Alma Mater Studiorum – Università di Bologna
in cotutela con Université Bretagne Sud

DOTTORATO DI RICERCA IN

Ingegneria elettronica, telecomunicazioni e tecnologie dell'informazione

Ciclo 35

**Settore Concorsuale: Elettronica**

**Settore Scientifico Disciplinare: ING-INF/01 Elettronica**

# Integrated Programmable-Array accelerator to design heterogeneous ultra-low power manycore architectures

**Presentata da:** Rohit Prasad

**Coordinatore Dottorato**

**Prof. Aldo Romani**

**Supervisore**

**Prof. Luca Benini**

**Supervisore**

**Prof. Philippe Coussy**

**Esame finale anno 2022**

# Integrated Programmable-Array accelerator to design heterogeneous ultra-low power manycore architectures

*Rohit Prasad*

Thesis Directors

Philippe Coussy & Luca Benini

Thesis Supervisors

Kevin J. M. Martin & Davide Rossi

2022

Academic thesis, which with the approval of the Lab-STICC, UMR 6285, Université Bretagne-Sud (France) & Department of Electrical Energy and Information Engineering "Guglielmo Marconi", Università di Bologna (Italy) will be presented for public review in fulfilment of the requirements for a Doctor of Philisophy in Electronics Engineering.

# Abstract

There is an ever-increasing demand for energy efficiency (EE) in rapidly evolving Internet-of-Things end nodes. This pushes researchers and engineers to develop solutions that provide both Application-Specific Integrated Circuit-like EE and Field-Programmable Gate Array-like flexibility. One such solution is Coarse Grain Reconfigurable Array (CGRA). Over the past decades, CGRAs have evolved and are competing to become mainstream hardware accelerators, especially for accelerating Digital Signal Processing (DSP) applications. Due to the over-specialization of computing architectures, the focus is shifting towards fitting an extensive data representation range into fewer bits, e.g., a 32-bit space can represent a more extensive data range with floating-point (FP) representation than an integer representation. Computation using FP representation requires numerous encodings and leads to complex circuits for the FP operators, decreasing the EE of the entire system. This thesis presents the design of an EE ultra-low-power CGRA with native support for FP computation by leveraging an emerging paradigm of approximate computing called transprecision computing. We also present the contributions in the compilation toolchain and system-level integration of CGRA in a System-on-Chip, to envision the proposed CGRA as an EE hardware accelerator. Finally, an extensive set of experiments using real-world algorithms employed in near-sensor processing applications are performed, and results are compared with state-of-the-art (SoA) architectures. It is empirically shown that our proposed CGRA provides better results w.r.t. SoA architectures in terms of power, performance, and area.

# List of Figures

# List of Tables

# Nomenclature

| | |
|---|---|
| $\mu J$ | microJoule |
| $\mu m^2$ | microMeter Square |
| $\mu W$ | microWatt |
| $AGU$ | Address Generation Unit |
| $ALAP$ | As Late As Possible |
| $ALU$ | Arithmetic Logic Unit |
| $ALU8$ | 8-bit integer based Arithmetic Logic Unit |
| $API$ | Application Programming Interfaces |
| $ASAP$ | As Soon As Possible |
| $ASIC$ | Application Specific Integrated Circuit |
| $ASIP$ | Application-specific Instruction Set Processor |
| $AXI$ | Advanced eXtensible Interface |
| $BA$ | Base Address |
| $BB$ | Basic Block |
| $CCA$ | Canonical Correlation Analysis |
| $CDFG$ | Control Flow Data Graph |
| $CFG$ | Control Flow Graph |
| $CGRA$ | Coarse Grain Reconfigurable Array |
| $CONV$ | Convolution |
| $CPU$ | Central Processing Unit |
| $CR$ | Control Register |
| $CRF$ | Constant Register File |
| $CT$ | Computerized Tomography |

| | |
|---|---|
| $DARPA$ | Defense Advanced Research Projects Agency |
| $DFG$ | Data Flow Graph |
| $DLP$ | Data Level Parallelism |
| $DMAC$ | Direct Memory Access Controller |
| $DRA$ | Destination Register Address |
| $DS$ | Divide-Square-root |
| $DSP$ | Digital Signal Processing |
| $DWT$ | Discrete Wavelet Transform |
| $ECG$ | Electrocardiography |
| $EEG$ | Electroencephalography |
| $EMF$ | Eclipse Modeling Framework |
| $ERI$ | Electronics Resurgence Initiative |
| $FC$ | Fabric Controller |
| $FD-SOI$ | Fully Depleted Silicon On Insulator |
| $FFG$ | Fast-Fast Global |
| $FFT$ | Fast Fourier Transform |
| $FIR$ | Finite Impulse Response |
| $FLL$ | Frequency-Locked Loop |
| $FMA$ | Fused-Multiply-Add |
| $FP$ | Floating-Point |
| $FPGA$ | Field-Programmable Gate Array |
| $FPU$ | Floating-Point Unit |
| $GCC$ | GNU Compiler Collection |
| $GCM$ | Global Context Memory |
| $GNU$ | GNU's Not Unix! |
| $GP-CPU$ | General Purpose-CPU |
| $GPIO$ | General-Purpose Input/Output |
| $GPP$ | General Purpose Processor |

| | |
|---|---|
| $HDL$ | Hardware Description Language |
| $HDTV$ | High-Definition Television |
| $HLS$ | High Level Synthesis |
| $I2C$ | Inter-Integrated Circuit |
| $I2S$ | Inter-Integrated-Circuit-Sound |
| $I\$$ | Instruction |
| $iCache$ | Instruction Cache |
| $IEEE$ | Institute of Electrical and Electronics Engineers |
| $IIR$ | Infinite Impulse Response |
| $ILP$ | Instruction Level Parallelism |
| $IoT$ | Internet of Things |
| $IP$ | Intellectual Property |
| $IRF$ | Instruction Register File |
| $IS$ | Instruction Synchronizer |
| $ISA$ | Instruction Set Architecture |
| $JTAG$ | Joint Test Action Group |
| $KiB$ | Kilo Byte |
| $LN$ | Logarithmic Number |
| $LSB$ | Least Significant Bit |
| $LSU$ | Load-Store Unit |
| $LV$ | Loop Variable |
| $MAC$ | Multiply and Accumulate |
| $matMUL$ | Matrix Multiplication |
| $MCU$ | Microcontroller Unit |
| $MHz$ | Mega Hertz |
| $MIMD$ | Multiple-Instruction stream-Multiple-Data stream |
| $MISD$ | Multi-Instruction stream-Single-Data stream |
| $ML$ | machine Learning |

| | |
|---|---|
| *MOPS* | Million Operations Per Second |
| *MRI* | Magnetic Resonance Imaging |
| *MSB* | Most Significant Bit |
| *mSFU* | mini-SmallFloat Unit |
| *NMOS* | N-channel Metal–Oxide–Semiconductor |
| *NOP* | No Operation |
| *OPR* | Output Register |
| *PCA* | Principal Component Analysis |
| *PE* | Processing Element |
| *PMOS* | P-channel Metal–Oxide–Semiconductor |
| *PnR* | Place-and-Route |
| *PPA* | Power, Performance,m and Area |
| *PULP* | Parallel Ultra-Low-Power |
| *PWM* | Pulse Width Modulation |
| *RF* | Register File |
| *RISC* | Reduced Instruction Set Computer |
| *RRF* | Regular Register File |
| *SCMD* | Single Configuration Multiple Data |
| *SDH* | Software-Defined Hardware |
| *SIMD* | Single-Instruction stream-Multiple-Data stream |
| *SISD* | Single-Instruction stream-Single-Data stream |
| *SoA* | State-of-the-Art |
| *SoC* | System-on-Chip |
| *SPI* | Serial Peripheral Interface |
| *SRAM* | Static Random Access memory |
| *SSG* | Slow-Slow Global |
| *SVM* | Support Vector Machine |
| *TCDM* | Tightly Coupled Data Memory |

| | |
|---|---|
| $TLP$ | Thread Level Parallelism |
| $TP-FPU$ | Transprecision Floating-Point Unit |
| $tSFU$ | tiny-SmallFloat Unit |
| $TT$ | Typical-Typical |
| $UART$ | Universal Asynchronous Receiver/Transmitter |
| $ULP$ | Ultra-Low-Power |
| $UML$ | Unified Modeling Language |
| $UNIX$ | UNiplexed Information and Computing System |
| $UNUM$ | Unversal Number |
| $UTBB$ | Ultra-Thin Body and Buried oxide |
| $VLIW$ | Very Long Instruction Word |

# Contents

# Chapter 1

# Introduction

Internet of Things (IoT) devices are rapidly evolving, and therefore, there is an ever-increasing demand for ultra-low-power and energy-efficient computing architectures, i.e., IoT end nodes. Applying software-based optimizations to augment the performance of a system is not enough [1]. A good trade-off between architectural characteristics and computational capabilities must be considered to boost the performance of IoT end nodes [2]. Such resource-constraint ultra-low-power (ULP) architectures face challenges while executing algorithms involving near-sensor computing or embedded machine learning (ML). Multiple efforts to keep these architectures energy-efficient and flexible enough to cope with the evolving technologies are seen in recent research works [3, 4, 5, 6].

Coarse Grain Reconfigurable Array (CGRA) architectures hold the potential to fit the requirement of being energy-efficient and at the same time provide flexibility to support evolving algorithms involving near sensor computing or embedded ML. A CGRA consists of an array of Processing Elements (PE), which are relatively smaller and simpler than General Purpose Processors (GPP). These PEs are connected through a simple on-chip network to provide data movement within the array. Each PE is configurable to execute part of a kernel in parallel with other PEs in the array. To provide an efficient parallel execution, each PE features a small register file to temporarily keep data loaded from memory or immediate results, which can be shifted/moved to other PEs using the on-chip network, resulting in avoiding register spilling or memory traffic. CGRAs can provide silicon efficiency approaching that of an Application-Specific Integrated Circuit (ASIC) by exploiting spatial computation typical of dedicated hardware blocks while keeping the programmability of GPPs [7, 8]. Furthermore, CGRAs have always been competing for being adopted as high-performance accelerators [7, 8, 9, 10, 11].

Energy-efficient execution of fixed-point workloads is prevailing in the recent ULP architectures. However, emerging IoT end nodes demand support for operations on datatype with high dynamic data range, i.e., floating-point (FP) numbers [12]. An alternative is to port FP operations into low-cost fixed-point operations [13] but porting an FP application into its fixed-point equivalent is laborious and demands in-depth knowledge of the application domain. Besides, porting an FP application into a fixed-point is not always energy-efficient due to the execution of instructions required for organization and normalization of such operations to overlook the dynamic range of FP numbers,

resulting in significant overheads [14].

An optimization technique to meet the computational demands of an application uses mixed, i.e., floating-fixed point representation. Such mixed representation is featured in State-of-the-Art (SoA) FP capable low-power architecture end-nodes like Cortex-M processors [15]. Again, this approach requires a tedious manual analysis of the applications for format selection between float and fixed for tuning the dynamic range and results in software overheads for such on-the-fly conversions. Furthermore, using such mixed representation results in degradation of the overall computational efficiency of Cortex-M processors, like, energy dissipated while managing the pipelines and register files in Cortex-M4 processor due to flushing or stalls [16].

A noticeable number of commercial architectures fail to turn off the FP units while fixed-point operators are executing in the cores; this also further lowers the energy efficiency of such architectures [17].

Approximate computing is a technique where the computing system does not compute the precise result but computes an approximated result. In this technique, to boost the performance of a computing system, a trade-off to the quality of computation with the effort spent is realized, resulting in fewer circuits and less energy consumption. An emerging paradigm of approximate computing is *transprecision* computing [1] that aims to improve the energy efficiency of low-power systems by adopting multiple FP formats to satisfy the accuracy requirements of the target application without manual adjustments. *Transprecision* computing is an alternative for applications demanding operations with high dynamic data range (i.e., FP operations) in IoT end nodes. *Transprecision* computing provides both hardware and software-based control mechanisms to fine-tune the approximation of computations in an application. Integration of *transprecision* computing in ULP architectures holds the potential to achieve high energy efficiency while executing applications requiring high dynamic range FP operations [17].

This thesis explores the design of an energy-efficient ULP CGRA with native support for FP computation by leveraging *transprecision* computing. Different design optimizations and architectures are presented for developing the CGRA with support for multiple FP datatype, including SoA IEEE 754-2008 standard FP and new custom *transprecision* FP datatype. A design is incomplete without the aid of compilation tools, so the contributions in the compilation toolchain to add support for FP in CGRA are also presented. To envision the proposed CGRA as an energy-efficient hardware accelerator, system-level integration of CGRA in a System-on-Chip is performed to demonstrate the applicability of CGRA as an energy-efficient hardware accelerator by using real-world algorithms employed in near-sensor processing application fields (i.e., image, audio, bio-signals, and embedded Machine Learning). Finally, two sets of results are presented in this thesis. First, CGRA with a 4x2 PE array is compared with a single-core SoA processor resulting in CGRA achieving a maximum of 10.06× better latency performance and consuming 12.91× less energy, with an area overhead of 1.25× only. Second, CGRA with a 4x4 PE array is compared with an 8-cores SOA processor sub-system resulting in the CGRA sub-system achieving a maximum of 4.20× better

latency performance and consuming 7.80× less energy, while CGRA sub-system also being 2.19× smaller than 8-cores SoA processor sub-system.

