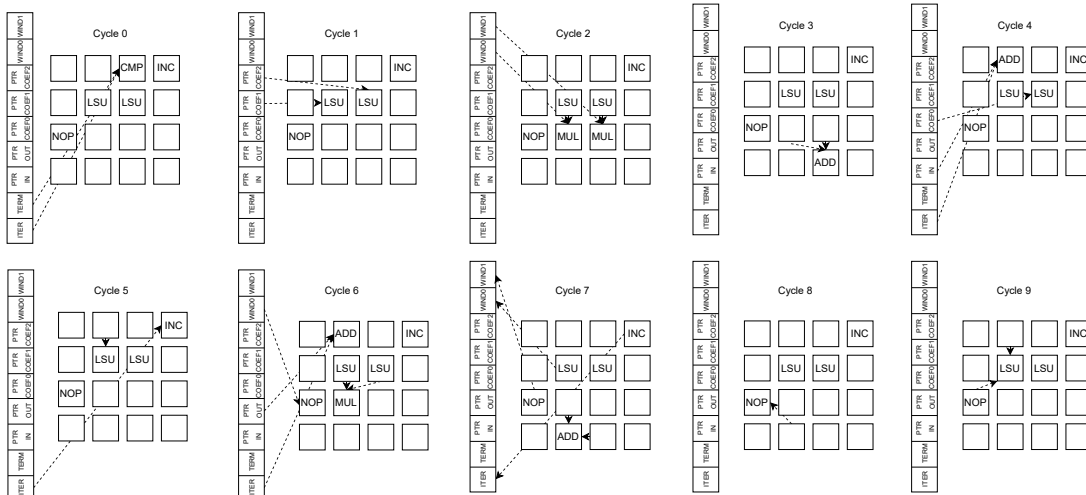


Figure 5.4: **FIR** filter procedural graph. Yellow circles represent stored or input values, blue represents the operations, and green represents the termination event.

Figure 5.5: FIR filter mapping in a  $5 \times 5$  architecture.Figure 5.6: FIR filter mapping in a  $4 \times 4$  architecture.

```

Address: 4 Value: 1
Address: 5 Value: 4
Address: 6 Value: 10
Address: 7 Value: 16

```

Figure 5.7: The output of the **FIR** filter with four input values.

### 5.2.2 Functional Verification

Each experiment's results were validated during each execution. In the case of the dot product, the test bench compares the accumulator's value (stored in address 4 of the scratchpad memory) to an expected value computed in a for-loop with the same input values. The **FIR** filter kernel is similar; however, since the output is a set of values, the test needs to compute the expected result and compare the entirety of the array with the memory output from **CGRA** execution. Additionally, the program outputs the memory content to the terminal (Figure 5.7), allowing the manual inspection of the results. The test benches are listed in Appendix G.

### 5.2.3 Co-simulation experiments methodology

To verify the components developed throughout this work, we collected different metrics from running the platform in two scenarios.

The first scenario (in Figure 5.8) considered the simulation in the isolated Java process. The program reads and instantiates the **CGRA** from the architecture file, timing the continuous **I/O** loop that reads the written instructions until an end-of-file and executes the kernel. The experiments for this scenario were performed within the eclipse Integrated Development Environment (**IDE**), which accounts for file **I/O** operations that read the instruction from a file (compared to a **IPC** socket connection) and additional **GUI** with debug output over-head.

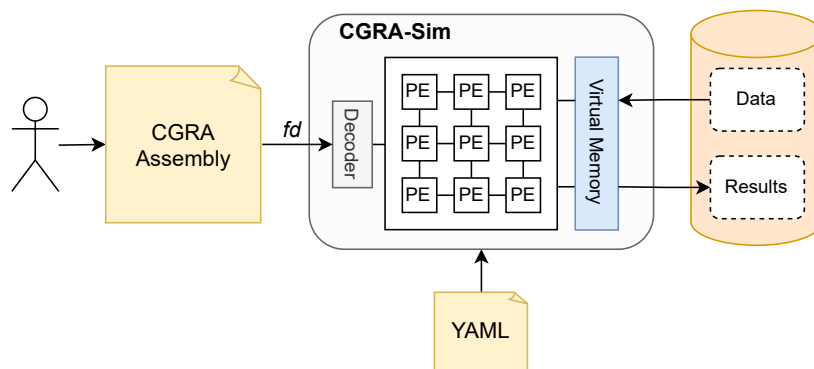


Figure 5.8: First verification strategy for the simulator

For the second scenario (illustrated by the diagram in Figure 5.9), we considered the full simulation with bi-directional communication between the **X-HEEP** and the **CGRASim** platforms. The instructions were stored as char arrays (one for configuration/execution and another for live-out retrieval) in the executable supplied to the **CPU** model, and a pre-determined function call

executes in the main scope. The program iterates the arrays and sends instructions over an **IPC** connection. If it needs to extract live-out variables, it issues separate requests followed by a **DPI** call that stalls until it receives the data in question. This scenario ran through manually activated scripts in two separate terminal emulators. It is important to note that, at the moment, the simulated **RAM** is located in CGRASim, which means any **LSU** action is contained in the Java process, as opposed to accessing a shared memory in the **SoC** which could add intricacy and delays. Each **LSU** operation has 1 cycle of latency.

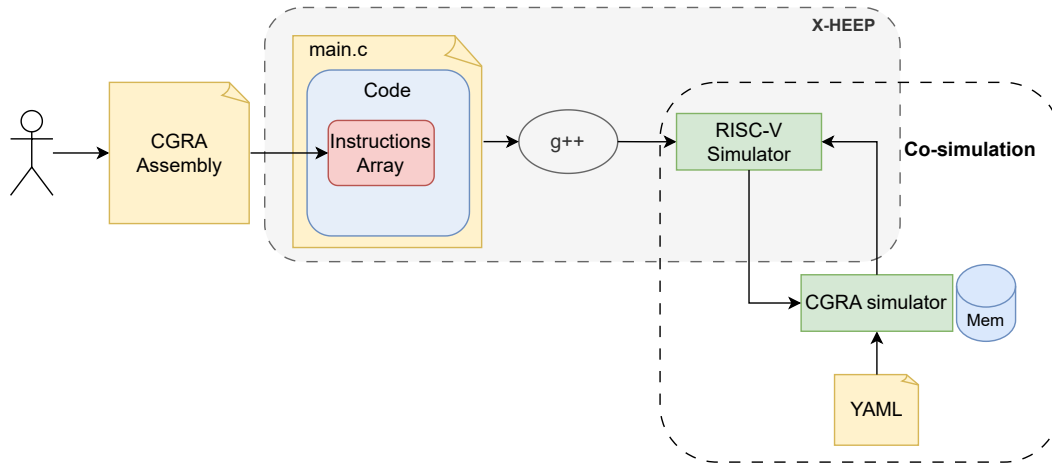


Figure 5.9: Second verification strategy, incorporating the co-simulation

## 5.2.4 Timing and computing metrics for platform

### 5.2.4.1 Output of each experiment

Each experiment yields the following results:

- A evaluation of the results correctness (unit-tests).
- The runtime of each process (the elapsed time it takes to complete the execution).
- A baseline case for a CPU-only execution (in the 64 input elements scenarios).
- The hypothetical simulated co-simulation time, gathered through a heuristic that will be explained.

### 5.2.4.2 In-depth explanation of the metrics

As previously mentioned, the project's scope is focused on the viability of this platform as a functional analysis tool. Therefore, the only timing metrics that are relevant to an end-user are those that pertain to the utilization of the simulator and that can impact the user's satisfiability. These include execution time for the first and second verification scenarios, revealing how it scales with

more considerable input and mesh sizes. The full co-simulation is challenging to time accurately, as it requires the launch of two separate processes. Still, the upper bound for the two can be considered the time it took to complete the co-simulation operation.

A theoretical total simulated time for each naive kernel implementation is also considered. As demonstrated in Equation (5.2), this value combines the total **CPU** simulated time and the **CGRA** simulated time. The **CPU** execution time is given by dividing the total number of cycles executed by the CV32E20 **CPU** model and its frequency, which in these examples is set at 100MHz by default. The *sim\_time* variable accumulates a count for all clock cycles in the top test bench of the Verilator simulation. This value is deterministic (independent of the computer's runtime conditions) since it is solely influenced by the number of instructions that result from the executable. However, this value does not account for the stalled **DPI** calls while waiting for the **CGRA** results. Since we cannot accurately measure its performance as a hardware module, this value is computed from the number of executed **CGRA** cycles divided by a hypothetical cycle frequency. The results in this scenario include **CPU**-only versions of the kernels (seen in Appendix H) to compare time-related effects.

$$\Delta t_{simulated} = \#CPU_{cycles} \times \frac{1}{CPU_{Freq}} + \#CGRASim_{cycles} \times \frac{1}{CGRA_{Freq}} \quad (5.2)$$

To avoid misapprehension, the following experiments, graphs, and discussions that evaluate the aforementioned value extracted from the formula will be mentioned using simulated time terminology. In contrast, the timing made to the co-simulation processes, precisely, the number of **OS** process seconds registered by the Java time package or the time Unix utility, use a runtime execution or simply runtime terminology.

Each experiment was performed four times, upon which we verified the correct output and collected the average time.

Although multiple mesh sizes are established throughout the results, it is essential to note that they represent the same kernel with added non-active **ALUs** (i.e., without performing any operation) to represent a bigger mesh. The only exception to this scenario is the step from the  $4 \times 4$  to the  $5 \times 5$  mapping in the **FIR** filter kernel, which, as mentioned, has two different mappings. Above the  $5 \times 5$  mesh, we encircle it with the inactive **PEs**. Ideally, additional **PE** availability could imply increased pipeline capabilities and decreased necessary cycles to complete the operation. However, creating such configurations for larger meshes without proper mapper and scheduler tools to facilitate this step was not feasible. Regardless, as seen in Section 5.3 it still returned insightful results.

#### 5.2.4.3 Test-bench specifications

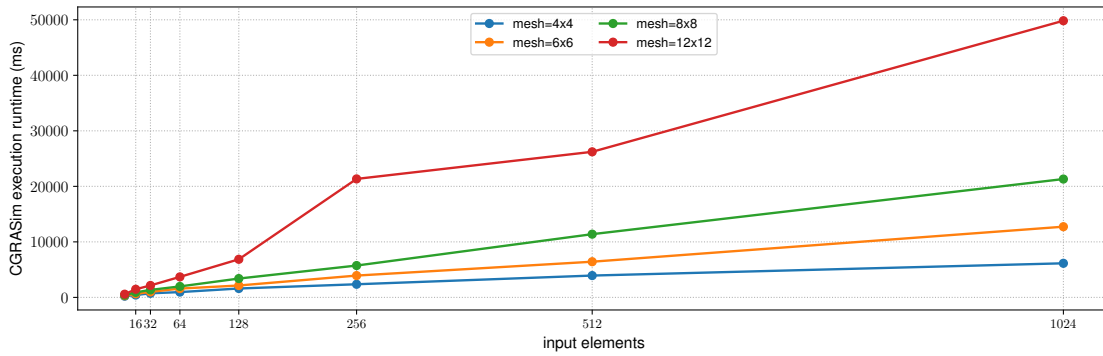
These experiments ran on a Ubuntu 20.04 laptop with minimal background process interference and with the following specifications:

- Processor: Intel i7-1065G7; 8 cores at maximum frequency of 3.9000GHz.