## Motivation

CGRAs are a raw coarse-grained implementation of the perception of reconfigurable computing proposed in the 1960s [18]. CGRA architectures were first introduced in the 1990s [19, 20] and have been developing extensively since the 2000s [21, 22, 23, 24]. CGRAs continuously attract both academia and industries, as CGRAs can provide near-ASIC energy efficiency and performance while maintaining software-like programmability with post-fabrication [11, 25, 26, 27, 28]. Figure 1.1 provides an overview of the positioning of hardware accelerator and processor architectures, i.e., Application-Specific Integrated Circuit (ASIC), Application-Specific Instruction Set Processor (ASIP), General Purpose Processor (GPP), Graphics Processing Unit (GPU), Field-Programmable Gate Array (FPGA), and Coarse Grain Reconfigurable Array (CGRA) with three design corners, where all architectures are implemented with the same operating parameters [7, 8, 11, 25, 29]. Breakdown of Figure 1.1 is as follows:

- ASICs are custom logic designs with fixed functions, making them the most energy-efficient architectures with the highest computing performance. However, these architectures are comprised of custom and fixed functions with limited libraries and support. Hence, they are least flexible among all the architectures in Figure 1.1.

- ASIPs are designed for a specific application domain, and these architectures exhibit an instruction set designed to accelerate most executed and critical functions. These architectures provide a trade-off between energy efficiency, computing performance, and flexibility.

- GPPs are designed to provide maximum flexibility to support all applications. Hence, these architectures are least energy-efficient and have minimum computing performance among all architectures in comparison in Figure 1.1.

- GPUs exhibit a large parallel array of cores designed for rendering graphics. These architectures have relatively high energy consumption but somewhat similar computing performance and flexibility as that of ASIPs.

- FPGA consists of a large number of configurable logic blocks that are connected through programmable interconnects. Each configurable logic block is essentially an array of programmable gates. These architectures provide high flexibility. However, these architectures provide low energy efficiency and computing performance due to unavoidable overheads with such fine granularity.

- CGRAs are outset to provide a balance between these three design corners. These

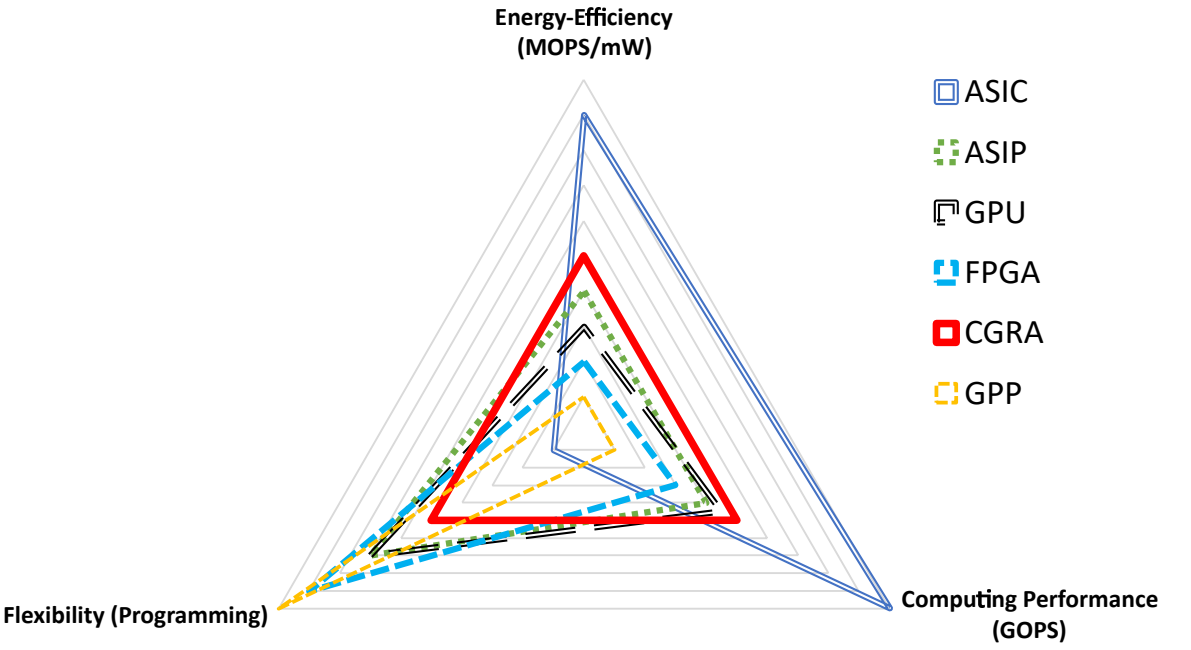



Figure 1.1: Three design corners of six architectures, implemented with the same operating parameters.

hybrid architectures are essentially systolic arrays [1] that are expanded to be programmed using instructions, making them coarser than FPGAs in terms of flexibility. CGRAs exhibit simple architecture to provide moderately high energy efficiency and computing performance.

GPPs are the most flexible architectures, ASICs are the most energy-efficient and provide the highest computing performance. On the other hand, CGRAs provide a balance between these three design corners.

In academia, the researchers consider CGRAs as a firm competitor for mainstream computing architectures because CGRAs provide a good trade-off among efficiency, performance, and flexibility for a certain application domain [7, 8, 30, 31] and also because of extensive supports by organizations like the Defense Advanced Research Projects Agency (DARPA) [32]. The objective of DARPA ERI (Electronics Resurgence Initiative) is software-defined hardware (SDH), i.e., to enable hardware to provide near-ASIC performance (≈10×) while maintaining the programmability for data-extensive kernels/algorithms. AHA Agile Hardware Project is an initiative to enable an agile hardware development flow. Under this project, especially for the rapid development of CGRAs, three domain-specific languages have been developed to generate individual components for CGRA. These are (1) PEak for processing tiles, (2) Lake for memory tiles, and (3) Canal for the interconnect [33].

[1] Systolic array is a network of tightly coupled data processing elements. These processing elements perform the same logic operation with different data at the different timestamp.

In industry, Samsung presented their 8K high-definition television (HDTV) and Exynos SoC featuring a CGRA accelerator [10, 34]. The CGRA intellectual property (IP) cores are present in the satellite payload of Astrium, where the IP cores are developed by PACT Incorporated [24]. An Intel project to integrate CGRAs into its Xeon processor was initiated in 2016 [35]. Other companies also have related projects, plans, prototypes, or products like DRP [36] and DAPDNA [37].

Ever-increasing computational demand from applications attracts researchers/industries to implement FP-like computations (i.e., computing FP workload using integer-based operators or converting FP workloads into fixed-point workloads) in CGRAs [11, 38, 39, 40, 41, 42]. A recent trend to equip IoT platforms with FP unit is also seen in micro-controller units like M4 and M7 [43]; a factor leading to such trend is with technology node scaling below 40nm, the cost of FP operation is getting near 1pJ per operation [44, 45], so it has become affordable in terms of absolute power to use FP in IoT. This gave us an incentive to implement FP operators with variable dynamic range and variable precision in a CGRA targeting the ULP domain and applications involving near-sensor computing and embedded ML.

## Contribution

This thesis contributes to the following characteristics of practicing CGRAs as hardware accelerators in computing platforms.

1. Design of CGRA with support for FP computation: the main contribution of the thesis is the design of an energy-efficient ULP CGRA with support for FP computation. Multiple design optimizations and their architectures to add support for multiple FP datatype in CGRA are presented.

2. Efficient mapping of FP operations: the changes in the hardware must be propagated to the compilation flow, and integration of new features in the toolchain must produce efficient mappings of the FP operations. This thesis presents the techniques used and the challenges tackled during the change propagation from hardware to compilation flow.

3. Implementation of Flexible-AGU: to improve the performance of CGRA, address generation is decoupled from the compilation flow, and a hardware Flexible-Address Generation Unit is introduced in the PEs of the proposed CGRA. The Instruction Set Architecture (ISA) width of the CGRA is limited; encoding of data required to compute the address and decoding instructions in the hardware was a challenge and is also presented in this thesis.

4. System-level integration in an SoC: the point of presenting an energy-efficient ULP CGRA is to improve the overall performance of the computing platforms. This integration is necessary to envision the CGRA as a hardware accelerator

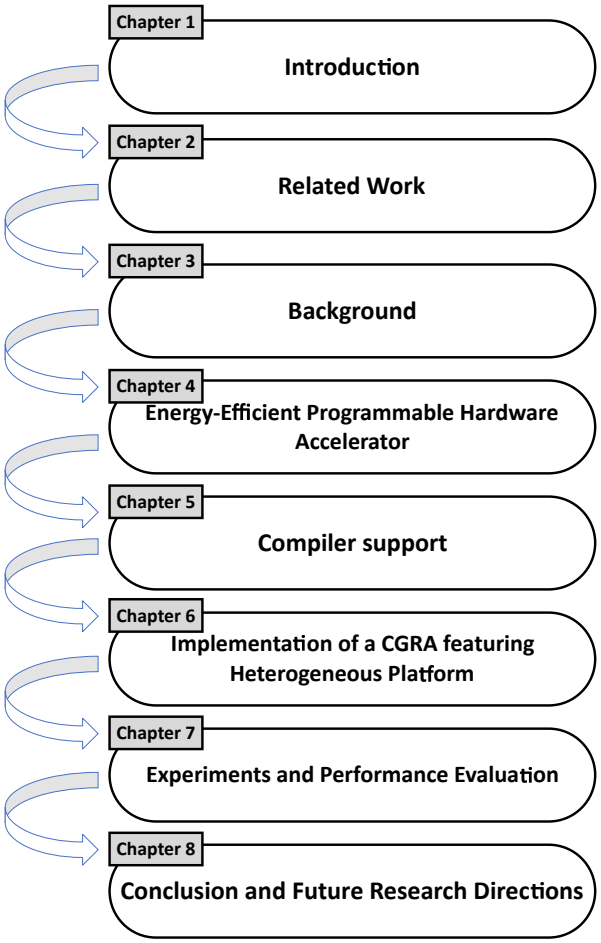



Figure 1.2: Organization of chapters

properly. The challenges in integrating the CGRA sub-system in an SoC to propose a heterogeneous platform are presented in this thesis.

5. Benchmark: this thesis empirically shows that the CGRA can improve the overall performance when integrated into a system. A wide range of real-world applications used in near sensor computing and embedded ML is used to benchmark the CGRA w.r.t. SoA architectures and all the experimental results obtained using gate-level simulation silicon-proven PULP platform [46].

## Organization

Figure 1.2 shows the outline of the thesis and follows the points described in the contribution section. In chapter 2 and chapter 3, the related work and the necessary background information associated with the contribution of this thesis are presented respectively. Chapter 4 discusses the architecture and design optimizations of the CGRA with support for multiple FP datatype. Then, chapter 5 discusses the compiler support to enable energy-efficient support for FP in CGRA. In chapter 6, a heterogeneous platform fea-

turing the proposed CGRA sub-system is presented. Chapter 7 presents the evaluation methodology and performances of the proposed CGRA and the heterogeneous platform. Finally, in chapter 8, the thesis summarizes the given work and suggestions for possible future research directions.

## List of Publications

1. R. Prasad, S. Das, K. J. M. Martin, G. Tagliavini, P. Coussy, L. Benini and D. Rossi, "TRANSPIRE: An energy-efficient TRANSprecision floating-point Programmable archItectuRE," 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 2020, pp. 1067-1072, https://doi.org/10.23919/DATE48585.2020.9116408

2. S. Das, R. Prasad, K. J. M. Martin and P. Coussy, "Energy Efficient Acceleration Of Floating Point Applications Onto CGRA," ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Barcelona, Spain, 2020, pp. 1563-1567, https://doi.org/10.1109/ICASSP40776.2020.9054613

3. R. Prasad, S. Das, K. J. M. Martin, and P. Coussy, "Floating Point CGRA based Ultra-Low Power DSP Accelerator," Journal of Signal Processing Systems (2021). https://doi.org/10.1007/s11265-020-01630-2

4. Journal to be submitted for publication. The title is as follows:
   R. Prasad, G. Tagliavini, K. J. M. Martin, P. Coussy, L. Benini and D. Rossi, "Hopalong: A Heterogeneous Cluster for Transprecision Computing on IoT End Nodes".

# Chapter 2

# Related Work

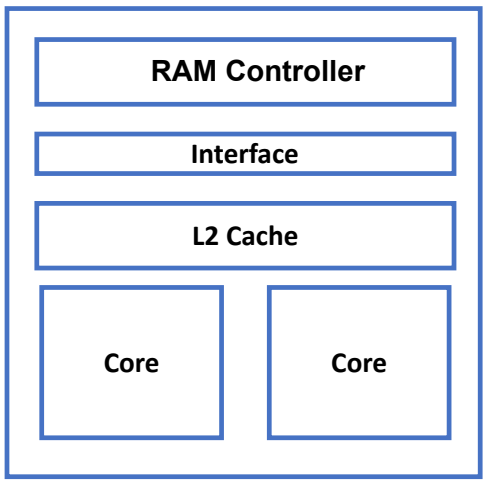
## Contents



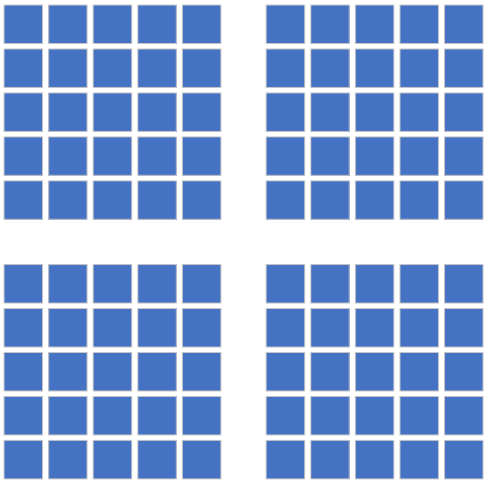
This chapter discusses the topics of (1) Taxonomy of architectures, (2) Floating-Point (FP) support in low-power circuits, (3) Transprecision computing, (4) CGRA architecture, (5) CGRAs supporting FP computations, (6) Compiler support for the development of CGRA, and (7) PULP SoC [46] based heterogeneous computing systems featuring different hardware accelerators. These topics serve as the related work for the thesis.