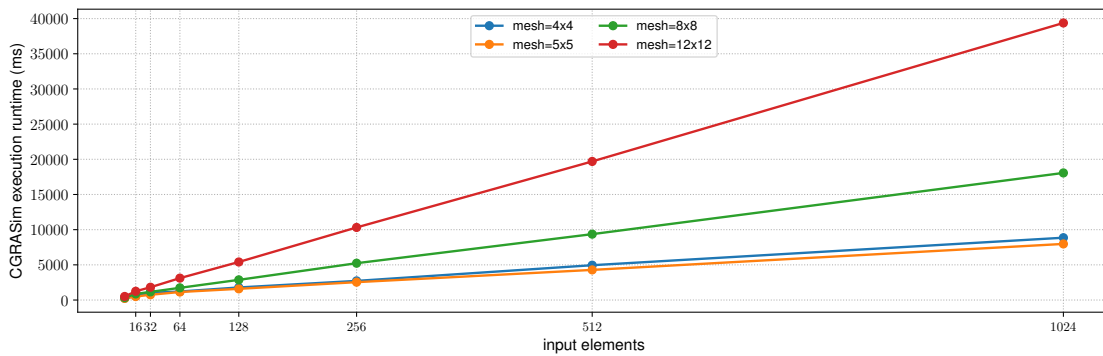
- 15665MiB of **RAM**.
- Cache system: L1d (192 KiB); L1i (128 KiB), L2 (2 MiB); L3 (8 MiB).
- Java Runtime Environment (**JRE**) version: 17.
- Verilator version: 4.210.

### 5.3 Results and Discussion

Both kernels demonstrated similar results regarding time performance and scalability. The average number of operations per iteration of the kernel is 1.8 for the **FIR** filter ( $4 \times 4$ ), 3 for the **FIR** filter ( $5 \times 5$ ), and 1.43 for the dot-product. Despite being the more significant subset of the instructions in both kernels, context configuration occupies the least time, especially with larger mesh sizes and input elements (above 256), compared to the **CGRA** kernel's accumulated time of the loop iterations.



(a) Dot-Product



(b) **FIR**

Figure 5.10: Variation of the simulation runtime depending on input size of arrays and different mesh configurations. Performed as the first verification scenario.

Considering the first verification scenario, where the instructions are stored in a file, and the **I/O** loop that executes instructions is timed, we can observe in Figure 5.10 that, intuitively, larger input

and mesh sizes tend to increase the duration of the simulation linearly. Although the difference is almost imperceptible initially, the interval eventually increases, and the  $12 \times 12$  mesh diverges from the rest of the tested sizes. One explanation for this is that a growth of the number of elements in the live-in and live-out variables requires more iterations to complete the kernel and in-memory allocations (from both data and instantiated objects), deferring to Java's garbage collection and memory mechanisms. Further, in the current implementation, for each propagate-and-execute cycle, the program iterates over each **PE** independently if its operation is required to complete the kernel. In case the **PE** is not configured with an operation or inputs, it completes an action that causes no side effects (a "dummy" operation).

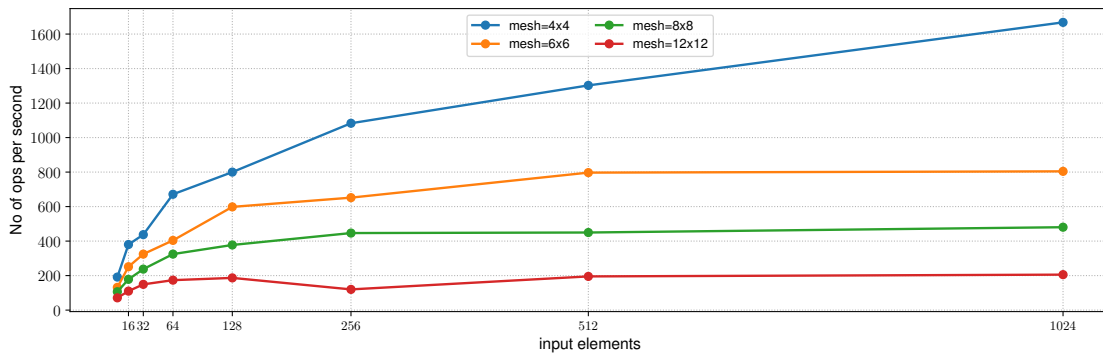
The  $4 \times 4$  and  $5 \times 5$  implementation of the **FIR** filter demonstrate the scalability of adding more **PEs**, as both lines plot near each other throughout the different input elements values. The  $5 \times 5$  configuration outperforms even though it contains nine units in comparison, albeit taking less time to complete an iteration. The lower simulation runtime throughout the graph means that this scenario rewards the trade-off between the number of **PEs** in the mesh and the pipeline capabilities it enables. In the other experiments, the unit increase represents no additional benefit or pipeline process. As referred, the other mesh cases (i.e.,  $6 \times 6$ ,  $8 \times 8$  and  $12 \times 12$ ) are merely the original  $4 \times 4/5 \times 5$  mappings, encircled by inactive **ALUs**. It would be possible to retrieve information from each instruction and notify the following step action that only certain **PEs** demand activation in a given cycle, but that would break the **CGRA** real-life hardware model and its intended behavior/purpose as an independent device.

Figure 5.11 includes two different plots that help to visualize the number of valuable operations (without counting the inactive **PEs**) that occur in each execution divided by the number of seconds it took to finish:  $\frac{\#operations}{s}$ . This is analogous to the Floating point operations per second (**FLOP**) measurement in typical computing devices, which performs as a throughput score. Figure 5.11a validates our argument concerning the dot-product: the  $4 \times 4$  mapping takes advantage of the given mesh resources, resulting in a higher throughput. The other mesh sizes simply occupy more of the simulation with inactive **PE** units (non-useful operations). Figure 5.11b demonstrates an equivalent outcome, except for the  $5 \times 5$  mesh, which achieves higher throughput than the  $4 \times 4$ , the latter not taking advantage of being smaller in terms of number units, with a lower **PE** activation burden, seeing that it requires more cycles to complete each iteration, without compensating the effort.

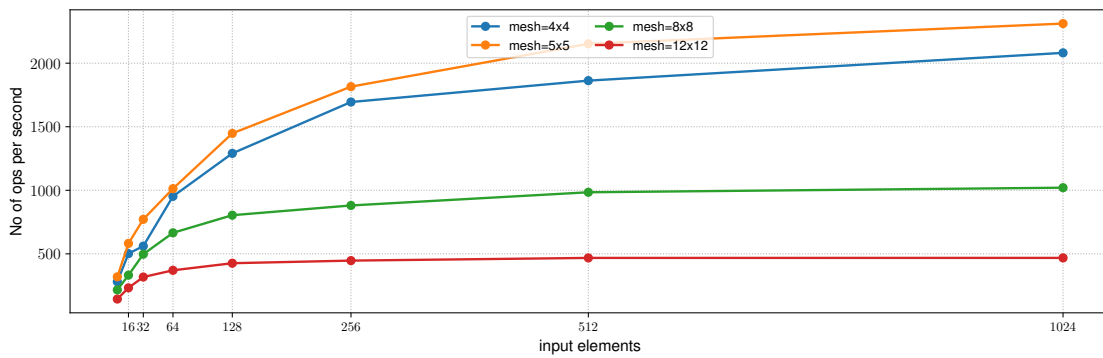
Table 5.1:  $4 \times 4$  and  $5 \times 5$  simulated runtime for 64 input elements.

| Kernel                         | CPU clocks | CGRA clocks | CPU-only clocks |
|--------------------------------|------------|-------------|-----------------|
| Dot-Product $4 \times 4$       | 5123       | 448         | 3623            |
| <b>FIR</b> filter $4 \times 4$ | 5623       | 641         | 9123            |
| <b>FIR</b> filter $5 \times 5$ | 5623       | 385         | 9123            |

Figure 5.12 plots each kernel simulated time for the dot-product and **FIR** filter, with 64 data points (two 64 integer arrays and one 64 integer array and 3 coefficients, for each kernel re-

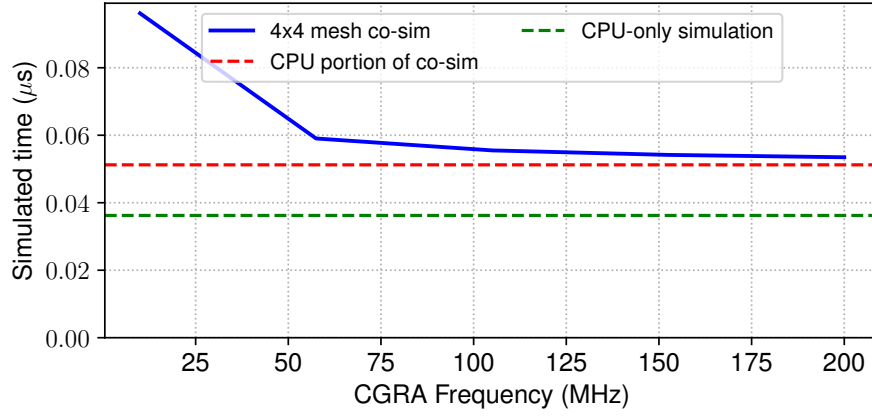


(a) Dot-Product

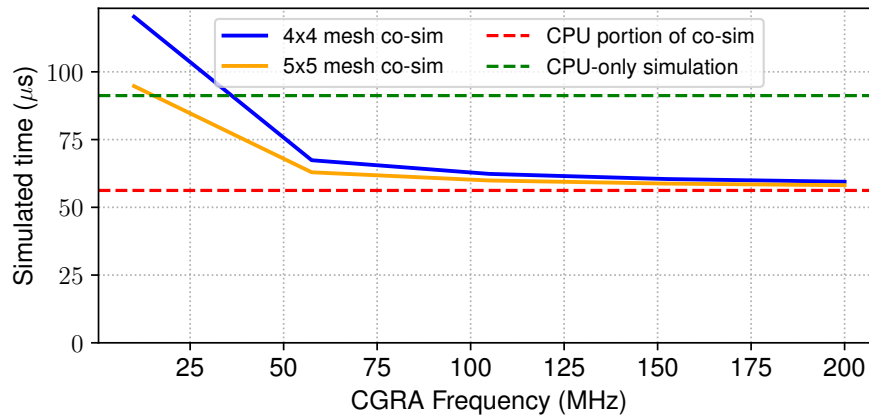


(b) FIR

Figure 5.11: Number of useful operations per second of simulation runtime for different mesh sizes and number of input data elements.



(a) Dot-Product



(b) FIR

Figure 5.12: Variation of simulated time (according to Equation (5.2)) for a kernel with two 64-element arrays and the  $4 \times 4$  and  $5 \times 5$  mesh sizes, depending on a hypothetical frequency for the CGRA accelerator. Additionally, the CPU-only execution (pure software implementation) is also plotted as the green line.

spectively). The results are available in Table 5.1. This time is based on Equation (5.2), which considers the **CPU** and **CGRA** cycles during the simulation. Figure 5.12a plots the dot product's  $4 \times 4$  and Figure 5.12b the  $4 \times 4$  and  $5 \times 5$  executions. The domain of the graph is an idealized frequency range that encompasses most of the frequencies found in synthesized state-of-the-art designs (e.g., OpenCGRA's [77] 833MHz and Morpher's [82] 488MHz architectures). The dot-product executes in 449 **CGRA** cycles and the **FIR** filter in 641 and 385 cycles (for the  $4 \times 4$  and  $5 \times 5$ ). This computed heuristic is experimentally far from an actual deployment measurement. However, it helps us understand how **CGRA** frequency variations affect the overall co-simulation span, implicating a decay curve when we increase the **CGRA** frequency, with an asymptote in the **CPU** simulated time. While the graphic shows that the dot-product may be best implemented as a **CPU** only solution (since the bottleneck is the core during the co-simulation), the **FIR** filter benefits from adopting a RISC-V and **CGRA** hybrid system right after the 50MHz **CGRA** frequency mark.

From Figure 5.13 and Figure 5.14, we observe an identical pattern: the co-simulation duration increases when more units are included in the matrix and with more elements as input. The  $5 \times 5$  mesh for the **FIR** filter kernel (in Figure 5.14c) results in lower co-simulation runtimes compared to its  $4 \times 4$  counter-part. The Java process finishes after the **X-HEEP** process, as it is actively waiting for instructions. Once it determines that the communication is completed (i.e., sends the last live out), the Java environment includes garbage collection and memory mechanisms that may delay terminating the process. This aspect should not impact the platform's usability as the simulation is finished. More extensive runtimes may affect the employment of the simulator from the user's perspective, particularly in more complex designs and kernels. The number of clock cycles in the **CGRA** is expected to decrease when using larger meshes, as the optimal mapping of a kernel will take advantage of the expanded footprint with more **PE**, decreasing the total duration. However, the future action of moving the shared memory to **X-HEEP**'s **SoC** necessitates that the **LSUs** utilize the synchronous socket channel connected to the **X-HEEP** platform, which could add overhead to memory accesses and increase the total runtime of the co-simulation. As a side note, the values for the CGRASim's simulation runtime are lower than the ones in Figure 5.10 (with the same number of elements). This is due to the experimental methodology described in Section 5.2.3. The 2nd co-simulation scenario is achieved through different terminal emulator scripts. At the same time, the first verification is reproduced inside the Eclipse **IDE**, with file **I/O** operations to read each instruction (compared to a **IPC** channel). Either way, both graphs illustrate the growth of the simulation time within the same order of magnitude, which is what we intended to demonstrate.

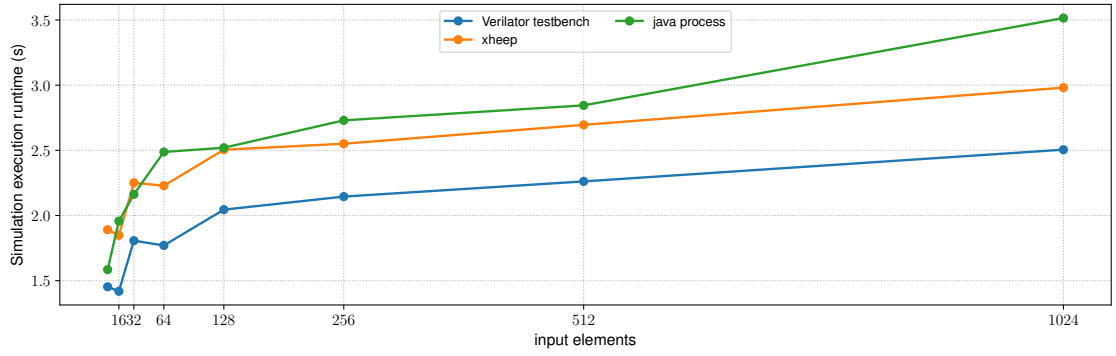
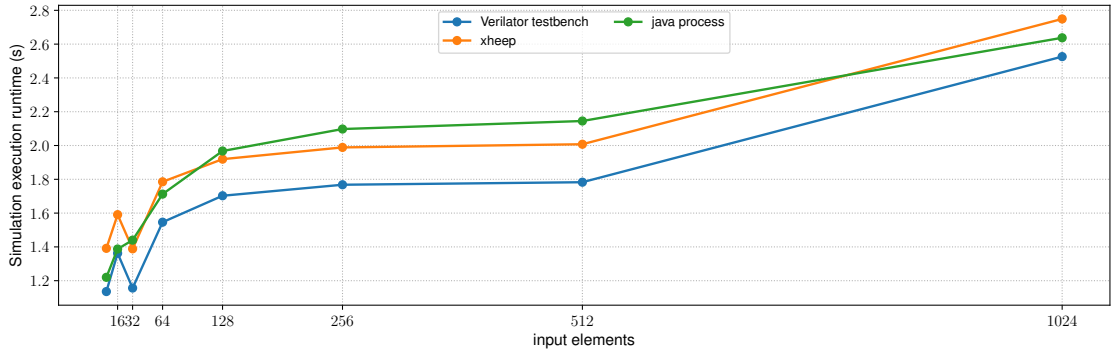
(a) Dot-Product  $12 \times 12$ (b) FIR  $12 \times 12$ 

Figure 5.13: Measured runtime for each kernel and a  $12 \times 12$  mesh size, according to the second verification scenario. The blue curve represents the elapsed time in Verilator's top test bench (*tb\_top.cpp*). In yellow, the time command measurement of the *Vtestharness* executable that results from the platform's build process and starts the **X-HEEP** simulation. In green is the Java CGRASim's process.

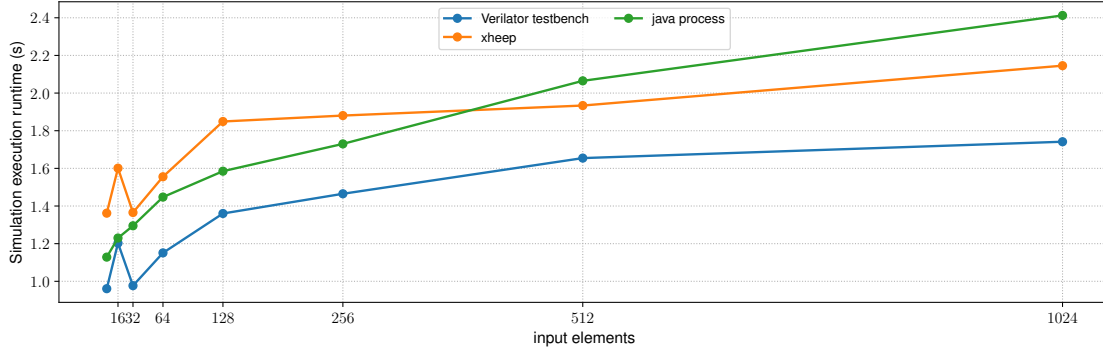
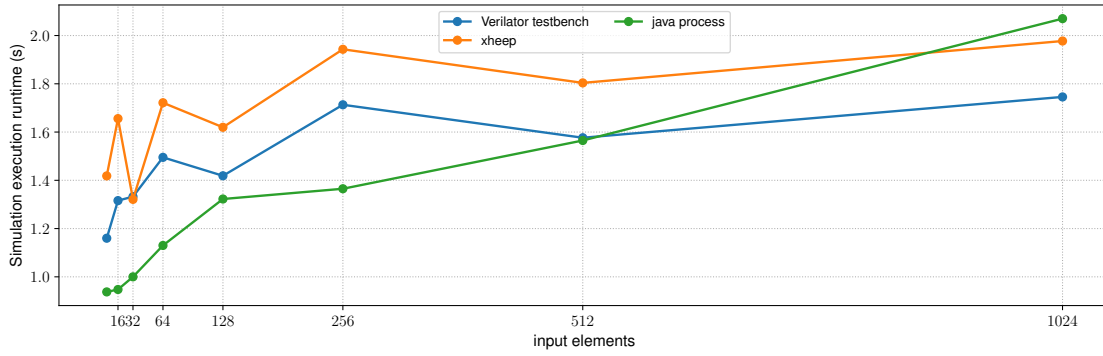
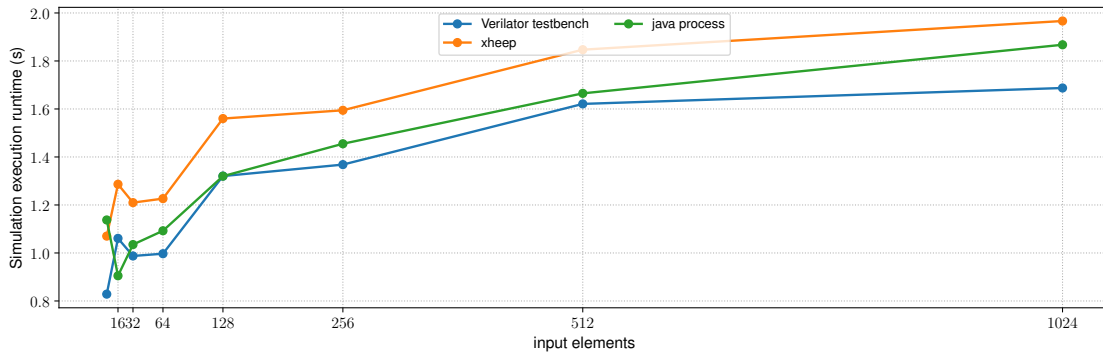
(a) Dot-Product  $4 \times 4$ (b) FIR  $4 \times 4$ (c) FIR  $5 \times 5$ 

Figure 5.14: Measured runtime for each kernel (with the  $4 \times 4$  and  $5 \times 5$  mesh sizes), according to the second verification scenario. The blue curve represents the elapsed time in Verilator's top test bench (*tb\_top.cpp*). In yellow, the time command measurement of the *Vtestharness* executable that results from the platform's build process and starts the **X-HEEP** simulation. In green is the Java CGRASim's process.

## Chapter 6

# Conclusion

### 6.1 Closing remarks

The current technological capacity to manufacture smaller transistor chips and increase their performance has imposed some limitations on the growth predicted by Moore’s Law [84]. This event has led to a renewed interest in heterogeneous architectures and reconfigurable accelerators for specific application domains, such as Coarse-Grained Reconfigurable Arrays. **CGRAs** are composed of small Processing Elements connected by different types of network topology, with countless applications, from Machine Learning to Health. **HW/SW** methodologies for systems development, such as simulation in Design Space Exploration, play a significant role in abstracting the design process and increasing the productivity of engineers and the subsequent quality of **CGRA** devices. This methodology, however, holds multiple challenges, such as the lack of an established research platform that can be easily integrated into the RISC-V ecosystem and provide an **SoC** overview.