## 2.1 Taxonomy of Architectures

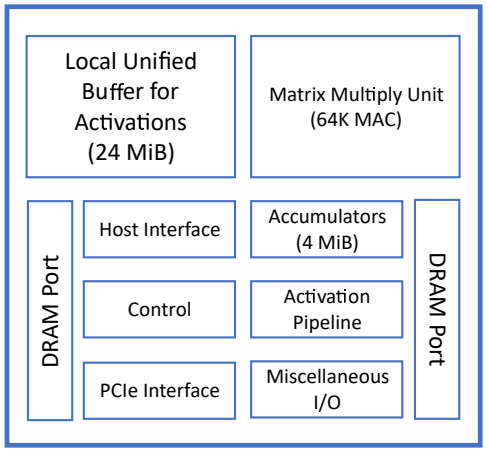
There are billions of IoT devices that execute digital signal processing (DSP) applications and the majority of them are battery operated. In this thesis, the proposed CGRA targets DSP applications and adopts multiple optimization techniques to boost the energy efficiency of the entire system. Figure 2.1 shows six architectural options that are interesting for digital signal processing applications. These architectures are described below.



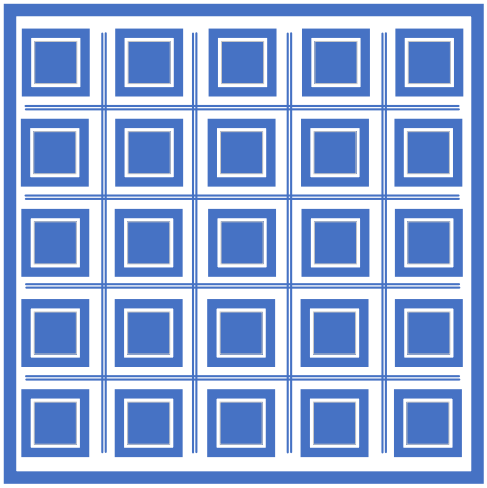

(a) GPP



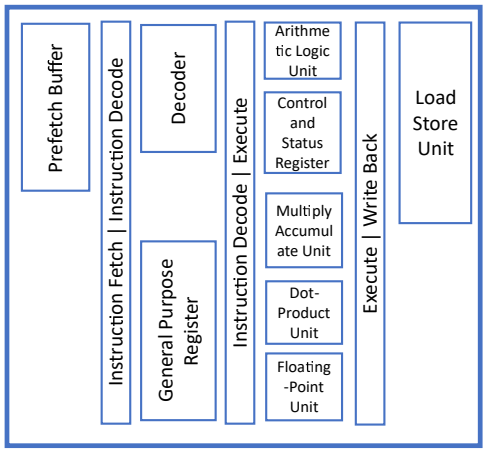
(b) GPU



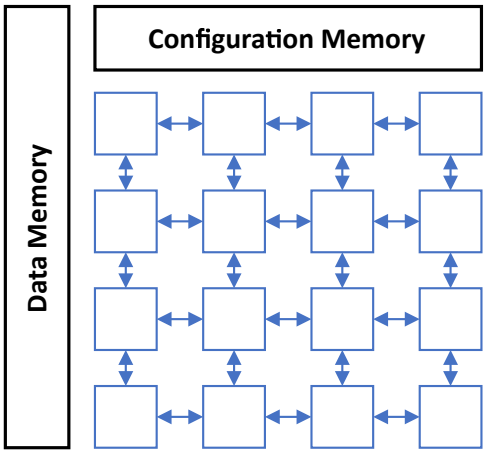

(c) ASIC (TPU [47])



(d) FPGA




(e) ASIP (RI5CY [48])




(f) CGRA

Figure 2.1: Architecture options for Digital Signal Processing applications

### ASIC

Figure 2.1c shows the organization of an Application-Specific Integrated Circuit (e.g., Tensor Processing Unit or TPU [47]). Typically, ASICs are non-programmable and target only one application. ASICs were a preferred solution where the level of integration was limited, like Very Large-Scale Integration (VLSI) with the introduction of the first MOS Integrated Circuit by General Microelectronics in 1964 [49]. However, multiple solutions have been proposed over the past decades, and also, due to technological advancements, ASICs are not the only choice of architecture to be integrated into a system. Typically, ASICs are introduced for matured architectures with the highest level of optimizations. ASICs are able to reach the highest energy efficiency and computing performance for a target application.

### ASIP

Figure 2.1e shows the organization of an Application-Specific Instruction Set Processor (e.g., RI5CY [48]). An ASIP is designed for a specific application domain, and the instruction set architecture (ISA) of these architectures is designed to accelerate the frequently occurring functions or critical functions. The cost of hardware and the power consumption is relatively lower than other architectures in Figure 2.1 because ASIPs usually target predictable computing, so the design optimizations can be much higher while providing a substantial amount of flexibility for the target domain applications.

### GPP

Figure 2.1a shows the organization of a General Purpose Processing unit with two cores. A GPP runs complex tasks and aims to facilitate maximum flexibility for all applications. Typically, GPPs are limited to perform certain concurrent tasks as these architectures consume a large amount of energy with little computing performance. The compiler and Operating System are designed to support all applications, which also requires a huge amount of engineering design time. Overall, building a GPP-based computer infrastructure from scratch is very expensive.

### GPU

Figure 2.1b shows a typical Graphic Processor Unit. A GPU consists of many smaller specialized cores. These cores work together to deliver a massive performance by dividing the task and processing them in parallel. Initially, GPUs began as specific ASIC to accelerate specific 3D rendering tasks. Gradually, GPU cores evolved from fixed-function engines to become programmable and flexible cores. Nowadays, GPUs are also capable

| Type | Power Consumption | Strengths | Constraints |
|---|---|---|---|
| **ASIC** | Ultra-Low | Small silicon footprint | Fixed function |
| **ASIP** | Low to Ultra-Low | Targets application domain | Complex instructions and tasks |
| **GPP** | High | Flexible | Few cores |
| **GPU** | High | Highly parallel cores | Large silicon footprint |
| **FPGA** | Medium | Re-programmable | Programming Complexity |
| **CGRA** | Low to Ultra-Low | Targets application domain | Simple instructions and tasks |

Table 2.1: Comparison of architectures

of handling a wide range of applications and compete against other general-purpose parallel processors.

## FPGA

Figure 2.1d shows a typical Field-Programmable Gate Array architecture. an FPGA consists of many logic blocks which can be programmed and reprogrammed to perform numerous functions at any point in time. These logic blocks are interconnected through a programmable interconnect network. Development on an FPGA is relatively cheaper due to nearly non-existent Non-Recurring Engineering (NRE) cost than other architectures in Figure 2.1. Such affordability makes FPGAs their preferred choice for designers to perform experiments or even realize any architecture design (mainly in the early development stages).

## CGRA

Figure 2.1f shows the organization of a typical Coarse Grain Reconfigurable Array architecture. A CGRA consists of an array of simple and small processing elements that are connected through an interconnect network. CGRA can efficiently exploit parallel computing due to the availability of an array of low-cost processing elements and also due to affordable data movement of the spatial-temporal data between the processing units. CGRAs were studied at first for accelerating the inner loop computation of the applications. Over time, CGRAs have evolved to become a competitive solution for high-performance accelerators.

Table 2.1 presents a comparison table of the six architectures. CGRA is the essence of this thesis, and all the topics presented in this thesis involves the development of CGRA architecture, its optimization techniques, and its associated toolchain.

## 2.2 FP support in low-power circuits

Using a data format with a fewer precision bits format to represent decimal FP numbers suffers substantially from the loss of accuracy. Applying such data formats with limited precision bits in real-world DSP applications, where intermediate results are frequently accumulated, can lead to undesirable loss of accuracy and devastating consequences like ARIANE 5: Flight 501 Failure (i.e., failure of the maiden flight of the Ariane 5 launcher on 4 June 1996) due to occurrence of errors while converting 64-bit FP number to 16-bit signed integer value [50].

FP operations are much more complex than integer operations, so it is familiar to trade-off between speed and accuracy while implementing FP computing modules. An example of combining two FP datatype modules (i.e., IEEE 754-2008 Decimal FP and IEEE 754-1985 Binary FP formats) is presented in [51, 52], where hardware resource sharing resulted in saving up to 58% of less area w.r.t. the combined areas of individual modules, with a negligible impact on the critical path delay for the combined module over the individual decimal modules. Representing a decimal digit in IEEE 754-2008 standard requires 4-bit, and this leads to a larger datapath, making it unsuitable for the FP units in the proposed ULP CGRA.

A good balance between delay, area, and accuracy for FP representation is provided by IEEE 754-2008 standard. However, FP computing with increased precision bits results in larger energy consumption, making those computing systems less attractive for ULP architectures. An alternative is to explore custom FP formats for enabling low-power circuits with FP operations. One such effort is presented in [53], where a Logarithmic Number System (LNS) based LNS Unit (LNU) efficiently performs FP operations by translating FP operations into simple integer operations. However, a fallacy to this implementation is that the LNUs take a larger area than their single-precision FP counterparts. These LNUs become attractive when shared among several cores. They can be 4.1× more energy-efficient than a similar area design using four private FPUs while executing standard nonlinear processing kernels.

An FP format similar to IEEE 754-2008 FP standard is Universal Numbers (UNUM). UNUM features an additional tag for self-description of the represented value (i.e., number of bits for exponent and precision), enabling them to represent variable-sized FP numbers [54]. The silicon implementation of an ASIC featuring UNUM-based ALU [55] showed 7% less memory footprint w.r.t. an IEEE 754-2008 single-precision FP unit. UNUM-based ALU also offered a more comprehensive range for half-precision FP formats with an increased datapath complexity, making the design choice less attractive for the proposed CGRA to avoid a complex interconnect network for PEs.

A new set of custom FP formats called smallFloat formats that can also be regarded as an alternative to IEEE 754-2008 FP standard (i.e., *float* and *float16*) are *binary16alt* and *binary8* [1]. These new FP formats exhibit the same dynamic range as that of IEEE counterparts with fewer precision bits. smallFloat formats are useful in applications that require low precision, like ML applications. The silicon implementation of a 32-bit

| Format | #Bits | Exponent | Mantissa | Range |
|---|---|---|---|---|
| IEEE *float* | 32 | 8 | 23 | 1.2 x $10^{-38}$ - 3.4 x $10^{38}$ |
| *binary16alt* | 16 | 8 | 7 | 1.2 x $10^{-38}$ - 3.4 x $10^{38}$ |
| IEEE *float16* | 16 | 5 | 11 | 5.9 x $10^{-8}$ - 6.5 x $10^{4}$ |

Table 2.2: FP formats used in IoT devices

RISC-V core based ASIP, including a multi-format FPU featuring both IEEE 754-2008 standard FP formats and smallFloat formats, showed the multi-format FPU could achieve a latency gain of 1.67× w.r.t. an IEEE *float* based FPU without losing any precision in the results, and reducing the system energy by 37% [56]. These results make smallFloat formats very attractive for the proposed CGRA because the target domain applications of CGRA also require high dynamic range, low precision, and low power consumption.

## 2.3 Transprecision Computing

ML workloads executed by modern near-sensors applications demand both high dynamic data range and low-power consumption. IEEE 754-2008 standard provides double-precision (64-bit; IEEE *double*) and single-precision (32-bit; IEEE *float*) FP representations to realize the high dynamic data range demand, but such FP computations are too costly in terms of energy consumption for IoT end nodes leveraging ULP architectures. The half-precision FP representation introduced in IEEE 754-2008 standard (16-bit; IEEE *float16*) is a trade-off between dynamic range and energy cost, which is suitable for approximate computing. Over the years, significant advancements in approximate computing are seen. However, there is still a considerable demand for high dynamic data range and low-power consumption from IoT end nodes.

Typically, IEEE *float* and *float16* are used in IoT devices but these FP representations either high energy consumption (in IEEE *float*) or lack high dynamic data range (in IEEE *float16*). Recently, a new custom datatype has been introduced, called *binary16alt*, to cover the fallacies of these IEEE FP formats, i.e., provides a high dynamic range as of IEEE *float* and at the same time consumes similar energy as of IEEE *float16*. Table 2.2 shows these FP formats.

Range of SoA ML applications like Convolutional Neural Networks [57] and Temporal Neural Networks [58] relies on ML models which can tolerate lower precision computations without losing their accuracy [59] and IEEE *float16* and *binary16alt* FP datatype is well suited for such applications. Adopting such lower precision FP formats enables a system to achieve high performance and also be energy-efficient while executing such ML applications, but embracing such FP formats will require full software support and thorough analysis for tuning precision of FP variables to safely replace the high-cost FP formats with low-cost FP formats [60] in those applications.

*Transprecision* computing is an emerging paradigm of approximate computing which aims to fulfill those demands of providing high dynamic data range and be energy-efficient [1]. In *transprecision* computing, the intermediate results preserve the dynamic range, and precision is tuned to fulfill the accuracy requirements of an application, allowing programmers to adapt to a new FP format with fewer bits. In [56], a silicon-proven architecture has been introduced, which is an integral part of the *transprecision* computing framework by enabling programmers to fine-tune the workloads ranging from algorithms and software down to hardware and circuits.