The work described in the previous chapters assembles the required infrastructure for a co-simulation process that, through an extension set that applies to other compilation environments, enables a RISC-V Instruction Set Simulator to offload workloads to a **CGRA** simulator, named **CGRASim**. It integrates a decoder, a controller, and other utilities to promote a better co-design experience, enhanced by familiarity with the Java programming language, and achieved promising results in functionality. The platform was evaluated in two test scenarios: a dot product and a **FIR** filter loop.

Besides evaluating the runtime incurred from each process, the simulation’s recorded values can also present the opportunity to arrange theoretical scenarios that allow the users to analyze the benefits of a host-accelerator system, compared to a **CPU**-only execution.

Through this approach, we hope to contribute to the broader field of Design Space Exploration of **CGRAs** and propel the development of efficient and adaptable accelerators.

## 6.2 Future work

The developments in this work provide a solid foundation for future enhancements of the co-simulation. This section outlines potential avenues for further research and implementation, building on the findings discussed.

### 6.2.1 Platform functionalities

**Multi-threading execution** The primary focus of this platform is to produce functional executions of different kernels so that users may test their designs and mappings. For this purpose, the parallel execution of each **PE** in a given cycle is not required. However, micro-specification establishes some operations that require asynchronous responses, such as polling instructions that should be answered with the device's current status (e.g., configuring, executing, busy) and the halt command, that may occur during execution. If the **CGR**A executes synchronously, it cannot answer an *isBusy* poll until finished. Contemplating this issue, the platform's architecture could divide execution into the central **CGR**A model, operating independently of the socket handling and decoding mechanisms, enabling the receiver and decoding of commands in the meantime.

**Communication protocol enhancements** Most of the study concerning this subject was previously discussed in Section 4.3, which elaborated on the communication protocol and its current limitations. As of now, bi-directional communication for input-output retrieval in a pre-determined sense is supported in the co-simulation. In future implementations, it is possible to assume that the instructions will not be stored in an array but part of the executable arising from a mapper or specific kernel compilation. This dynamic behavior could enable more complex functionalities, such as particular error handlers if **CGR**ASim returns an error code during an operation. No baud rate or any other symbol-boundary considerations are considered in the current SystemVerilog module. Still, it would be an added benefit for the **DSE** aspect of the system, having this limitation as a configuration parameter to achieve more intricate scenarios or system specifications. A complete version of this system can also consider asynchronous communication with the **CGR**A that would permit more intricate scenarios (e.g., error handling) or even proceed with other computations until it notices the **CGR**A is finished (e.g., interrupt-based halting). Asynchronous communication does, of course, surface many network-related preoccupations that demand careful planning when improving the module. At the moment, the host processor simply iterates over an array of instructions it needs to send and awaits upon sending a live-out retrieval command.

**RAM hardware component** Another issue mentioned in Section 4.3 is the inability of the current system to utilize the memory-hierarchy of the RISC-V **SoC**, such as the **RAM** and **DMA** components. This would create a more realistic execution, increasing access costs, given that the shared memory would be located somewhere in the **SoC** instead of being paired within direct reach of the **CGR**A. The **CGR**ASim process would have to interface with the *EXT\_S/EXT\_M* ports in the bus subsystem (enabling the use of the **XAIF**) and possibly use the transaction-based **DMA**

mechanism to retrieve values. Each **LSU** instance could have a generalized interface enabling it to use this bus protocol, retrieving data stored in a particular address.

**FP operations** **FP** operations were considered in the motivation for this project, regarding the growing interest in **IoT DL** applications that require non-integer-based arithmetic operations, often running quantized models [52]. Currently, only integer values are stored and supported in the operations. Still, the foundation for more complex data types was already laid out in previous implementations of CGRASim through the *PEData* abstract type. This abstraction can enable heterogeneous applications by accommodating integers, floating and fixed points, or other data types. Further, new elements can be added to the system and should help to achieve this goal: a new supported data-type tag in each **YAML PE** component, conversion methods for each type in the communication protocol (e.g., interpret bits as float or integer) and modifications to the *SystemVerilog* module and **DPI** calls to ensure the proper handling. Additionally, the framework can include mechanisms for alternating between data types, such as an instruction to set a specific type as the default for subsequent live-in insertions before the execution begins.

**Compilation of the source kernel** The second verification scenario in Section 5.2.3 depends on a manual step that places the instructions from the output of the parser in two arrays that replicate the kernel loop offloaded to the **CGRA**, reflecting back to Figure 3.1’s diagram. Although not addressed in the context of this work, a specialized compilation pipeline, illustrated in Figure 6.1, could detect pragmas or in-line assembly to partition the general executable code (with a generalized compiler) from the accelerated code section/kernel that can be compiled into our format with a separate assembler tool, and embedded in the full executable binary.

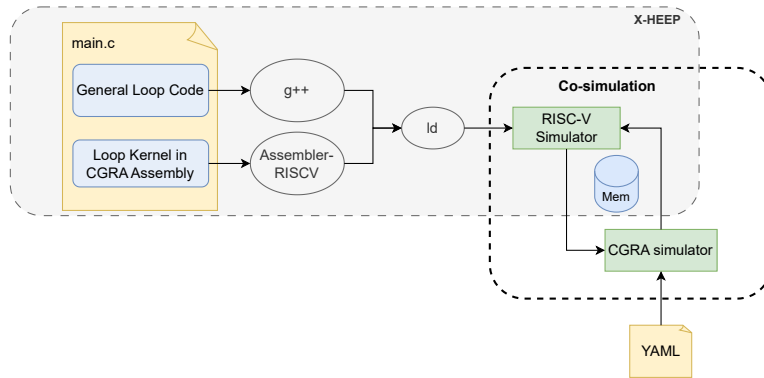


Figure 6.1: Workflow for a co-simulation scenario with specialized compilation.

**Replacing the trigger procedure of the communication protocol** Following the future compilation of the source kernel, the generated binary is provided to the **CPU** model, which executes the code and commands the co-simulation. This capability demands the extension of the base **ISA**

included in the **X-HEEP** platform (for example using the, with modifications in the decoding *SystemVerilog*'s hardware modules to interpret the instructions correctly. Alternatively, the OpenHW Group created the **CV-X-IF** [65], which extends a **CPU** with (custom or standardized) instructions implemented in a co-processor. It can be used to integrate standard RISC-V extensions without the need to modify the **CPU** decoder and pipeline infrastructure in the model. **X-HEEP** includes the cv32e40x RISC-V core which supports this interface. The memory-written instructions that are iterated through and sent to the **CGRA** will be replaced by the actual assembly in the executable. When the processor finds a **CGRA** opcode, it could use the available infrastructure of the **X-HEEP SoC** to forward them to the accelerator. **X-HEEP** includes the **XAIF**, adaptable to several types of accelerators and establishing state-of-the-art requirements (e.g., bandwidth, programmability, interrupts, and power modes). HEEPocrates [55] employs this interface for establishing the communication between the RISC-V and the **CGRA**, visualized in Figure 2.18a and Figure 6.2.

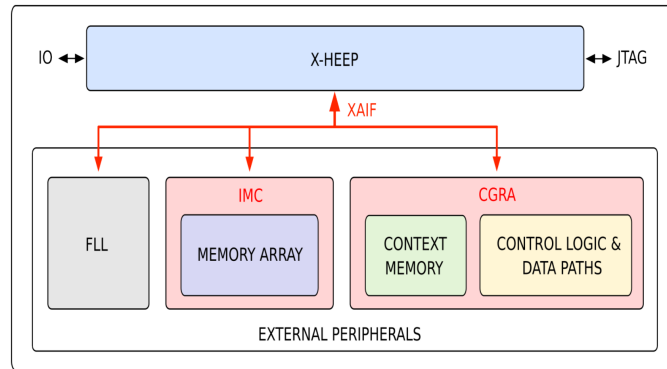


Figure 6.2: HEEPocrates architecture and **XAIF** communication that interacts with always-on peripherals and the accelerator, adapted from [56].

**Other improvements to the CGRASim simulator** Other minor modifications could widen the **DSE** of the co-simulation platform and increase usability. The implementation of the required context procedures to store and apply Prologue, Kernel, and Epilogue configurations increases compatibility with state-of-the-art mappers, in which the Prolog and Epilogue set initial and final stages for the executing (e.g., load, retrieve values, propagate data to determined locations). At the same time, the Kernel defines repeated behavior.

Other tasks include modifiable cycle-latency for each **PE** execution (e.g., determining that the **LSU** takes 2 cycles to complete a load, and the result is only provided thereafter), refactoring current Java classes (e.g., central **CGRA** model) to assure clean code practices, adding pre-made architectures described in **YAML** (e.g., **ADRES**) and the execution of more test cases (e.g. matrix multiplication), validating the generalization capabilities of these tools.

### 6.2.2 GUI interaction

Previous research in [22] that worked on this topic had created a user-friendly GUI application, built within CGRASim's Java process, which allows users to visualize the PE mesh, stored and propagated data, configure the array and manually step the device. This increases overall usability since the user does not depend on terminal-based output to collect information and debug the execution. This interface was not integrated into this work. However, it can be extended to incorporate information regarding the communication protocol with the X-HEEP core and the *Decoder* input and output values.



# References

- [1] Ensieh Aliagha and Diana Göhringer. Energy efficient design of coarse-grained reconfigurable architectures: Insights, trends and challenges. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–11, 2022.
- [2] Rubén Rodríguez Álvarez, Benoît Denkinger, Juan Sapriza, José Miranda Calero, Giovanni Ansaloni, and David Atienza Alonso. An open-hardware coarse-grained reconfigurable array for edge computing. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, page 391–392, New York, NY, USA, 2023. Association for Computing Machinery.
- [3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.
- [4] Hela Belhadj Amor, Carolyn Bernier, and Zdeněk Přikryl. A risc-v isa extension for ultra-low power iot wireless signal processing. *IEEE Transactions on Computers*, 71(4):766–778, 2022.
- [5] Jason Anderson, Rami Beidas, Vimal Chacko, Hsuan Hsiao, Xiaoyi Ling, Omar Ragheb, Xinyuan Wang, and Tianyi Yu. Cgra-me: An open-source framework for cgra architecture and cad research : (invited paper). In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 156–162, 2021.
- [6] Krste Asanović and David A Patterson. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.
- [7] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.
- [8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, 2012.

- [9] Hela Belhadj Amor, Carolynn Bernier, and Zdenek Prikryl. A RISC-V ISA extension for ultra-low power IoT wireless signal processing. *IEEE Transactions on Computers*, 2021:1–1, March 2021.
- [10] A. Bink. *OBI 1, Version 1.6.0*. Silicon Labs Inc., 2023.
- [11] 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, aug 2011.
- [12] Mark Bohr. A 30 year retrospective on dennard’s mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.
- [13] Alex Carsello, James Thomas, Ankita Nayak, Po-Han Chen, Mark Horowitz, Priyanka Raina, and Christopher Torng. Enabling reusable physical design flows with modular flow generators. *arXiv preprint arXiv:2111.14535*, 2021.
- [14] Christopher Celio, David A. Patterson, and Krste Asanović. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.
- [15] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.
- [16] 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 *Proceedings of the 2018 International Symposium on Physical Design, ISPD ’18*, page 48–55, New York, NY, USA, 2018. Association for Computing Machinery.
- [17] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189, 2017.
- [18] Enfang Cui, Tianzheng Li, and Qian Wei. Risc-v instruction set architecture extensions: A survey. *IEEE Access*, 11:24696–24711, 2023.
- [19] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ramp: Resource-aware mapping for cgras. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.
- [20] Shail Dave and Aviral Shrivastava. Ccf: A cgra compilation framework, 2018.
- [21] Shail Dave and Aviral Shrivastava. *CCF 20.04 User Manual*. Compiler Microarchitecture Lab, Arizona State University, Tempe, AZ, September 2021.
- [22] Gonçalo de Albuquerque Ferreira. Designing an instruction set based coarse grain accelerator. Dissertation, Faculdade de Engenharia da Universidade do Porto (FEUP), March 2024.

- [23] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, 2011.
- [24] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf. Introduction to co-simulation and emulation. In Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors, *Readings in Hardware/Software Co-Design*, Systems on Silicon, pages 523–525. Morgan Kaufmann, San Francisco, 2002.
- [25] G. De Michell and R.K. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.
- [26] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, and Christophe Jego. Qbox: an industrial solution for virtual platform simulation using qemu and systemc tlm-2.0. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016. fhal-01292317.
- [27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [28] 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*, 64(9):2448–2461, 2017.
- [29] R. Ernst. Codesign of embedded systems: status and trends. *IEEE Design & Test of Computers*, 15(2):45–54, 1998.
- [30] Xitian Fan, Huimin Li, Wei Cao, and Lingli Wang. Dt-cgra: Dual-track coarse-grained reconfigurable architecture for stream applications. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, page 1–9, Lausanne, Switzerland, August 2016. IEEE.
- [31] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.
- [32] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1027–1040. IEEE, 2021.
- [33] Brain floating point format (bfloat16). <https://cloud.google.com/tpu/docs/bfloat16>. Accessed: 2024-06-28.
- [34] Dennis Agyemanh Nana Gookyi, Eunchong Lee, Kyungho Kim, Sung-Joon Jang, and Sang-Seol Lee. Deep learning accelerators’ configuration space exploration effect on performance and resource utilization: A gemmini case study. *Sensors*, 23(5), 2023.
- [35] Yijiang Guo and Guojie Luo. Pillars: An integrated cgra design framework. In *Third Workshop on Open-Source EDA Technology (WOSET)*, 2020.
- [36] Catalina Gómez, Pablo Arbeláez, Miguel Navarrete, Catalina Alvarado-Rojas, Michel Le Van Quyen, and Mario Valderrama. Automatic seizure detection based on imaged-eeeg

- signals through fully convolutional networks. *Scientific Reports*, 10(1):21833, December 2020.
- [37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
  - [38] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. Risc-v based virtual prototype: An extensible and configurable platform for the system-level. *Journal of Systems Architecture*, 109:101756, 2020.
  - [39] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.
  - [40] Stavros Kalapothas, Manolis Galetakis, Georgios Flamis, Fotis Plessas, and Paris Kitsos. A survey on risc-v-based machine learning ecosystem. *Information*, 14(2), 2023.
  - [41] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, 2018.
  - [42] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.
  - [43] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 1–6, Austin TX USA, June 2017. ACM.
  - [44] Mahmoud Khairy, Amr G. Wassal, and Mohamed Zahran. A survey of architectural approaches for improving gpgpu performance, programmability and heterogeneity. *Journal of Parallel and Distributed Computing*, 127:65–88, 2019.
  - [45] Taeyoung Kong, Kalhan Koul, Priyanka Raina, Mark Horowitz, and Christopher Torng. Hardware abstractions and hardware mechanisms to support multi-task execution on coarse-grained reconfigurable arrays, 2023.
  - [46] Guilherme Korol, Michael Guilherme Jordan, Marcelo Brandalero, Michael Hübner, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. Mcea: A resource-aware multicore cgra architecture for the edge. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 33–39, 2020.
  - [47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.
  - [48] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

- [49] Xiaoyi Ling, Takahiro Notsu, and Jason Anderson. An open-source framework for the generation of risc-v processor + cgra accelerator systems. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 35–42, 2021.
- [50] 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), oct 2019.
- [51] 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 Computing Surveys*, 52(6):1–39, November 2020.
- [52] Shiya Liu, Dong Sam Ha, Fangyang Shen, and Yang Yi. Efficient neural networks for edge devices. *Computers & Electrical Engineering*, 92:107121, 2021.
- [53] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292, 2014.
- [54] Tao Lu. A survey on risc-v security: Hardware and architecture, 2021.
- [55] Simone Machetti, Pasquale Schiavone, Thomas Müller, Alexandre Levisse, Miguel Peón-Quirós, and David Atienza. *HEEPocrates: An Ultra-Low-Power RISC-V Microcontroller for Edge-Computing Healthcare Applications*. Europractice, March 2024.
- [56] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators, 2024.
- [57] Laurent Maillet-Contoz and Frank Ghenassia. *Transaction Level Modeling*, pages 23–55. Springer US, Boston, MA, 2005.
- [58] Kevin J. M. Martin. Twenty years of automated methods for mapping applications on cgra. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 679–686, 2022.
- [59] B. Mei, M. Berekovic, and J-Y. Mignolet. *ADRES & DRESC: Architecture and Compiler for Coarse-GrainReconfigurable Processors*, pages 255–297. Springer Netherlands, Dordrecht, 2007.
- [60] 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 Peter Y. K. Cheung and George A. Constantinides, editors, *Field Programmable Logic and Application*, pages 61–70, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [61] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and systemc: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 62–69, 2017.
- [62] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen

- Mellempudi, Stuart Oberman, Mohammad Shoeybi, Michael Siu, and Hao Wu. Fp8 formats for deep learning, 2022.
- [63] Tom R. Miller and Kondapuram S. Sampathkumaran. Design and application of finite impulse response digital filters. *European Journal of Nuclear Medicine*, 7(1):22–27, 1982.
- [64] Anh Quynh Nguyen and Hoang Vu Dang. Unicorn: Next generation cpu emulator framework. Presented at BlackHat USA, August 2015. <https://www.unicorn-engine.org>.
- [65] OpenHW Group. CORE-V eXtension Interface. <https://docs.openhwgroup.org/projects/openhw-group-core-v-xif>, 2021. [Accessed: Jun. 29, 2024].
- [66] 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, March 2015.
- [67] Biagio Peccerillo, Elham Cheshmikhani, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. Ixiam: Isa extension for integrated accelerator management. *IEEE Access*, 11:33768–33791, 2023.
- [68] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.
- [69] Omar Ragheb, Stephen Wicklund, Matthew Walker, Rami Beidas, Adham Ragab, Tianyi Yu, and Jason Anderson. Cgra-me 2.0: A research framework for next-generation cgra architectures and cad. In *Proceedings of the 3rd International Workshop on Coarse-Grained Reconfigurable Architectures for High-Performance Computing (CGRA4HPC)*, 2024.
- [70] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, and Luca Benini. Pulp: A parallel ultra low power platform for next generation iot applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–39, 2015.
- [71] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.
- [72] Manoj Sharma, Ekansh Bhatnagar, Kartik Puri, Amitav Mitra, and Jatin Agarwal. A survey of risc-v cpu for iot applications. *SSRN Electronic Journal*, 2022.
- [73] 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.
- [74] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh. Yaml: a tool for hardware design visualization and capture. In *Proceedings 13th International Symposium on System Synthesis*, pages 9–14, 2000.
- [75] Wilson Snyder. *Verilator Manual*. Veripool, release 5.024 edition, April 2024. Chapter 6.5, pp. 26-29.
- [76] Stuart Swan. An introduction to system level modeling in systemc 2.0. *Cadence Design Systems, Inc., draft report*, 2001.

- [77] Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388, 2020.
- [78] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.
- [79] Cristian Tirelli, Lorenzo Ferretti, and Laura Pozzi. Sat-mapit: A sat-based modulo scheduling mapper for coarse grain reconfigurable architectures. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.
- [80] Shihang Wang, Jianghan Zhu, Qi Wang, Can He, and Terry Tao Ye. Customized instruction on risc-v for winograd-based convolution acceleration. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 65–68, 2021.
- [81] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214*. RISC-V International, December 2019. Version 20191214.
- [82] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. Morpher: An open-source integrated compilation and simulation framework for cgra. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*, 2022.
- [83] Pengcheng Xu and Yun Liang. Automatic code generation for rocket chip rocc accelerators. *CARRV*, 2020.
- [84] Fulai Zhu, Peiyu Xu, and Jiahao Zong. Moore’s law: The potential, limits, and breakthroughs. *Applied and Computational Engineering*, 10:307–315, 09 2023.
- [85] Tomasz P. Zieliński. *FIR Digital Filters*, pages 227–256. Springer International Publishing, Cham, 2021.



## Appendix A

### LSU element class

The following listing contains the complete file that includes *LSElement* class. This class mirrors what the **PE** performs, with a control bit specifying which action to undertake. The store function uses the first input operand, the address, to store the value provided in the second input operand. The load operation returns the value in the first address operand, ignoring the other. The *MemoryComm* provides an interface assigned through dependency injection, which facilitates the adoption of the shared memory in the **SoC**, by inheriting from the *MemoryComm* with new method implementations.

```
1
2 public class LSElement extends BinaryProcessingElement implements
   ↪ PEControl<LSControlSetting> {
3
4     private void debug(String str) {
5         SpecsLogs.debug(LSElement.class.getSimpleName() + ": " + str);
6     }
7
8     /*
9     * At the next "_execute" step assumes:
10    * - Operand 1 is base addr
11    * - Operand 2 is offset
12    */
13    private LSControlSetting ctrl;
14
15    private MemoryComm memoryComm;
16
17    public LSElement(int latency) {
18        super(1, 1);
19        this.ctrl = LSControlSetting.NULL;
20    }
21
22    public LSElement(int latency, int memorySize, boolean RFEnabled) {
23        super(latency, memorySize, RFEnabled);
24        this.ctrl = LSControlSetting.NULL;
25    }
26
27    public LSElement(int latency, int memorySize) {
28        super(latency, memorySize);
29        this.ctrl = LSControlSetting.NULL;
30    }
31
32    @Override
33    public LSControlSetting getOperation() {
34        return this.ctrl;
35    }
36
37
38    private PEData load() {
```

```

39         PData addr = this.getOperand(0);
40
41         PData value = this.memoryComm.accessMemory(addr.getValue().intValue());
42
43         return value;
44     }
45
46     private PData store() {
47         PData addr = this.getOperand(0); // addr
48         PData data = this.getOperand(1); // data to store
49
50         this.memoryComm.writeMemory(addr.getValue().intValue(), data);
51
52         return data;
53     }
54
55     @Override
56     protected PData _execute() {
57         if (ctrl == LSControlSetting.LOAD)
58             return this.load();
59         else if (ctrl == LSControlSetting.STORE)
60             return this.store();
61         else
62             throw new RuntimeException("LSElement: invalid control setting");
63         return new PDataNull();
64     }
65
66     @Override
67     public void reset() {
68         this.ctrl = LSControlSetting.NULL;
69         super.reset();
70     }
71
72     @Override
73     public boolean setOperation(LSControlSetting ctrl) {
74         this.ctrl = ctrl;
75         this.debug("set control on " + this.getAt()
76             + " for instruction " + this.getOperation().getName());
77         return true;
78     }
79
80     @Override
81     protected LSElement getThis() {
82         return this;
83     }
84
85     @Override
86     public PType getType() {
87         return PType.LSElement;
88     }
89
90     public void setMemoryComm(MemoryComm memoryComm) {
91         this.memoryComm = memoryComm;
92     }
93
94 }
95
96

```

Listing 8: The LSU Java class.