## 2.4 CGRA Architecture

The applications used in IoT devices keep evolving due to increasing computational demands in IoT end nodes. Going for highly optimized and hardwired ASIC architectures could lead to a very aggressive time-to-market to deliver the next iteration of current designs. In this scenario, employing programmable hardware accelerators (i.e., CGRAs) as co-processors in such systems can ease the before-mentioned deadlines. CGRAs have proven to be effective in accelerating target applications by achieving silicon efficiency near an ASIC by exploiting spatial computation typical of dedicated hardware while keeping programmability typical of GPP [7].

Recent CGRAs have demonstrated effectiveness in executing fixed-point workloads because fixed-point units employ simpler architecture of integer arithmetic units [61]. However, there is an increasing demand for support for FP computation in emerging IoT end nodes [13]. Few CGRAs have been presented in the past to support FP computations. However, efficient and dedicated support for FP still lacks in those CGRAs [8, 11, 39, 40, 41, 42] which makes them less attractive for adoption in IoT end nodes or ULP architectures.

CGRA architectures have an ample design space as these architectures feature (1) tightly or loosely coupling to host processors, (2) types of on-chip interconnects and the interactions between producer-consumer resources, (3) reconfigurability/programmability of the array, (4) ways to control the executions in array, (5) support for different types of parallelism. Typical characteristics of a CGRA that motivate a ULP CGRA to execute FP computations efficiently are discussed below.

### Reconfigurability

The salient feature of a CGRA is its ability to be reconfigured (i.e., flexibility). The three types of reconfiguration are:

1. Dynamic reconfiguration where CGRAs use small controllers to fetch new configuration every clock cycle from configuration buffers.

2. Static reconfiguration where configuration bits of CGRAs remain fixed for the whole execution of a loop.

3. Hybrid reconfiguration where CGRAs use a combination of both static and dynamic configurations.

Notably, in the case of dynamic reconfiguration, a full reconfiguration follows every execution cycle, and to better utilize such array, a simpler instruction fetching mechanism w.r.t. GPPs is employed for iterating through a loop body without any control flow like seen in ADRES [21], Silicon Hive [62], and MorphoSys [63]. Hybrid reconfigurable CGRAs like RaPiD [64] where static and dynamic reconfigurability is controlled by a small sequencer or PACT [65], where CGRA can self trigger events for partial reconfiguration and consumes a substantial amount of time. In contrast to these CGRAs, KressArray [66] features a fully static reconfiguration approach where CGRA is configured before a loop is executed. In such CGRAs, efficiently mapping of kernels onto hardware is prioritized, which is done at a higher level in the compiler.

While all before mentioned CGRAs avoid executing control statements mainly to minimize hardware complexity, our priority was to opt for dynamic reconfiguration of CGRA and execute a complete kernel by implementing low-cost control flow support in CGRA by combining efficient mapping of control statements in compiler and simple design choices in hardware.

## Scheduling and Issuing

Execution of operations and data transfers can be controlled in a CGRA either by dynamic, static, or hybrid reconfiguration. Such reconfiguration can be achieved in a compiler by scheduling the static code schedules like seen in VLIW processors [67] or the approach used in out-of-order processors can be used where processors issue instructions the availability required operands [68] or by combining static and dynamic reconfiguration and static and dynamic scheduling [21, 62, 63, 65, 66, 69, 70].

In the case of dynamic reconfiguration, the dynamic execution of control instructions consumes more power. In static scheduling, all possible paths, even the slower ones that might be taken infrequently during the execution, are scheduled and consumes more resources. It can be concluded that the design choices depend on the target application domain. In our case, we opted for a dynamic scheduling approach due to the simple architecture of the PEs. All architectural decisions are driven by the energy consumption of the modules implemented in PEs, including low power consuming controllers for fetching new instructions.

### Parallel Processing

With the availability of the number of cores of a CGRA, it is recognizable to implement support for different types of parallelism. In [65, 66], dynamic scheduling is implemented through a distributed event-based control mechanism, and thus implementing Thread Level Parallelism (TLP) is relatively simple and cheap. In this case, the execution of independent small loops (with a combined resource can fit) on the CGRA can be mapped on different parts of the distributed control resource. A different approach is to design an architecture with centralized control mechanism where parallel threads can run by implementing additional controllers or extending the central controller to support the execution of parallel threads concurrently. Such extensions can increase power consumption but might be suitable for certain code parts by saving datapath energy and energy consumed while fetching configuration data.

Alternatively, TLP can be converted into a combination of Instruction Level Parallelism (ILP) and Data Level Parallelism (DLP). At compile time, such a combined approach can exploit kernels with multiple threads by scheduling those threads together as one kernel and then selecting the appropriate combination of the scheduled kernel at run time [71].

In our CGRA, parallel processing is achieved by combining the implementation of additional controllers in hardware and exploitation of ILP and DLP in the kernels.

### Interconnect Network

The type of reconfiguration scheme plays an essential role in the selection of interconnecting networks. Interconnect networks are required in both phases of execution in CGRAs, i.e., (1) *configuration phase*, where a network is required to distribute the instructions to each core, and (2) *compute phase*, where a network is required to route the data flow. PEs of a CGRA can be interconnected in a wide range of connections (1) with direct connections (i.e., interconnect network), (2) with Register Files or other memories, and (3) with IO ports. In ULP architectures, energy efficiency is the main focus, and low power networks could be implemented to achieve the low-power consumption goal.

In the case of dynamic reconfiguration, if there is a centralized configuration network, then the network for configurations or instructions is accessed as frequently as the network for data flow by the cores of CGRA. In such a scenario, merging the two networks could be a convenient choice and the use of proper routing of data and instructions can avoid conflicts due to shared resources. If there is no centralized configuration memory, then it may result in high power consumption due to the implementation of separate networks for configurations and instructions.

In the case of static reconfiguration, a network dedicated for *configuration phase* is

used once per execution cycle, and a network dedicated for *compute phase* is used extensively, depending on the kernel. A low-cost interconnect network (like 2D mesh) should be implemented to obtain high energy efficiency in such a scenario.

### Register Files

Compilers for CGRA schedule the operations and their respective data flow is routed over the connections between the PEs. These connections could be direct or latched, or connections that go through RF, depending on the type of operations are being scheduled. So, most compilers treat RFs as interconnects that can span over multiple cycles instead of temporary storage. Thus, RFs in between cores of a CGRA are included during design space exploration for interconnects. During such routing, (1) the number of interconnecting wires, (2) their topology, i.e., RF size, (3) their location, and (4) the number of ports contribute to determining the interconnectivity of cores of a CGRA.

### Instruction Set Architecture

Implementing simple Instruction Set Architecture (ISA) could help minimize the cost of instruction fetching and decoding due to simple decoder design and also leads to the simple hardware design of the cores of CGRA, resulting in low-power consumption.

For example, a 32-bit ISA of RISC-V encodes a lot of information in each instruction. To execute an operation, the RISC-V core fetches 32-bit instruction and applies a complex decoder to decode the already fetched instruction. This gives a little room for optimization in the instruction fetch and instruction decode stage of the pipeline architecture of the RISC-V core. In the case of conditional instructions, most of the instruction space is non-utilized which also results in unavoidable power consumption due to fetching and decoding of 32-bit instruction where most of the instruction bits are zeros. While a simple ISA implementation with less number of bits (like 21-bit ISA of the proposed CGRA) can avoid such a complex decoder or fetching of instructions with large unused spaces (i.e., in case of conditional instructions), resulting in relatively low-power consumption than the RISC-V core featuring 32-bit ISA.

## 2.5 CGRA with FP support

CGRAs can efficiently exploit both Data Level Parallelism (DLP) and Instruction Level Parallelism (ILP) in a Digital Signal Processing (DSP) application because these architectures take advantage of the execution pattern characterized by repeated execution of a combination of operators like addition, subtraction, and multiplication. Mainly, DSP applications demand sharing or shifting of immediate data with the neighboring

entities, which CGRAs can efficiently execute by featuring cheap $MOVE$ operations due to simple cores and interconnect networks.

Real-world DSP applications are rapidly evolving and demand a more complex execution pattern than ever. These applications also demand FP operations to satisfy the wide dynamic range of the input/output data. In addition, recent CGRAs lack the flexibility to efficiently execute such demanding, complex execution patterns and support for FP operations. Microcontrollers like M4 and M7 [43] feature FP units and have been implemented below 40nm process node. Their cost of executing FP operation is near 1pJ [44, 45], demonstrating that cost-efficient implementation of FP operators in IoT devices is attainable.

Very few works have been demonstrated where a CGRA is featuring support for FP operations. These reconfigurable/programmable architectures are limited with low ILP and/or low DLP while adding support for FP operations. Imagine [38] is one of the early programmable architectures that featured FP unit hardware for single-precision FP operations, and the execution model of Imagine follows data streaming. RASP [39] is a DSP architecture featuring an array of coarse-grain computing elements with support for single-precision FP operations. RASP can efficiently execute sub-algorithms of filters applied in radar signal processing, namely, FFT, IIR, and matrix-multiplication, to help improve the overall computational efficiency of the radar system. Imagine and RASP lack the capability to execute an entire application, while the proposed CGRA can efficiently execute a whole application.

FloRA [40] and Wave CGRA [11] adopted a different approach of combining the integer-based operators to perform FP computations, resulting in a complex execution model, increased interconnect width, and degraded output quality. Another such architecture is Butter array [41], which combines integer-based addition, multiplication, and additional units for packing and rounding the output, resulting in extra area overhead and a lack of native FP support. An approach to converting FP workloads into fixed-point workloads before computing output is SDT-CGRA [42], which features a flexible interconnect network to support multiple computation models with reduced hardware complexity but the conversion of all FP inputs into fixed-point before the output is calculated brings a significant overhead and loss of accuracy. In conclusion, all of these architectures are too demanding in terms of power consumption to be considered for IoT end nodes or as ULP architecture.

## 2.6 Compiler support

Compiler plays a crucial role in determining the efficiency of a CGRA because a compiler inputs a human-readable application code to produce the target machine code (bit-stream). While compiling, a compiler faces challenges like (1) producing code according to the level of granularity of the target hardware system or (2) maintaining high utilization of the cores (with efficient exploitation of algorithmic parallelism) in

the CGRA array.

Typically, dataflow languages expose the potential parallelism of any application. Still, while generating required machine code for parallel General-Purpose-Processors (GPPs), the compiler considers the overall dataflow network of the application only (i.e., Data Flow Graph of the application). Due to such considerations by the compilers, potential parallelism extracted from the internals of the code is left out (i.e., a loop structure, pipelining of intra-agents, or task splitting and rescheduling). Allowing a CGRA compiler to exploit both ISA and micro-architecture of the CGRA enables the CGRA compiler to efficiently optimize the application to obtain maximum performances from the target CGRA.

A CGRA compiler could efficiently exploit architectural support of multilevel parallelism (i.e., data, instruction, memory, speculative, and thread) to obtain higher utilization of the cores in the CGRA array. Still, there are substantial trade-offs that must be taken into account. For instance, superscalar processors are more efficient by dispatching multiple instructions per clock cycle than scalar processors (Single Instruction Single Data processors). Hence superscalar processors can efficiently utilize ILP; multi-core processors are more efficient than superscalar ones due to the efficient utilization of Thread Level Parallelism (TLP); out-of-order processors exhibit better performance than in-order processors due to efficient utilization of speculative parallelism. In CGRAs, a good trade-off between area and power can lead to the implementation of efficient parallelism because when area and power overheads counterbalance the (latency) performance gain, the area efficiency and energy efficiency are degraded. Such degradation depends on both applications and architectures.

TRIPS [23] is a CGRA architecture that supports three modes of execution to support various parallelism (i.e., DLP, ILP, and TLP) [72]. Another example is Polymorphic Pipeline Array [73], which supports fine-grained parallelism by exploiting ILP and TLP with the help of software pipeline and coarse-grained pipeline parallelism. CGRAs presented in [74, 75, 76, 77] exploit a technique of software pipelining to explore coarse granularity of loop-level or kernel-level parallelism but pipelining of operations or instructions does not comply with the typical characteristics of CGRAs because CGRAs mainly rely on energy-efficient spatial computations instead of time-multiplexed instruction or operation pipelines. Such CGRAs are specifically proposed to counter out-of-order processors which dynamically exploit ILP in hardware. CGRAs like [23, 30, 78, 79, 80] use spatial computation model to implement ILP. In this technique, order of execution and data preparation is done by static scheduling of a DFG. This technique is also used to accommodate multiple DFGs on a single array. Domain-specific CGRA like [22, 31] implement DLP in Single Configuration Multiple Data (SCMD) manner to reduce area and power overheads due to fetching of same instructions for multiple-data, i.e., provides a programmable interface to minimize memory accesses and bypass redundant address generation. CGRAs with centralized control schemes are presented in [80, 81, 82, 21] where execution and reconfiguration of all cores of CGRA occur in lockstep, i.e., using a large configuration to reduce energy consumption and maximize resource utilization. However, integrating such CGRAs into a larger (hierarchical) CGRA could result in a CGRA with coarse-grained TLP.

On the other hand, in [83], a distributed control scheme has been implemented in a CGRA to execute multiple tasks simultaneously. Dataflow technique can be exploited to implement TLP in CGRAs like TRIPS [23] which uses static dataflow to enable spatial parallel execution of multiple DFGs, and CGRAs like SGMF [79] which exploits dynamic dataflow to execute multiple DFGs in an overlapping manner. TFlex [84] is a pipelined CGRA that implements speculative parallelism by introducing a predicate/branch predictor that uses all predicates in a block (i.e., code in both True and False paths of a condition statement) next block. In TFlex, if there is a misprediction, then the speculative block is flushed from the pipeline.