## Appendix B

### *CGRAController* methods

Listing 9 lists portions of the *CGRAController* that are related to the execution of a **CGRA** kernel. One of the possible execution modes terminates the process when a given **PE** outputs a boolean value from a logical expression. The *executeWithCondition* loops the application of the next appropriate context and steps the **CGRA** (propagates and executes the mesh) until it is signaled by the **CGRA** (which internally has access to the *setFinishedExecution* call-back). The controller uses the *applyNextContext* method for each execution, depending on the mode it requires to place the next context configuration.

```
1
2 public class CGRAController{
3     .....
4     /**
5      * Executes the CGRA until the execution is finished (halt by condition).
6      */
7     public void executeWithCondition() {
8         int currCycle = 0;
9         this.cgra.setParent(this);
10
11         do {
12             applyNextContext(currCycle);
13             stepCGRA();
14             currCycle++;
15         } while (!state.equals(CGRAState.FINISHED_EXECUTION));
16     }
17     .....
18     /**
19      * Sets the CGRA state to finished execution.
20      * @param controller
21      */
22     public static void setFinishedExecution(CGRAController controller) {
23         controller.setState(CGRAState.FINISHED_EXECUTION);
24     }
25     .....
26
27     /**
28      * Applies the next context in the CGRA, according to the context utilization
29      * ↪ mode.
30      *
31      * @param iteration
32      *     - current iteration
33      */
34     public void applyNextContext(Integer iteration) {
35         if (contextUtilization == ContextUtilization.FIXED) { // Unaltered context
36             if (cgra.getContextMemory().size() == 0)
37                 throw new CriticalException("No context available for execution.");
38         } else if (contextUtilization == ContextUtilization.ALTERNATED) { //
39             ↪ Alternate context
```

```

39         if (!cgra.applyContext(iteration % cgra.getContextMemory().size()))
40             throw new CriticalException("Could not apply context for iteration
41                                     ↪ " + iteration);
42     } else if (contextUtilization == ContextUtilization.LINEAR) { // Linear
43         ↪ context (depends on current iteration)
44         if (!cgra.applyContext(iteration))
45             throw new CriticalException("Could not apply context for iteration
46                                     ↪ " + iteration);
47     } else {
48         throw new CriticalException("Invalid context utilization mode");
49     }
50 }
51 .....
52 }

```

Listing 9: Portion of the *CGRAController* that controls the execution with a boolean halting PE.

## Appendix C

# Three main components of the X-HEEP simulation

This appendix collects three components required to create within the **X-HEEP**'s environment to pair with the CGRASim instance, illustrated in Figure 4.9. Firstly, Listing 10 represents the **CGRA** interface in *SystemVerilog* and the *always\_ff* flip-flop that ensues upon a read or write on a specific memory address. Secondly, Listing 11 lists the three **DPI** functions that are embedded in the hardware module and allow the interface to communicate to CGRASim with UNIX address domain sockets. Finally, Listing 12, which is referenced throughout the document, is the boilerplate provided to the user to include their specific kernel instructions.

### C.1 CGRAItf SystemVerilog module

```
1
2 module cgraitf #(
3     parameter type reg_req_t = logic,
4     parameter type reg_rsp_t = logic,
5     parameter type obi_req_t = logic,
6     parameter type obi_rsp_t = logic
7 ) (
8     input logic clk_i,
9     input logic rst_ni,
10
11     input reg_req_t reg_req_i,
12     output reg_rsp_t reg_rsp_o,
13
14     output obi_req_t masters_req_o,
15     input obi_rsp_t masters_rsp_i,
16
17     input obi_req_t slave_req_i,
18     output obi_rsp_t slave_rsp_o
19 );
20
21 assign reg_rsp_o.rdata = '0;
22 assign reg_rsp_o.error = 1'b0;
23 assign reg_rsp_o.ready = 1'b1;
24 assign reg_rsp_o.rdata = 32'b0;
25
26 assign masters_req_o.req = '0;
27 assign masters_req_o.addr = '0;
28 assign masters_req_o.we = '0;
29 assign masters_req_o.be = '0;
30 assign masters_req_o.wdata = '0;
```

```

31  assign slave_resp_o.gnt      = '0;
32  ;
33  assign slave_resp_o.rdata = '0;
34  ;
35  assign slave_resp_o.rvalid = '0;
36  ;
37
38  import "DPI-C" functionchandle cgraitfdpi_create();
39  import "DPI-C" function int cgraitfdpi_read(chandle ctx);
40  import "DPI-C" function void cgraitfdpi_close(chandle ctx);
41  import "DPI-C" function void cgraitfdpi_write(
42      chandle ctx,
43      int data
44  );
45
46
47  chandle ctx;
48
49
50  initial begin
51      ctx = cgraitfdpi_create();
52  end
53
54  final begin
55      cgraitfdpi_close(ctx);
56      ctx = null;
57  end
58
59
60
61  always_ff @(posedge clk_i or negedge rst_ni) begin
62      if (rst_ni) begin
63          if (reg_req_i.valid && reg_req_i.write) begin
64              cgraitfdpi_write(ctx, reg_req_i.wdata);
65          end
66
67          if (reg_req_i.valid && ~reg_req_i.write) begin
68              reg_rsp_o.rdata = cgraitfdpi_read(ctx);
69          end
70      end
71  end
72
73  endmodule
74

```

Listing 10: SystemVerilog module for the **CGRA** interface in the **X-HEEP** platform.

## C.2 DPI function definition for the Co-simulation

```

1
2  #include "cgraitfdpi.h"
3
4  #ifdef __linux__
5  #include <pty.h>
6  #elif __APPLE__
7  #include <util.h>
8  #endif
9
10 #include <stdio.h>
11 #include <sys/stat.h>
12 #include <assert.h>
13 #include <stdlib.h>
14 #include <sys/un.h>
15 #include <sys/socket.h>
16 #include <errno.h>
17 #include <unistd.h>
18 #include <sys/types.h>          /* See NOTES */
19 #include <sys/time.h>
20
21

```

```

22
23 void *cgraitfdpi_create() {
24     // will block until the sock is opened for reading
25     // Assumes that the sock is re-created every time
26     struct cgraitfdpi_ctx *ctx = (struct cgraitfdpi_ctx *)malloc(sizeof(struct
    ↪ cgraitfdpi_ctx));
27
28
29     int sfd, cfd;
30     struct sockaddr_un *addr = (struct sockaddr_un *)malloc(sizeof(struct
    ↪ sockaddr_un));
31
32     // Create a socket with Unix domain
33     printf("Creating socket\n");
34     sfd = socket(AF_UNIX, SOCK_STREAM, 0);
35
36     if (sfd == -1) {
37         perror("socket");
38         exit(1);
39     }
40
41     if(remove(SV_SOCKET_PATH) == -1 && errno != ENOENT) {
42         perror("remove");
43         exit(1);
44     }
45
46
47     // set timeout in socket
48     struct timeval tv;
49     tv.tv_sec = 10;
50     tv.tv_usec = 0;
51     setsockopt(sfd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof tv);
52
53     printf("Creating address\n");
54     memset(addr, 0, sizeof(struct sockaddr_un));
55     addr->sun_family = AF_UNIX;
56     strncpy(addr->sun_path, SV_SOCKET_PATH, sizeof(addr->sun_path) - 1);
57
58     printf("Binding to %s\n", SV_SOCKET_PATH);
59     if(bind(sfd, (struct sockaddr *)addr, sizeof(struct sockaddr_un)) == -1) {
60         perror("bind");
61         exit(1);
62     }
63
64     printf("Listening on %s\n", SV_SOCKET_PATH);
65     if(listen(sfd, 1) == -1) {
66         perror("listen");
67         exit(1);
68     }
69
70     printf("Accepting connection\n");
71
72     fd_set rfd;
73     FD_ZERO(&rfd);
74     FD_SET(sfd, &rfd);
75     int status = select(sfd + 1, &rfd, (fd_set *)0, (fd_set *)0, &tv);
76
77     if (status == -1) {
78         perror("select");
79         exit(1);
80     } else if (status == 0) {
81         printf("Timeout\n");
82         exit(1);
83     }
84
85
86     cfd = accept(sfd, NULL, NULL);
87     if(cfd == -1) {
88         perror("accept");
89         exit(1);
90     }
91
92     ctx->sock = cfd;
93     printf("Connected\n");

```

```

94     return ctx;
95 }
96
97
98
99 int cgraitfdpi_read(void *ctx_void){
100     struct cgraitfdpi_ctx *ctx = (struct cgraitfdpi_ctx *)ctx_void;
101     int data;
102     int ret = recv(ctx->sock, &data, sizeof(data), MSG_WAITALL);
103
104     if (ret == -1){
105         perror("recv");
106         cgraitfdpi_close(ctx_void);
107         exit(1);
108     }
109     return data;
110 }
111
112
113 void cgraitfdpi_close(void *ctx_void){
114     struct cgraitfdpi_ctx *ctx = (struct cgraitfdpi_ctx *)ctx_void;
115     close(ctx->sock);
116     free(ctx);
117 }
118
119 void cgraitfdpi_write(void *ctx_void, int data){
120     printf("Writing %d\n", data);
121     struct cgraitfdpi_ctx *ctx = (struct cgraitfdpi_ctx *)ctx_void;
122
123     unsigned char buf[4];
124     buf[0] = (data >> 24) & 0xFF;
125     buf[1] = (data >> 16) & 0xFF;
126     buf[2] = (data >> 8) & 0xFF;
127     buf[3] = data & 0xFF;
128
129     int ret = send(ctx->sock, &buf, sizeof(buf), 0);
130     if (ret == -1){
131         perror("send");
132         cgraitfdpi_close(ctx_void);
133         exit(1);
134     }
135 }
136

```

Listing 11: DPI functions definition for read and write operations in the main CPU executable.