The compiler support used for the proposed CGRA plays a crucial part by providing quick and efficient mapping solutions for the target applications for the proposed CGRA. The mapping process is automated through a software tool implemented using an Eclipse-based modeling framework and code generation solution called Eclipse Modeling Framework (EMF) [85, 86]. The compiler shares a common framework with a High-Level Synthesis (HLS) tool called GAUT [87]. The contributions of this thesis to the existing compilation flow [88, 80, 89, 90] are (1) adding support for custom FP formats and multi-cycle operation support, and (2) decouple the data required for address generation from the mapping process and encode the data in the generated instructions that are to be decoded by a dedicated hardware module to calculate the corresponding address in the CGRA during execution.

## 2.7 Heterogeneous Computing Systems

A series of work has been presented on PULP-Cluster [91] extended with hardware accelerators to combine general-purpose computing with domain-specific processing capabilities. Flumine [5] is a PULP architecture [91] based SoC featuring four OpenRISC [92] ISA-based GPPs and two cluster-coupled hardware accelerator engines, namely, HWCE and HWCRYPT. Both engines can directly access the coupled data space as of the GPPs. The engines are used in an inter-leaved form that allows each engine to access the data space with zero-copy data exchange with other processing elements on the cluster. The architectural solution of combining GPP and hardware accelerators, presented in Flumine, demonstrated improvements of more than one order of magnitude in energy and time w.r.t. a purely software-based solution. NeuroCluster [3] is a many-core platform featuring RISC-V-based GPPs and co-processors (NeuroStream) to offer a scalable and energy-efficient processor-in-memory solution to compute deep convolution networks. NeuroCluster achieved 3.5× better energy efficiency w.r.t. a GPU implemented in a similar process node. XNOR Neural Engine (XNE) [4] is a flexible hardware accelerator for binary neural networks integrated into a microcontroller system for edge computing and is implemented in a 22nm process node. XNE achieves 21.6 fJ/Op at 0.4V, and XNE is also able to execute SoA ResNet-34 under 2.2 mJ/frame at 8.9 frames/second. A heterogeneous cluster featuring RISC-V-based GPPs and a 32-bit integer datatype compatible programmable accelerator is presented in [6]. The heterogeneous cluster can surpass a similar homogeneous cluster (i.e., cluster featuring GPPs only) by 4.8× in

latency and 4.5× energy-efficiency performances while executing a wide range of DSP applications.

The heterogeneous cluster presented in this thesis explores the design space with different configurations of combining multi-core 32-bit RISC-V-based CPUs enhanced with ISA supporting SIMD-Style vectorization and support both IEEE 754-2008 standard FP formats and custom smallFloat formats and proposed CGRA with 4x4 PE array featuring multiple optimization techniques to pull off the highest performances from the heterogeneous cluster in terms of latency, power, and area.

## 2.8 Summary and Concluding Remarks

In this chapter, a series of topics which act as the related work for this thesis are discussed, i.e., (1) an overview of the taxonomy of architectures highlighting the architecture options for Digital Signal Processing applications, (2) different low-power architectures featuring FP support, (3) an emerging paradigm of approximate computing called Transprecision computing, (4) CGRA architecture, (5) CGRA with FP support, (6) Compiler support for CGRA, and (7) PULP-SoC architecture-based multiple heterogeneous computing systems. In the next chapter, the background of the work presented in this thesis is presented.

# Chapter 3

# Background

## Contents



This chapter discusses the background work, particularly, topics of (1) architecture of Integrated Programmable Array which lays the basis of the proposed CGRA, (2) Flynn's taxonomy (3) Eclipse Modeling Framework, (4) Compiler support for proposed CGRA, (5) Transprecision computing-based Floating-Point units, and (6) PULP architecture [91] are discussed, which provide a foundation of the research work presented in this thesis.

## 3.1 Integrated Programmable Array

Integrated Programmable Array (IPA) is an energy-efficient CGRA accelerating ultra-low-power domain application featuring 32-bit integer datatype [6, 80, 90]. IPA features a 4x4 PE array (Figure 3.1) and provides an architectural template for the proposed CGRA in this thesis. PEs are interconnected via a simple 2D mesh torus interconnect network and share data with the host CPU through the Tightly Coupled Data Memory (TCDM). Global Context Memory stores the context and data that the IPA Controller distributes before the start of execution in IPA. IPA is evaluated using various ultra-low-power domain applications executing 32-bit integer operations. IPA is able to achieve an

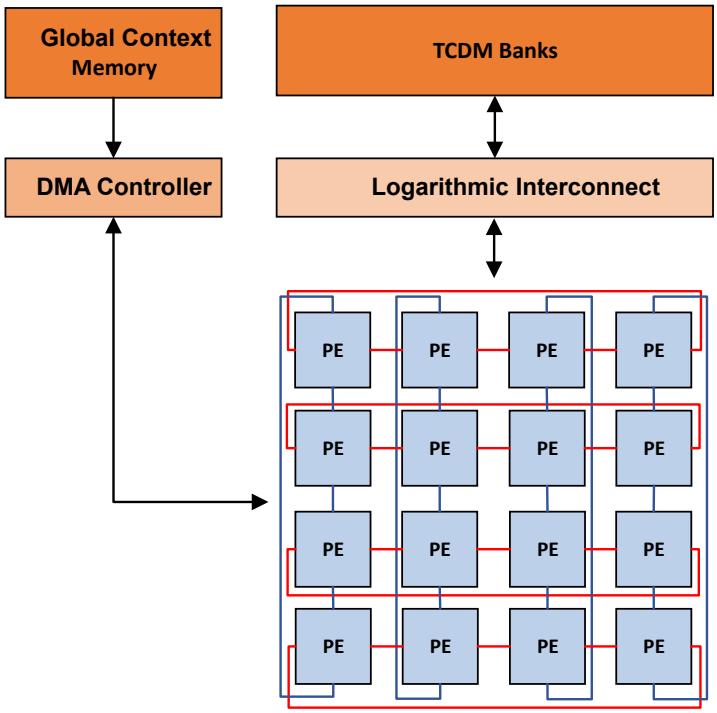



Figure 3.1: Integrated programmable Array integrated system

average performance of 507 MOPS and an average energy efficiency of 142 MOPS/mW at an operating voltage of 0.6V. IPA is able to surpass a RISC-V-based GPP, namely or10n [93] by 6× in terms of latency performance and 10× in energy efficiency. Below are the short descriptions of the components of the IPA sub-system.

## Context Memory

Context Memory stores the configuration data for the PEs. Before computation in the PEs array, the configuration data are loaded to their respective PEs through a bus network. Depending on the PE array size, the Context Memory is sized adequately to fit two configuration data.

## Direct Memory Access Controller

Direct Memory Access Controller (DMAC) fetches context and data from Context Memory and loads them onto their respective PEs. DMAC is an FSM that decodes the configuration data to determine the number of instructions and constants for each PE and redirects them to their respective PEs. After context data has been loaded in their respective PEs, DMAC initiates a signal to start execution in the PE array. DMAC also handles the synchronization between CGRA and the host CPU.

### TCDM and Logarithmic Interconnect

TCDM consists of multiple memory banks and is connected to the processing elements (i.e., CGRA and host CPU) through a high-throughput ultra-low-latency interconnect network [94]. Both sub-systems (i.e., CGRA sub-system and Host CPU sub-system) share data through TCDM.

## 3.2 Flynn's taxonomy

Flynn's taxonomy [95] categorizes parallel computer architectures into four classes. These are

1. SISD (single-instruction stream-single-data stream), i.e., a simplex overlapped processor is limited by data dependencies,

2. SIMD (single-instruction stream-multiple-data stream), i.e., a master instruction is applied over a vector of related operands, and SIMD processors are efficient when a common data storage is used,

3. MISD (multiple-instruction stream-single-data stream), i.e., specialized streaming organizations using multiple-instruction streams on a single sequence of data and such kind of organizations are not useful anymore, and

4. MIMD (multiple-instruction stream-multiple-data stream), i.e., multiprocessors.

## 3.3 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [85, 86] evolved around the idea of object-oriented modeling. EMF is a Java framework and code generation facility for rapidly building tools and applications based on a structured data model. EMF transforms the data models into well-organized, correct, and simple customizable Java code. Figure 3.2 represents the basic elements of the EMF model and the references used to generate a Java implementation. Mainly, a model is created and defined in Ecore format. Ecore is the subset of Unified Modeling Language [1] (UML) Class diagrams and Java code is generated from an Ecore model.

[1]Unified Modeling Language (UML) is a visual language or a general-purpose modeling language. UML aims to define a standard technique to visualize the design of a system. UML consists of a set of developed diagrams to help developers specify, visualize, construct, and document the product of software systems [96].

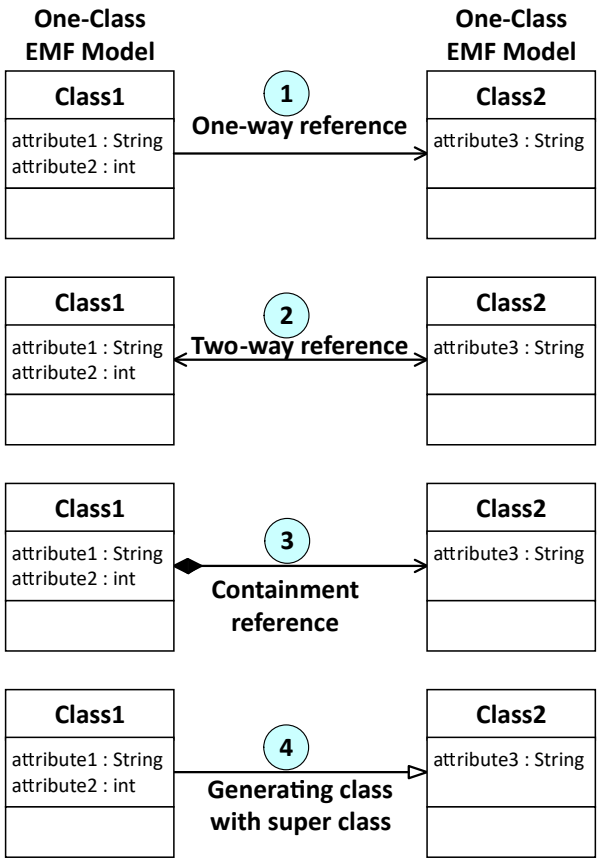



Figure 3.2: EMF model elements

In Figure 3.2, a single class called *Class1* with two attributes: *attribute1* of type String and *attribute2* of type int is shown. There are four types of references used in the EMF model of Control and Data Flow Graph (CDFG) for generating Java implementation. These references represent the association between the two classes.

1. One-way reference expands Class1 with Class2, and this type of reference is used to access Class2 from Class1 (i.e., one-way access).

2. Two-way reference removes the access exclusivity between the classes and defines the two-way relationship between two classes that is navigable both ways.

3. Containment reference is one of the important types of association because a containment reference identifies the parent or owner of a target instance.

4. Inheritance in EMF is represented by the fourth arrow in Figure 3.2. Inheritance is used to extend the definition of a superclass (Class2) by inheriting another subclass (Class1) and also to override the definition of methods from a superclass. In Java, there are three rules for inheritance, i.e. (1) a class can inherit from only one class, (2) an interface (i.e., collection of all abstract methods in a class) can inherit from other interfaces, and (3) a class can implement multiple interfaces. Inheritance in EMF is more permissive than in Java. However, Java code generated using the

EMF model always respects these rules of Java.

## 3.4 CGRA Compiler

In the compiler, (1) CGRA is modeled as a bipartite directed graph with the operator and register nodes. (2) An application is modeled as a Control and Data Flow Graph (CDFG), which itself is a Control Flow Graph (CFG) where building blocks are called Basic Blocks (BB), and each BB is a Data Flow Graph (DFG). A DFG is a bipartite directed acyclic graph made up of operation and data nodes, and arrows connecting the nodes in the DFGs represent the data dependencies. (3) The homomorphism between the CGRA model and DFG drives the problem of mapping application onto CGRA as a sub-graph finding problem. Each of these three parts is discussed below.

### CGRA Model

CGRA is modeled as a bipartite-directed graph with the operator and register nodes. A time extended model [77] of CGRA is used to determine the timing from the connection between operator and register nodes. There are two types of operator nodes in the CGRA, i.e., (1) nodes representing arithmetic and logic operations (+, -, *, AND, OR, etc.) or/and memory operations (LOAD and STORE), and (2) memorization operator [90] (memorization node is correlated to a register and represents the operation of explicitly retaining a value in a local register).

In Figure 3.4, a 2x1 PE array is shown with three regular registers, an ALU, and an output register. Not all connections are illustrated for simplicity, and the PEs are connected via a mesh torus interconnect (See Figure 4.1). Mapping of applications onto CGRA is discussed later in this section with the help of this CGRA model and an equation (See Figure 3.4).

### Application Model

An application is modeled as a Control and Data Flow Graph (CDFG) and featuring support for the execution of control flow in CGRA makes the proposed CGRA execute a complete application without involving a host processor to process any underlying control statements.

A CDFG combines (1) Control Flow Graph (CFG) i.e., Graph G = (V,E), where V = { $v_1$, $v_2$, ... $v_n$ } is a finite set of nodes and E $\subset$ VxV is a control flow relation of directed sequence edges, and (2) Data Flow Graph (DFG) or BB i.e., Graph

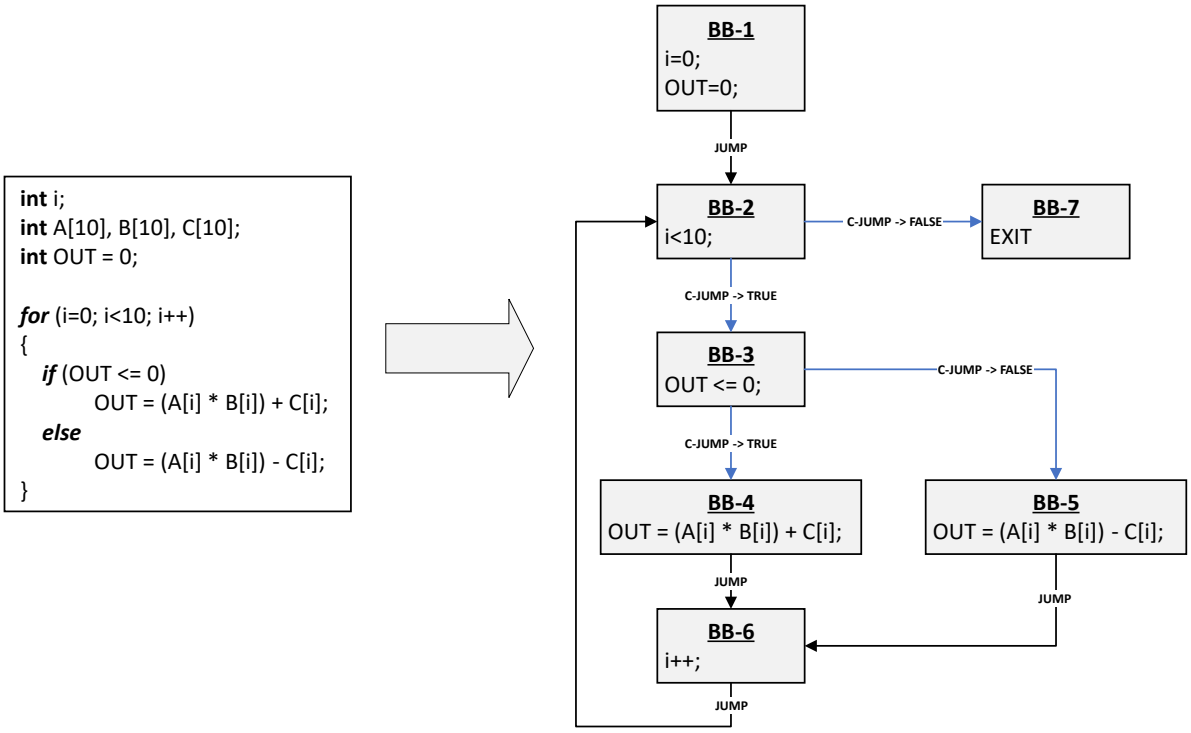



Figure 3.3: Application to CDFG

G = (V,E), where V = { $v_1$, $v_2$, ... $v_n$ } is a finite set of nodes and E $\subset$ VxV is an asymmetric data flow relation of directed data edges.

Figure 3.3 shows the CDFG representation of a sample kernel. A rectangle represents each BB, and the flow from one BB to another is represented by an arrow which is lead by a simple $JUMP$ operation. In the case of conditional-$JUMP$ or (C-$JUMP$), there are $TRUE$ and $FALSE$ paths as shown in the Figure 3.3. The execution of CDFG is as follows:

BB_1 $\rightarrow$ { either BB_2 ($START\ LOOP$) or BB_7 ($EXIT\ LOOP$) } ; if BB_2 $\rightarrow$ BB_3 $\rightarrow$ { either BB_4 or BB_5 } $\rightarrow$ BB_6 $\rightarrow$ BB_2 ($START\ LOOP$)

Such execution flow is preserved by executing the BBs in synchronization, i.e., while a BB is being executed, all PEs are engaged in executing the current BB. At the end of a BB, all PEs are synchronized, i.e., before a $JUMP$ or C-$JUMP$ statement, which allows all PEs to execute concurrently or sequentially. This makes a PE engaged in executing several BBs at different timestamps by mapping multiple operations and data onto a PE.

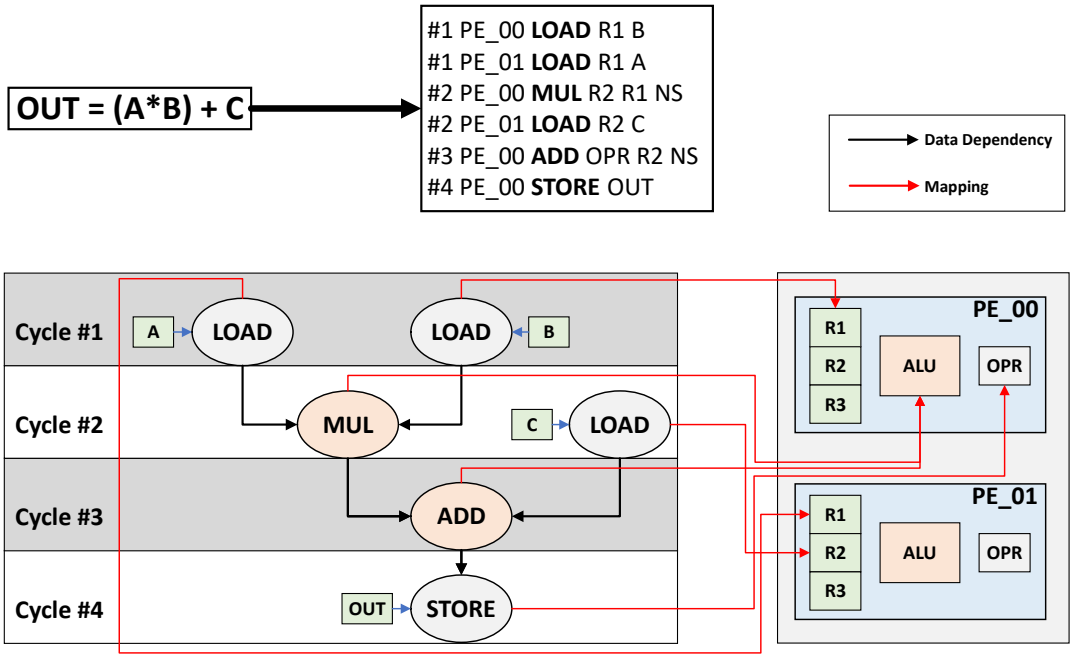



Figure 3.4: Mapping of BB_4 (See Figure 3.3) onto 2x1 CGRA model and outline of the generated assembly code

### Homomorphism

The two graphs, i.e., (1) DFGs in CDGF and (2) CGRA model, represents two homomorphic systems. First, DFG has three elements, i.e., (1) operation nodes, (2) data nodes, and (3) data dependencies and second, the CGRA model has three elements, i.e., (1) operators, (2) registers, and (3) connection between time extended PEs. As the two models are homomorphic, the compiler treats the mapping problem as a sub-graph finding problem [97, 89, 90], i.e., finding a DFG in the CGRA model graph. Figure 3.4 shows the mapping of a BB_4 from Figure 3.3 over 4 clock cycles using a 2x1 PE array.

## 3.5 Transprecision Computing based Floating-Point Units

### 3.5.1 smallFloat Unit (SFU)

Figure 3.5 shows the architecture of smallFloat Unit (SFU). SFU is a *transprecision* FP unit capable of executing *binary16alt* and *binary8* operations in addition to standard *binary32* and *binary16* operations [61]. SFU is evaluated using an SoC for ultra-low-power *transprecision* computing extending the PULPino [48] microcontroller architecture with SFU integrated into the RI5CY processor core [1]. Using precision tuning and vectorization, SFU is able to achieve 12% latency improvements and reduce the memory

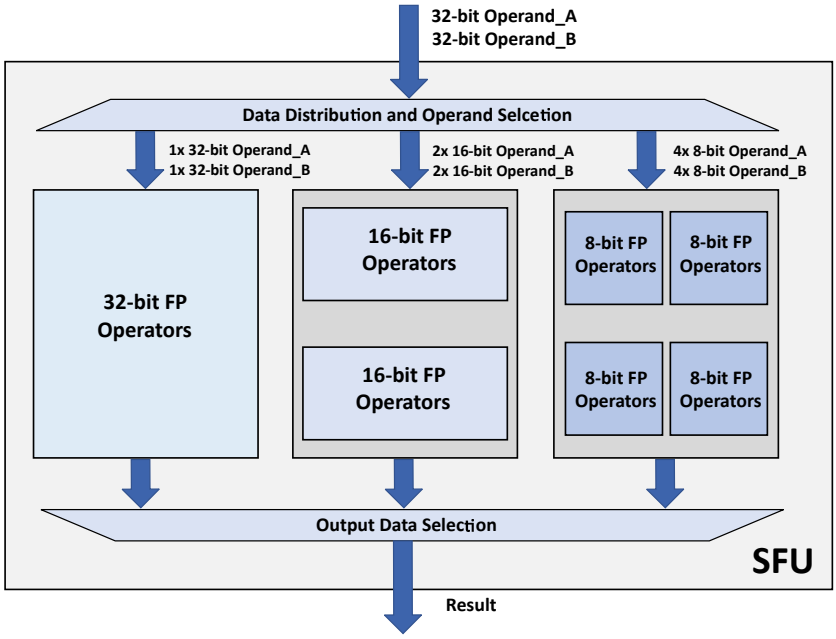



Figure 3.5: smallFloat Unit

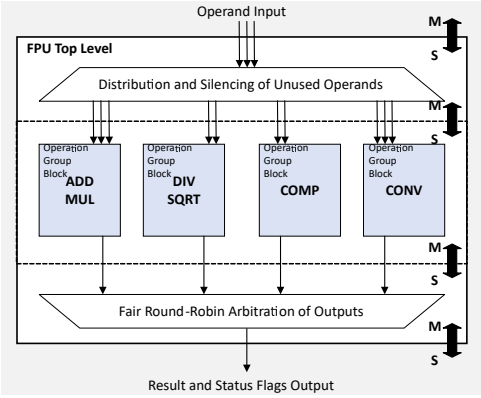
accesses by 27% on average while executing FP-intensive benchmarks w.r.t. the system executing standard FPs only. SoC is also able to consume 30% less energy by leveraging the knobs provided by the SFU w.r.t. only executing standard FP operations.

### 3.5.2 Transprecision FPU (TP-FPU)

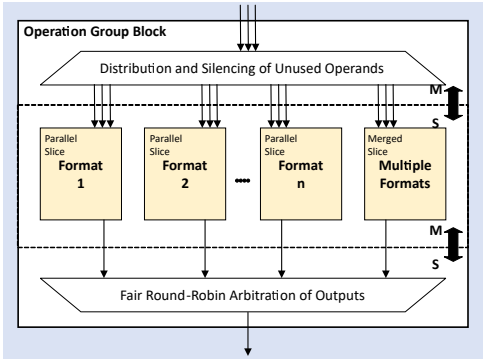
Transprecision Floating-Point Unit (TP-FPU) is a configurable open-source *transprecision* FP computing FPU shown in Figure 3.6 that is capable of executing 64-bit, 32-bit FP, 16-bit FPs, and 8-bit FP formats [56]. FPnew is capable of executing both scalar and SIMD-vectorized FP arithmetic and casting and packing operations.

#### TP-FPU Top Level

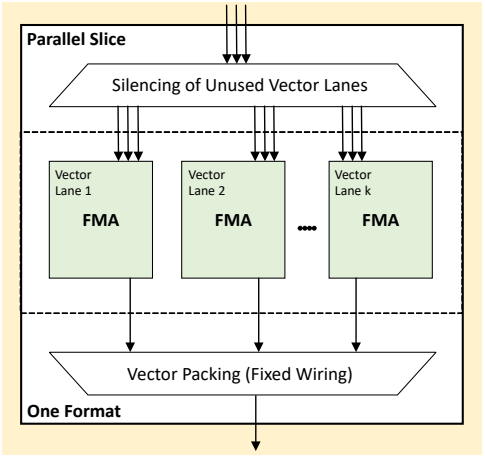
Figure 3.6a shows the top level of TP-FPU. TP-FPU can input a maximum of three FP operands per clock-cycle, alongside the control signals to determine the operation and format(s) required. An output leaves the unit along with a status flag augmented by the current FP operation according to IEEE 754-2008 standard. According to the class of instruction, input operands are routed to their respective operation group blocks. Output is collected through arbiters which takes the calculated results from the operation group blocks. During this process, unused operation group blocks are either clock-gated or datapath-gated to silence the unused branches in the TP-FPU.



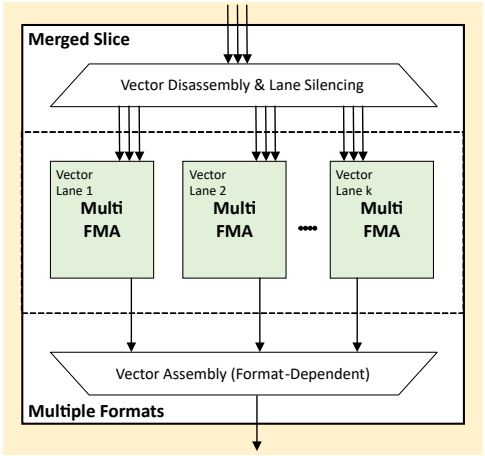

(a) TP-FPU Top Level




(b) Operation Group Block




(c) Parallel Slice




(d) Merged Slice

Figure 3.6: TP-FPU [56] and its underlying hierarchical blocks. M = Master port ; S = Slave port

## Operation Group Blocks

Figure 3.6b shows the organization and datapath of an operation group block. There are four operation group blocks in TP-FPU that are:

1. *ADDMUL*: addition, multiplication, and fused-multiply-add (FMA)
2. *DIVSQRT*: division, and square-root
3. *COMP*: comparison, and bit-manipulation
4. *CONV*: conversions among FP formats, and to/from integers

Each of these blocks features an independent datapath for their respective operations. In the case of a block featuring multiple FP formats operators, the blocks can feature multiple format slices that are instrumented either as parallel slices (i.e., specific to formats) or merged slices (i.e., for multiple formats). Figure 3.6c shows the organization in

a parallel slice that can only host one single FP format and allows flexibility to efficiently implement respective format operators but at the cost of increased total area. Unused format slices are clock-gated to reduce switching activities in those slices. Figure 3.6d shows a merged slice that can host multiple FP formats, which reduces their flexibility but decreases the total area by sharing hardware.