### C.3 Executable file (main.c)

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <stddef.h>
5 #include <stdint.h>
6 #include "core_v_mini_mcu.h"
7
8 #define CGRA_PERIPH_START_ADDRESS (EXT_PERIPHERAL_START_ADDRESS + 0x06000)
9 #define CGRA_PERIPH_SIZE 0x000100
10 #define CGRA_PERIPH_END_ADDRESS (CGRA_PERIPH_START_ADDRESS + CGRA_PERIPH_SIZE)
11
12
13
14 const uint32_t instructions[] = {
15     .... // hexadecimal instructions
16 };
17
18
19 #define N_LIVEOUTS 1
20 const uint32_t liveout_instr[] = {
21     .... // hexadecimal instructions
22 };

```

```

23
24
25
26 void kernel(uint32_t *data, uint32_t *extractint, uint8_t n_liveouts) {
27     volatile static uint32_t *cgra_cmem_ptr = (CGRA_PERIPH_START_ADDRESS);
28
29     // Config and start execution instructions
30     for (int i = 0; i < sizeof(instructions)/sizeof(uint32_t); i++) {
31         cgra_cmem_ptr[i] = data[i];
32     }
33
34     int a;
35     // Retrieve liveout instructions
36     for (int i = 0; i < n_liveouts; i++) {
37         cgra_cmem_ptr[i] = extractint[i]; // send the instruction to the cgra
38         a = cgra_cmem_ptr[i]; // read the result from the cgra (blocking wait)
39     }
40 }
41
42
43
44 int main(int argc, char *argv[])
45 {
46     kernel(instructions, liveout_instr, N_LIVEOUTS);
47     return EXIT_SUCCESS;
48 }

```

Listing 12: Content and operations included in the main.c file that is provided to the CPU model to execute the co-simulation.



## Appendix D

# Micro-code generation with python

This appendix presents the main parser script, in Listing 13, used to decode a text file and generate micro-code instructions based on our proposed specification. The script translates mnemonics (with arguments) into corresponding hexadecimal codes. There is also an example of such translation included in Listing 14, with every hexadecimal code as the same-line comment.

### D.1 parser.py script

```
1 import re
2 from operations import startSetup, \
3     startContext, \
4     routeTwoPEs, \
5     linkRFTToPE, \
6     linkPEToRF, \
7     setControlBits, \
8     storeContext, \
9     storeLiveInFirstPacket, \
10    storeLiveInSecondPacket, \
11    startExecution, \
12    retrieveLiveOut, \
13    stopExecution, \
14    selectPEForHalt, \
15    stepExecution
16
17 """
18 Mapping of the operations to the functions in operations.py
19 """
20 operation_mapping = {
21     "startSetup": startSetup,
22     "startContext": startContext,
23     "routeTwoPEs": routeTwoPEs,
24     "linkRFTToPE": linkRFTToPE,
25     "linkPEToRF": linkPEToRF,
26     "setControlBits": setControlBits,
27     "storeContext": storeContext,
28     "storeLiveInFirstPacket": storeLiveInFirstPacket,
29     "storeLiveInSecondPacket": storeLiveInSecondPacket,
30     "startExecution": startExecution,
31     "retrieveLiveOut": retrieveLiveOut,
32     "stopExecution": stopExecution,
33     "selectALUForHalt": selectPEForHalt,
34     "stepExecution": stepExecution
35 }
36
37 '''
38 Parsing related functions.
39 @Author: Antonio
40 '''
```

```

41  '''
42
43
44  def empty_line(line: str) -> bool:
45      """Check if a line is empty
46
47      Args:
48          line (str): line to check
49
50      Returns:
51          bool: True if the line is empty, False otherwise
52      """
53      return line is None or line == "\n" or line == "\r\n"
54          \or line == "" or line == "\r"
55
56
57  def parse_line(line: str) -> tuple:
58      """Parse a line of the input file
59
60
61      Args:
62          line (str): The line to parse
63
64      Returns:
65          Tuple[str, List[int]]: The operation name and arguments
66      """
67
68
69      # Remove comments and strip whitespace
70      line = re.sub(r'#.+', '', line).strip()
71      line = line.replace(" ", "")
72
73      if empty_line(line):
74          return None, None
75
76
77      # Extract the operation name and arguments
78      match = re.match(r'(\w+)\s*(((^)\s*))', line)
79      if not match:
80          return None, None
81
82      operation_name, args_str = match.groups()
83      args = [int(arg.strip())
84              for arg in args_str.split(',')] if args_str else []
85
86      return operation_name, args
87
88
89  def parse_file(filename : str) -> str:
90      """Parse the input file and generate the output
91
92
93      Args:
94          filename (str): The input file
95
96      Returns:
97          str: The output string
98      """
99
100      output = ""
101      line_no = 0
102
103      with open(filename, 'r') as file:
104          for line in file:
105              try:
106                  operation_name, args = parse_line(line)
107                  except Exception as e:
108                      print(f"Error in line {line_no}" + e)
109                      exit(1)
110
111                  line_no += 1
112
113                  if operation_name is None:
114                      continue

```

```

115
116         if operation_name not in operation_mapping:
117             print(f"Invalid operation: {operation_name}")
118             exit(1)
119         print(operation_name)
120
121         try:
122             output += operation_mapping[operation_name](*args) + "\n"
123         except Exception as e:
124             print(f"Error in line {line}" + e)
125             exit(1)
126
127     # Remove the last newline character
128     return output[:-1] if output[-1] == "\n" else output
129

```

Listing 13: Python Parser that generates instructions from mnemonics.

## D.2 Example of input and output for the parser script

```

1
2  startSetup(1), #0x81
3
4  # CYCLE 0
5  # CMP retrieves loop term and verifies end
6  startContext(0), #0x9
7  selectPEForHalt(3), #0x1e3
8  setControlBits(3, 13), #0x36ab
9  linkRFTToPE(0, 3), #0x30020
10 linkRFTToPE(1, 3), #0x300a0
11 storeContext(0), #0x31
12
13 # CYCLE 1
14 # ADD above 1st LSU receives iterator and adds with pointer
15 startContext(1), #0x89
16 setControlBits(1, 1), #0x10ab
17 linkRFTToPE(0, 1), #0x10020
18 linkRFTToPE(2, 1), #0x10120
19 storeContext(1), #0xb1
20
21 # CYCLE 2
22 # First LSU loads value and ADD above second LSU adds pointer
23 startContext(2), #0x109
24 routeTwoPEs(1, 5), #0x50090
25 setControlBits(5, 1), #0x50ab
26 setControlBits(2, 1), #0x20ab
27 linkRFTToPE(0, 2), #0x20020
28 linkRFTToPE(3, 2), #0x201a0
29 storeContext(2), #0x131
30
31 # CYCLE 3
32 # LSU receives value and INC receives iterator
33 startContext(3), #0x189
34 routeTwoPEs(2, 6), #0x60110
35 setControlBits(6, 1), #0x60ab
36 linkRFTToPE(0, 10), #0xa0020
37 storeContext(3), #0x1b1
38
39 # CYCLE 4
40 # LSUs propagate the value to MUL and INC stores iterator
41 startContext(4), #0x209
42 routeTwoPEs(5, 9), #0x90290
43 routeTwoPEs(6, 9), #0x90310
44 setControlBits(9, 3), #0x91ab
45 linkPEToRF(10, 0), #0x518
46 storeContext(4), #0x231
47
48 # CYCLE 5
49 # Adder collects accumulator and result from multiplication
50 startContext(5), #0x289

```

```

51 routeTwoPEs(9, 13), #0xd0490
52 linkRFTToPE(4, 13), #0xd0220
53 setControlBits(13, 1), #0xd0ab
54 storeContext(5), 30x2b1
55
56 # CYCLE 6
57 # Adder stores in accumulator
58 startContext(6), #0x309
59 linkPEToRF(13, 4), #0x40698
60 storeContext(6), #0x331
61
62 # Store in addr 0 the value 0 - iterator
63 storeLiveInFirstPacket(0, 0), #0x54
64 storeLiveInSecondPacket(0, 0), #0x5
65
66 # Store in addr 1 the value 4 - loop end iter value
67 storeLiveInFirstPacket(1, 4), #0x200d4
68 storeLiveInSecondPacket(1, 4), #0x5
69
70 # Store in addr 2 the value 0 - first pointer begin addr
71 storeLiveInFirstPacket(2, 0), #0x154
72 storeLiveInSecondPacket(2, 0), #0x5
73
74 # Store in addr 3 the value 4 - second pointer begin addr
75 storeLiveInFirstPacket(3, 4), #0x201d4
76 storeLiveInSecondPacket(3, 4), #0x5
77
78 # Store in addr 4 the value 0 - accum starting value
79 storeLiveInFirstPacket(4, 0), #0x254
80 storeLiveInSecondPacket(4, 0), #0x5
81
82 # Start execution
83 startExecution(1, 0), #0xb8
84
85 # Retrieve live-out 4 (accumulator for the dot-product)
86 retrieveLiveOut(4) #0x259
87

```

Listing 14: Input file with the corresponding hexadecimal conversions.

## Appendix E

# *UnaryIncrement* YAML architecture and class files

The following Listing 15 is an example of an architectural description in the **YAML** format for a  $4 \times 4$  mesh that executes the dot-product kernel. It highlights that CGRASim has the ability to create a new type of **PE**, for instance, the *UnaryIncrement*, that increments an input value. This class, shown in Listing 16, is necessary to execute the dot product and **FIR** filter kernels, as it increments the iterator value.

```
1 CGRA:
2   rows: 4
3   columns: 4
4   connectionType: NN
5   registerFileSize: 100
6
7   processingElements:
8     - { type: "ALU", x: 0, y: 0, latency: 1, memorySize: 1, RFenabled: True}
9     - { type: "ALU", x: 0, y: 1, latency: 1, memorySize: 1, RFenabled: True}
10    - { type: "ALU", x: 0, y: 2, latency: 1, memorySize: 1, RFenabled: True}
11    - { type: "ALU", x: 0, y: 3, latency: 1, memorySize: 1, RFenabled: True}
12
13    - { type: "ALU", x: 1, y: 0, latency: 1, memorySize: 1, RFenabled: True}
14    - { type: "LSU", x: 1, y: 1, latency: 1, memorySize: 1, RFenabled: True}
15    - { type: "LSU", x: 1, y: 2, latency: 1, memorySize: 1, RFenabled: True}
16    - { type: "ALU", x: 1, y: 3, latency: 1, memorySize: 1, RFenabled: True}
17
18    - { type: "ALU", x: 2, y: 0, latency: 1, memorySize: 1, RFenabled: True}
19    - { type: "ALU", x: 2, y: 1, latency: 1, memorySize: 1, RFenabled: True}
20    - { type: "UNARYINC", x: 2, y: 2, latency: 1, memorySize: 1, RFenabled: True}
21    - { type: "ALU", x: 2, y: 3, latency: 1, memorySize: 1, RFenabled: True}
22
23    - { type: "ALU", x: 3, y: 0, latency: 1, memorySize: 1, RFenabled: True}
24    - { type: "ALU", x: 3, y: 1, latency: 1, memorySize: 1, RFenabled: True}
25    - { type: "ALU", x: 3, y: 2, latency: 1, memorySize: 1, RFenabled: True}
26    - { type: "ALU", x: 3, y: 3, latency: 1, memorySize: 1, RFenabled: True}
27
28
```

Listing 15: **YAML** architecture description that includes an *UnaryIncrement* unit.