#### Silicon Implementation

Silicon implementation of a single-core SoC platform based on RI5CY core integrated with TP-FPU is done using a 22nm process node. The experiment results showed that while executing a multi-format application kernel, with adaptive voltage and frequency scaling, TP-FPU is able to achieve energy efficiencies between 178 Gflop/sW (on IEEE 64-bit ) and 2.95 Tflop/sW (on *binary8*), and a latency performance between 3.2 Gflop/s and 25.3 Gflops/s.

## 3.6 PULP Architecture

This section reviews the architecture of PULP SoC, particularly PULP architecture based a silicon-proven SoC design called Mr. Wolf [2]. SoC design includes two separate voltage and frequency domains [91, 2] :

1. SoC domain featuring a RISC-V CPU, L2 memory, ROM, and peripherals.

2. Cluster domain including tightly coupled multiprocessors.

### SoC Domain

Figure 3.7 shows the block diagram of PULP Soc with focus on SoC domain. SoC domain features a microcontroller unit (MCU) built around 512KiB of L2 memory and a two-pipeline stage RISC-V processor (called zero-RI5CY [98]), referred to as Fabric Controller (FC). FC is optimized for low power consumption and implements RV32IMC RISC-V ISA [99]. FC includes a 32-bit integer-based sequential multiplier with a latency of 3 clock cycles and a 32-bit integer-based divider with a latency of 35 clock cycles. The processor configuration presents a trade-off between power and performance while performing control-oriented tasks, i.e., IO management [98].

SoC features a full set of peripherals that are also found on advance MCUs, i.e., a parallel camera interface, Quad Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter (UART), Inter-Integrated Circuit (I2C), General-Purpose Input/Out-

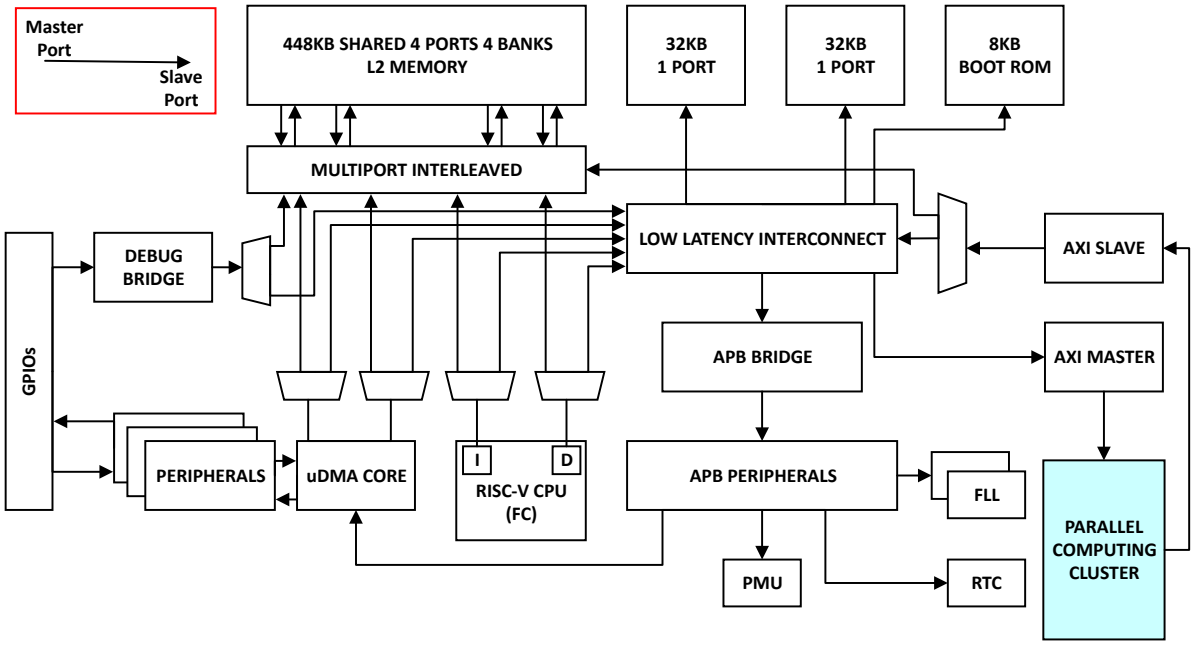



Figure 3.7: PULP SoC block diagram detailing SoC domain

put (GPIO), four Inter-IC-Sound (I2S), four-channel Pulse Width Modulation (PWM) interface, and a Joint Test Action Group (JTAG) interface for debugging.

FC, peripherals, IO, and parallel computing cluster share data through L2 memory. To perform efficient sharing of data among all functional units, a double-buffering mechanism is employed, i.e., overlapping of data transfers from peripherals and L2, and L2 to L1 memory, which results in high computing efficiency. To increase the access bandwidth of L2 memory and minimize the conflicts during parallel accesses through the six master ports present in the L2 memory interconnect (i.e., FC, $\mu$DMA, and parallel computing cluster), the L2 memory is organized in the following way:

- 448KiB memory arranged as four 112KiB word-interleaved logic banks, and
- two 32KiB private banks.

Splitting of each logic bank in L2 memory into eight physical memory banks allows power-gating of each bank and implements an incremental state-retentive mechanism. Each master in the PULP SoC can access all memory locations, as single address space is organized in the memory hierarchy. This eases the overall programmability of the system.

FC does not have instruction cache, so data and instruction accesses from FC need to be private and fast. When FC is active, a bandwidth of 3.2Gbps at 100MHz is available at the instruction port. If such bandwidth from shared memory is dedicated to FC, then this would degrade the performance of both FC and resources sharing data through L2 memory. To avoid such degradation, two 32KiB private banks are provided for FC to

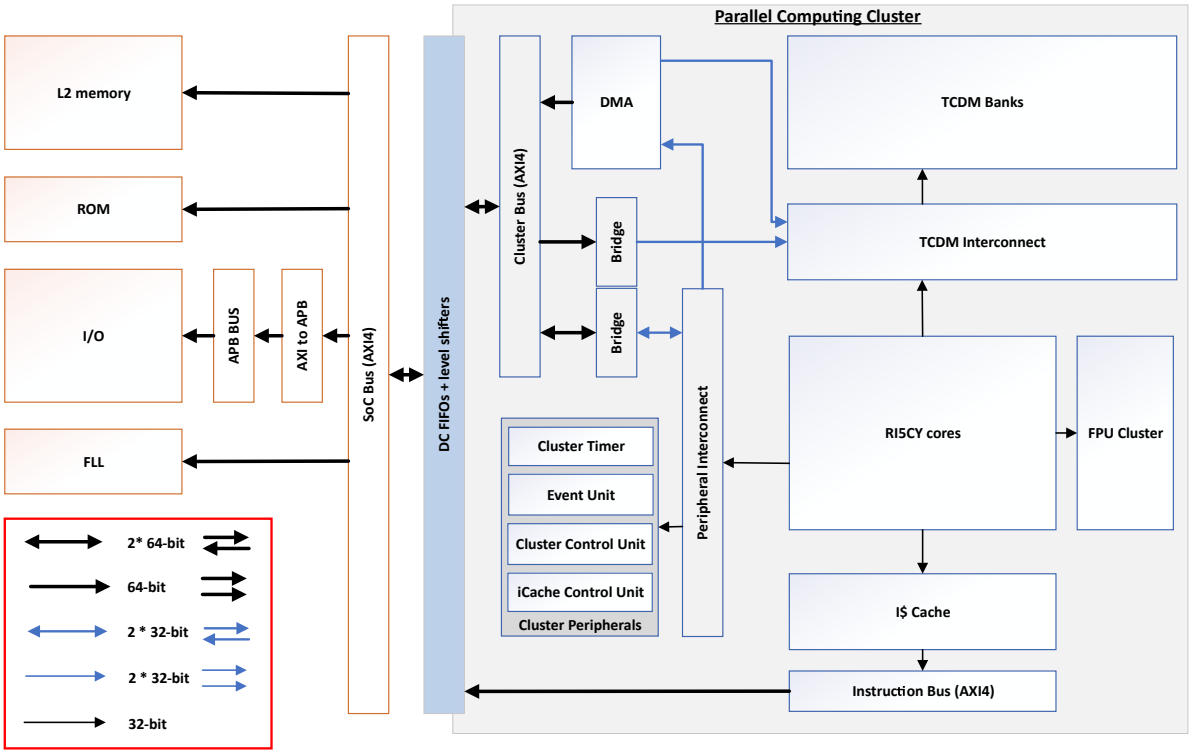



Figure 3.8: PULP SoC block diagram detailing Cluster domain

use privately for the stack, program, and private data. Such organization reduces the banking conflicts and improves the performance of FC (up to 2×) during the execution of applications involving high memory-intensive computations [2].

Two asymmetric AXI plugs are employed for connection with the parallel computing cluster, which are

1. a 64-bit wide for cluster-to-memory communication, and

2. a 32-bit wide for FC-to-cluster communication.

These asymmetrical plugs are implemented to save area because FC is the only master in the SoC domain that can generate up to 32-bit blocking transactions. DMA present in the cluster domain handles the high bandwidth data transfers through the 64-bit wide plug, which is connected to the AXI bus in the parallel computing cluster. Apart from high performance interconnect, PULP SoC also employs a low-cost APB subsystem for accessing the configuration registers of the different SoC IO peripheral IPs, i.e. (1) pad GPIO and multiplexing control, (2) timer, (3) clock and power control, (4) PWM controller, and (5) $\mu$DMA configuration port.

## Cluster Domain

Figure 3.8 shows the block diagram of PULP SoC with focus on Cluster domain. The parallel computing cluster is turned on and adjusted to the required voltage and frequency in the event of FC encounters highly intensive computation tasks and offloads them onto the cluster domain.

Parallel computing cluster includes eight RISC-V cores implementing the RVC32IMF instruction set [99], extended with custom instructions targeting energy-efficient digital signal processing, called *Xpulp* [48]. There are two sets of extended instructions, which are:

1. *XpulpV1*: This set of instructions are easily inferred by the compiler and includes (1) instructions for hardware loop for accelerating *for* loop statements, (2) LOAD/STORE with post-increment for accelerating stepped accesses to tensors and vectors, and (3) multiply and accumulate (MAC).

2. *XpulpV2*: This set of instructions exploits the built-in functions in the kernel code and includes (1) SIMD instructions dedicated to performing parallel arithmetic operations on 16-bit and 8-bit data, (2) bit manipulation instructions, and (3) instructions to support fixed-point arithmetic.

These custom instructions, when compared to a baseline RVC32IMF ISA, have improved the energy efficiency and performance of compute-intensive applications by up to 11× [2].

Cluster domain enables shared-memory parallel computations by including a 64-bit multi-banked L1 scratchpad memory, i.e., TCDM. TCDM is composed of 16 4-KiB SRAM banks and allows parallel memory requests with 1-clock-cycle latency. Such low latency is achieved by the implementation of a low-latency logarithmic interconnect. The logarithmic interconnect is characterized by a word-level interleaved design with round-robin arbitration, which results in a low contention rate [94]. Cluster peripherals (i.e., cluster timer, event unit, cluster control unit, and iCache control unit) and AXI-4 bus are accessed using a dedicated peripheral interconnect. A DMA controller is employed to manage the data movements between TCDM and L2 memory. The DMA controller supports a 2-dimensional addressing mode and is able to support a maximum of 16 transactions through the AXI-4 bus to hide the access latency of L2 memory.

The instruction cache (iCache) is implemented using latch-based memory that is able to save up to 4× the access energy of the instruction memory [100]. The latch-based design has significant area overhead w.r.t. SRAM-based implementation [100]. This overhead is compensated by sharing iCache among the cores and thus avoiding replication of instructions which is typically seen in private cache design employed in multi-core systems [101].

Apart from the classic software support for fast event management, synchronization, and

parallel thread dispatching, the parallel computing cluster also features an Event Unit to improve the performance and energy-efficiency of parallel workloads in the cluster domain. The Event Unit also handles the top-level clock gating of each core. Both the DMA controller and the Event Unit are connected to the dedicated ports present in each core for enabling fast and non-blocking accesses. Such design choice prefers the access to time-critical low-latency interconnect over peripherals.

### Configuration Employed for Gate-Level Simulation in the Thesis

PULP-Cluster presented in this thesis features eight 4-stage in-order RISC-V CPUs, namely RI5CY [48]. The open-source distribution, called PULP Platform [46], includes a set of IPs written in HDLs, a runtime software stack for low-power hardware support, and a compilation toolchain. The core design features a clock-gated mode to minimize the static and dynamic power waste while the cores are idle. The cores do not include a private data cache; instead, PULP-Cluster includes a shared TCDM organized in 16 word-level interleaved memory banks. These banks are connected through a non-blocking interconnect network to keep a minimum banking conflict [94]. A private instruction cache of 512 Byte is included in each core linked up on a shared instruction bus. L2 memory and peripherals are managed by the DMAC that allows access through an AXI4-compliant interconnect. Cluster events are managed by an Even Unit, enabling fast event management, parallel thread dispatching, and synchronization between cores [2]. This Event Unit also controls Clock-gating between the proposed CGRA and the RI5CY cores. There is an FPU cluster which the RI5CY cores share, and the ratio between SFU [61, 56] cores and RI5CY cores is 2:1, and the scheduling of FP operations is deterministic [14]. In chapter 6, a heterogeneous cluster is introduced to improve the performance of PULP-Cluster further [91] for near sensor processing and embedded ML.