```
1
2 package pt.up.specs.cgra.structure.pes.unary;
3
4 import pt.up.specs.cgra.datatypes.PEData;
5 import pt.up.specs.cgra.structure.pes.PEType;
```

```

6  import pt.up.specs.cgra.structure.pes.ProcessingElement;
7  import pt.up.specs.cgra.structure.pes.binary.AdderElement;
8
9  public class UnaryIncrement extends UnaryProcessingElement {
10
11
12      public UnaryIncrement(int latency, int memorySize, boolean rfEnabled) {
13          super(latency, memorySize, rfEnabled);
14      }
15
16      public UnaryIncrement(int latency, int memorySize) {
17          super(latency, memorySize);
18      }
19
20      public UnaryIncrement(int latency) {
21          this(latency, 0);
22      }
23
24      public UnaryIncrement() {
25          this(1, 0);
26      }
27
28      @Override
29      protected PEData _execute() {
30          return this.getOperand(0).increment();
31      }
32
33      @Override
34      public ProcessingElement copy() {
35          return new UnaryIncrement(this.getLatency(), this.getMemorySize());
36      }
37
38      @Override
39      protected UnaryIncrement getThis() {
40          return this;
41      }
42
43      @Override
44      public PType getType() {
45          return PType.UnaryBitwiseOr;
46      }
47  }
48
49

```

Listing 16: Unary Increment unit code in the CGRASim platform.

## Appendix F

### *FMT1Instruction* class

The *FMT1Instruction* in Listing 17 is one of the different five classes that use the hexadecimal codes as inputs and store useful information that is used when executing the instruction command, namely: the stage of execution (context configuration, runtime, query), the opcode, the *InstructionCommand* command (a functional call-back used by the controller), the operands array and the instruction. The class requires knowledge about which positions store specific values, extracted with the parser library.

```
1
2 import java.util.ArrayList;
3 import java.util.List;
4
5 import pt.up.specs.platform.decoder.Decoder;
6 import pt.up.specs.platform.decoder.instruction.DecodedInstruction;
7 import pt.up.specs.platform.decoder.instruction.InstructionCommand;
8
9 public class FMT1Instruction extends DecodedInstruction implements
10     ↳ OnePacketDecoding {
11     public FMT1Instruction() {}
12
13     @Override
14     public DecodedInstruction decode(int[] fields, long instr) {
15         this.opcode = fields[Decoder.OPCODE_POS];
16         this.instr = instr;
17
18         this.operands = new ArrayList<Integer>();
19         this.operands.add(fields[Decoder.OPERANDA_POS]);
20         this.operands.add(fields[Decoder.OPERANDB_POS]);
21
22         this.command = InstructionCommand.fromInt(opcode);
23         this.stage = command.getStage();
24
25         return this;
26     }
27
28 }
```

Listing 17: *FMT1Instruction* class, which interprets a FMT1, inheriting from *instructionDecodedInstruction*.



## Appendix G

# Functional verification

This appendix lists the two main test benches for the first validation scenario, that were used to run experiments and register their runtimes. Analyzing their methods, we can observe a setup operation (that executes before the test) and creates the main simulator structure (that aggregates the **CGRA**, simulated memory, Decoder infrastructure, and file or socket **I/O**). Then after correctly laying down the correct input in the main memory for each kernel, it executes the *IOLoop*, which is timed. Finally, the output is validated according to a pure software implementation of the algorithm.

### G.1 Dot product test bench

```
1
2 package pt.up.specs.cgra;
3
4 import org.junit.Before;
5 import org.junit.Test;
6
7 import pt.up.specs.cgra.datatypes.PEInteger;
8 import pt.up.specs.cgra.structure.SpecsCGRA;
9 import pt.up.specs.cgra.utils.FillMemory;
10 import pt.up.specs.platform.memory.Address;
11 import pt.up.specs.simulation.FileSimulation;
12
13 public class TestNaiveDotProduct {
14     private FileSimulation simulator;
15
16     static int NUMINPUTS = 32;
17
18     @Before
19     public void setup2x2Sim() {
20         simulator = new FileSimulation("4x4_dotproduct.yml", 1000);
21
22         FillMemory.insertArraysForDotProduct(simulator, NUMINPUTS);
23     }
24
25     @Test
26     public void testFileDotProductSimulation() {
27         long startTime = System.nanoTime();
28         simulator.IOLoop("resources/naivedotproduct.txt", 0);
29         long endTime = System.nanoTime();
30         long duration = (endTime - startTime);
31         System.out.println("Duration: " + duration);
32
33         SpecsCGRA cgra = simulator.getPlatform().getCGRA();
34     }
```

```

35 Integer accum = cgra.readRegisterFile(new
    ↪ PEInteger(4)).getValue().intValue();
36 // lambda dot product
37
38 Integer expected = 0;
39 for (int i = 1; i <= NUMINPUTS; i++) {
40     expected += i * (i + NUMINPUTS);
41 }
42
43 assert(accum.equals(expected));
44 }
45 }
46
47

```

Listing 18: Dot-Product CGRASim test-bench, for the first verification scenario.

## G.2 FIR filter test bench

```

1 public class TestFIRLoop {
2     private FileSimulation simulator;
3
4     static int NUMINPUTS = 16;
5
6
7     private void verifyResults(PlatformMemory memory) {
8         int[] input = new int[NUMINPUTS];
9         Arrays.setAll(input, i -> i + 1);
10
11         int[] output = new int[NUMINPUTS];
12
13         int[] samples = new int[3];
14         int[] coefficients = new int[3];
15         coefficients[0] = 1;
16         coefficients[1] = 2;
17         coefficients[2] = 3;
18
19         for (int i = 0; i < NUMINPUTS; i++) {
20             samples[2] = samples[1];
21             samples[1] = samples[0];
22             samples[0] = input[i];
23             output[i] = 0;
24             for (int j = 0; j < 3; j++) {
25                 output[i] += samples[j] * coefficients[j];
26             }
27         }
28
29
30         for (int i = 0; i < NUMINPUTS; i++) {
31             Address address = new Address(i + NUMINPUTS);
32             PEInteger value = (PEInteger) memory.accessMemory(address);
33             System.out.println("Address: " + address.getValue() + " Value: " +
    ↪ value.getValue());
34             assert (value.getValue() == output[i]);
35         }
36     }
37
38
39     @Before
40     public void setup2x2Sim() {
41         simulator = new FileSimulation("5x5_fir_redux.yml", 100000);
42         FillMemory.insertArraysForFir(simulator, NUMINPUTS);
43     }
44
45     @Test
46     public void testFileFIRSimulation() {
47         long startTime = System.nanoTime();
48         simulator.IOLoop("resources/naivefir.txt", 0);
49         long endTime = System.nanoTime();
50         long duration = (endTime - startTime);

```

```

51         System.out.println("Duration: " + duration);
52
53         PlatformMemory memory = simulator.getMemory();
54         verifyResults(memory);
55
56     }
57 }
58

```

Listing 19: FIR filter CGRASim test-bench, for the first verification scenario.

## G.3 Co-simulation test bench

```

1  package pt.up.specs.cgra;
2
3  import org.junit.Test;
4
5  import pt.up.specs.cgra.utils.FillMemory;
6  import pt.up.specs.simulation.CoSimulationWithProcess;
7
8  public class TestCoSimulationWithXHEEP {
9
10
11     static int NUMINPUTS = 4;
12
13     @Test
14     public void testCoSimulationWithXHEEP() {
15         long startTime = System.nanoTime();
16         CoSimulationWithProcess simulator = new
17             ↪ CoSimulationWithProcess("4x4_fir.yml", 10000);
18         FillMemory.insertArraysForFIR(simulator, NUMINPUTS);
19         simulator.IOLoop("/tmp/cgrasim", 1);
20         long endTime = System.nanoTime();
21         long duration = (endTime - startTime);
22         System.out.println("Duration: " + duration);
23     }
24 }

```

Listing 20: FIR filter CGRASim test-bench, for the second verification scenario



# Appendix H

## C implementation of the kernels

The following programs in Listing 21 and Listing 22 are the pure-software implementations of the two kernels used to verify the platform. They were used to compare the clock cycles (and subsequently, the time duration) to the co-simulation partially offloaded task, between the core and the CGRA, depicted in Figure 5.12.

### H.1 C version of the dot-product

```
1  include <stdio.h>
2  #include <stdlib.h>
3  #include <stddef.h>
4  #include <stdint.h>
5  #include "core_v_mini_mcu.h"
6
7  #define CGRA_PERIPH_START_ADDRESS (EXT_PERIPHERAL_START_ADDRESS + 0x06000)
8  #define CGRA_PERIPH_SIZE 0x000100
9  #define CGRA_PERIPH_END_ADDRESS (CGRA_PERIPH_START_ADDRESS + CGRA_PERIPH_SIZE)
10
11
12  char array1[64] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
13  ↪ 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
14  ↪ 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58,
15  ↪ 59, 60, 61, 62, 63, 64};
16  char array2[64] = {65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
17  ↪ 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
18  ↪ 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
19  ↪ 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128};
20
21
22  int kernel(char *array1, char *array2, int size)
23  {
24      int i;
25      int sum = 0;
26      for (i = 0; i < size; i++)
27      {
28          sum += array1[i] * array2[i];
29      }
30
31      return sum;
32  }
33
34
35  int main(int argc, char *argv[])
36  {
37      kernel(array1, array2, 64);
38      return EXIT_SUCCESS;
39  }
```

```

34 }
35

```

Listing 21: Dot-product kernel in C

## H.2 C version of the FIR filter

```

1
2 include <stdio.h>
3 #include <stdlib.h>
4 #include <stddef.h>
5 #include <stdint.h>
6 #include "core_v_mini_mcu.h"
7
8 #define CGRA_PERIPH_START_ADDRESS (EXT_PERIPHERAL_START_ADDRESS + 0x06000)
9 #define CGRA_PERIPH_SIZE 0x000100
10 #define CGRA_PERIPH_END_ADDRESS (CGRA_PERIPH_START_ADDRESS + CGRA_PERIPH_SIZE)
11
12
13 char array1[64] = {1, 2, 3, 4, 5, 6, 7, 8, 9,
14 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
15 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
16 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
17 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57,
18 58, 59, 60, 61, 62, 63, 64};
19 char arra2[64];
20
21
22 char coeffs[3] = {1, 2, 3};
23
24 void kernel(char *array1, char *array2, char *coeffs, int size)
25 {
26     char samples[3] = {0, 0, 0};
27
28     for (int i = 0; i < size; i++)
29     {
30         samples[2] = samples[1];
31         samples[1] = samples[0];
32         samples[0] = array1[i];
33
34         array2[i] = coeffs[0] * samples[0] + coeffs[1] * samples[1] + coeffs[2] *
           ↪ samples[2];
35     }
36 }
37
38
39
40 int main(int argc, char *argv[])
41 {
42     kernel(array1, arra2, coeffs, 64);
43     return EXIT_SUCCESS;
44 }
45

```

Listing 22: FIR filter kernel in C