## 3.7 Summary and Concluding Remarks

In this chapter, a discussion on the background work is presented, particularly CGRA architecture, Integrated programmable Array [80], Eclipse Modeling Framework, CGRA compiler, transprecision computing-based FP units, and PULP architecture are discussed. In the next chapter, a detailed discussion on the architecture of the proposed CGRA and the different design optimizations are presented.

# Chapter 4

# Energy-Efficient Programmable Hardware Accelerator

## Contents



This chapter explores the architecture and design optimizations for implementing FP support in the proposed CGRA. In the past, adding FP support in ULP reconfigurable architectures imposes many limitations and makes it challenging to keep the FP operations under the tight ULP power budget. Still, a recent trend to equip IoT platforms with FP Unit (e.g., microcontrollers like M4 and M7 [43]) has brought down the cost of an FP operation (under 40nm process node) near to 1pJ/op [44, 45]. This gave a significant incentive to explore and implement different FP datatype in our proposed CGRA.

## Contribution and Outline of the Chapter

In this chapter, each component of the CGRA sub-system is discussed, along with different optimization techniques employed in the proposed CGRA to implement FP support. Below are the highlights of this chapter:

- Implementation of Flexible-Address Generation Unit to decouple software-based address generation in CGRA.

- Implementation of Instruction Synchronizer to synchronize between multi-cycle operations and implement clock-gating between different modules.
- Design 1: Implementation of IEEE 754-2008 Standard compliant FP module in CGRA.
- Design 2: Implementation of *transprecision* computing compliant FP module in CGRA.
- Design 3: A version of CGRA featuring both IEEE 754-2008 Standard FP and *transprecision* FP datatype.
- Design 4: Implementation of a highly optimized version of CGRA featuring *transprecision* FP datatype for designing a heterogeneous cluster.
- Design 5: Exploration and implementation of 8-bit integer for accelerating Neural Network (NN) applications.
- Computation Model of the proposed CGRA discussing the mechanism of context loading and execution.

Finally, a summary and concluding remarks are provided at the end of this chapter.

## 4.1 CGRA Design Optimizations

In this section, five CGRA design optimizations are presented. All of these designs are part of design space exploration to add support for energy-efficient FP computations in ultra-low-power programmable architecture. Particularly, Design choices 1, 2, and 4 implement optimization techniques in an incremental pattern, such that each design shows better performance and energy efficiency over the previous one. Following are the overview of the design choices:

1. Design 1 (section 4.1.1) is an attempt to implement SoA IEEE 754-2008 standard compliant FPU in the CGRA. This addition enabled the support for multi-cycle operations in the CGRA and the associated toolchain.
2. Design 2 (section 4.1.2) is the first ever [1] implementation of *transprecision* FP based FPU featuring two custom FP datatype i.e., 16-bit *binary16alt* and 8-bit *binary8* in an ultra-low-power CGRA. This addition enabled the support for custom datatype in the CGRA and associated toolchain.

[1] to the best of our knowledge

3. Design 3 (section 4.1.3) is an attempt to integrate all the FP datatype operators into one hardware block. The design space exploration indicated that the resultant hardware block would not be high-performing or energy-efficient w.r.t. Design 1 and 2 due to the area overheads and lack of fine-grained clock or datapath gating schemes. This design choice is presented to inform the designers working on the presented CGRA with a similar configuration as Design 3 to further implement the optimization techniques required to obtain better performance and higher energy efficiency from the resulting CGRA design.

4. Design 4 (section 4.1.4) is an iteration of Design 2 that only features *binary16alt* operators along with 32-bit integer operators. The CGRA is also integrated into a parallel computing cluster to present a heterogeneous cluster that exploits both the CGRA sub-system and multi-core sub-system of the host CPU to pull off the highest performances from the heterogeneous cluster. This integration is explained in much detail in chapter 6.

5. Design 5 (section 4.1.5) is an attempt to support an 8-bit integer for accelerating Deep Neural Network algorithms. Five 8-bit integer operators along with few optimization techniques have been presented. However, SoA architectures, namely TPU [47], Tensorflow [102], and NVDLA [103] are far superior architectures w.r.t. the presented CGRA, and thus no further development of Design 5 is made during the course of the research work presented in this thesis.

Figure 4.1 represents the organization of the CGRA sub-system consisting of a Processing Element (PE) array interconnected via a mesh torus network, a Direct Memory Access Controller (DMAC), and Context Memory. CGRA is loosely coupled with a host CPU. Data is shared between two sub-systems through an L1 scratchpad memory called Tightly Coupled Data Memory (TCDM) and is connected to the processing units through an interconnected network.

## Processing Element

Figure 4.2 shows the organization of a typical PE in the CGRA. To exploit inexpensive data movement in PE array and also to avoid frequent memory operations due to temporal data, each PE features a Constant Register File (CRF) to reduce the size of the instructions, a Regular Register File (RRF) to store the temporary values, and an OutPut Register (OPR). Instructions that are local to a PE are stored in the Instruction Register File (IRF).

To perform 32-bit integer operations, each PE includes an Arithmetic Logic Unit (ALU). For FP operations, each PE includes an FP module (different design optimizations implement different versions of FP modules). To synchronize between multi-cycle operations, PE also features an Instruction Synchronizer (IS). PEs that feature Load-Store Unit (LSU), a Flexible-Address Generation Unit (Flexible-AGU), is glued to every LSU for hardware-based address generation. Each PE can directly share data with their

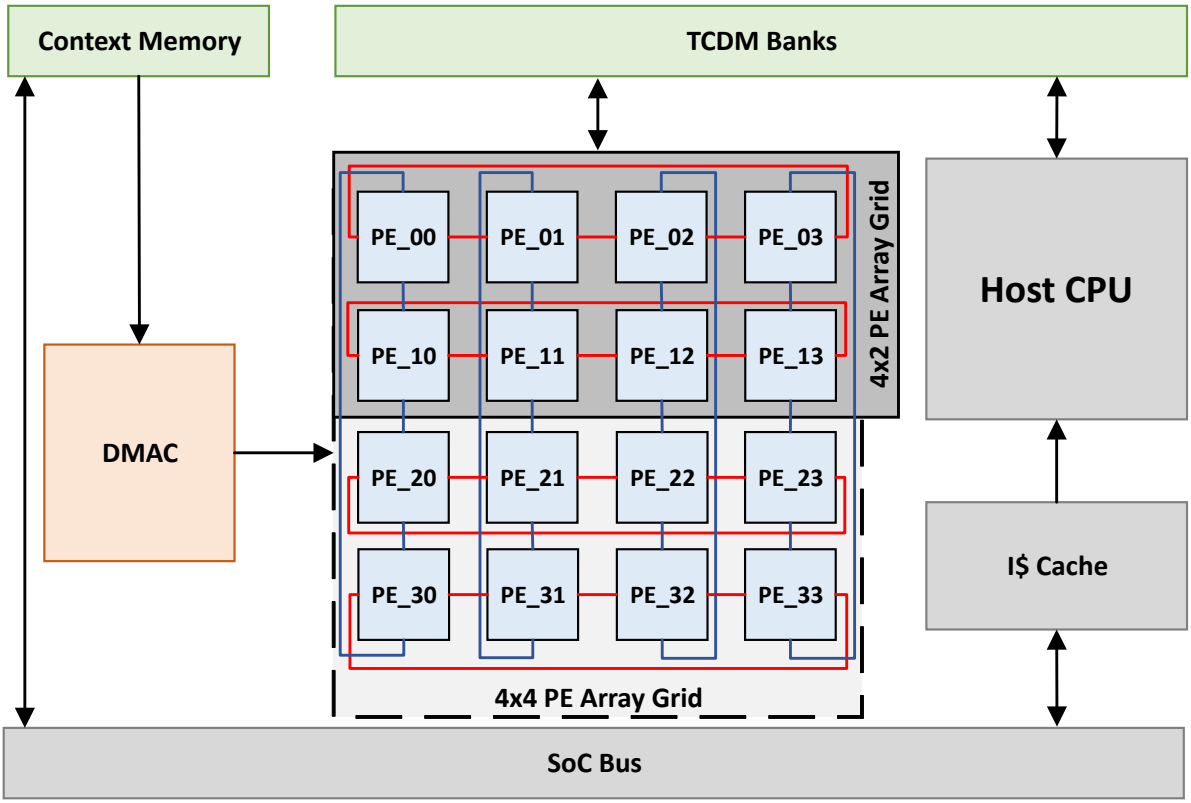



Figure 4.1: CGRA Integrated System

orthogonally adjacent PEs. Multiple optimization techniques have been implemented in PEs which are discussed later in this section.

### Flexible-AGU

In order to decouple the software based address generation in CGRA, Flexible-AGU is introduced for hardware based address calculation. This module is compatible with a Global Index representation i.e.,

$$Variable\ [(i + \mathrm{A}) * (j + \mathrm{B})]\ [(k + \mathrm{C}) * (l + \mathrm{D})]$$

where, $i,j,k,l$ are the loop variables (LV) and A, B, C, D are the constant/variables. This module can handle representing any 1D or 2D array variable in the target domain applications [2] e.g.
(1) Input[($i$+0)*(1+0)][(1+0)*(1+0)] is the representation for Input[i][1] and
(2) Input[($i$+0)*(1+0)][($j$+0)*(1+0)] is the representation for Input[i][j].

The organization of Flexible-AGU is shown in Figure 4.3, data required for address calculation is embedded in the LOAD/STORE instruction. Particularly, the essential information required for calculating the address of an element of an array are (1) the Base Address of the array and (2) index values. The ISA of CGRA is 21-bit wide and

[2]Flexible-AGU is a hierarchical design and can be updated to work for 3D array variables.

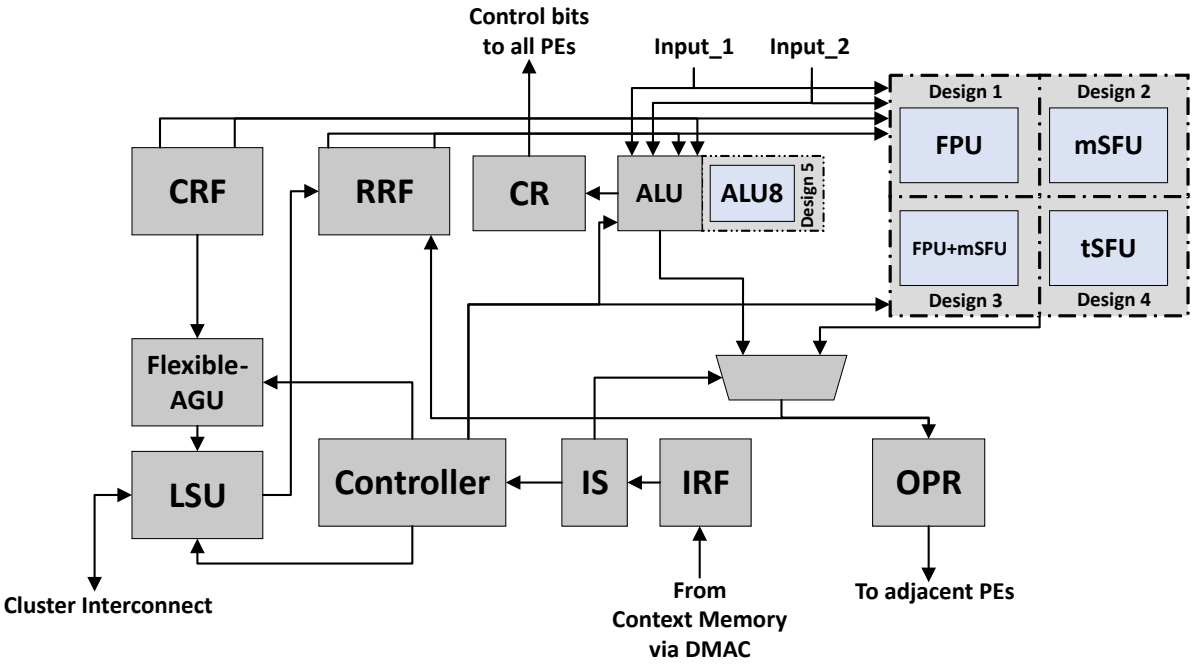



Figure 4.2: Processing Element

can not be included directly in the instruction. This information is stored in the CRF. The corresponding addresses are embedded in the LOAD/STORE instruction (as shown in Figure 4.3). Index information is encoded by the assembler, which is as follows:

1. Loop variables are represented in CRF as

| MSB | 10-bit | 8-bit | LSB |
|---|---|---|---|
| 1/0 | START | STEP | 1/0 |

   where MSB determines if START (10-bit) value is a loop variable (MSB=1) or a constant (MSB=0) and LSB determines if the STEP (8-bit) value should be subtracted (LSB=1) or added (LSB=0) from the START value in the next loop iteration.

2. Index Pair (i.e., ($i$+A) or ($j$+B) or ($k$+C) or ($l$+D)) lines are represented in CRF as

| MSB | 9-bit | 10-bit |
|---|---|---|
| 1/0 | LV address/Constant | Constant |

   where, MSB determines if *i,j,k or l* are loop variables (MSB=1) or a constant value (MSB=0). In the case of MSB=1, then the 5-bit address of the loop variable is

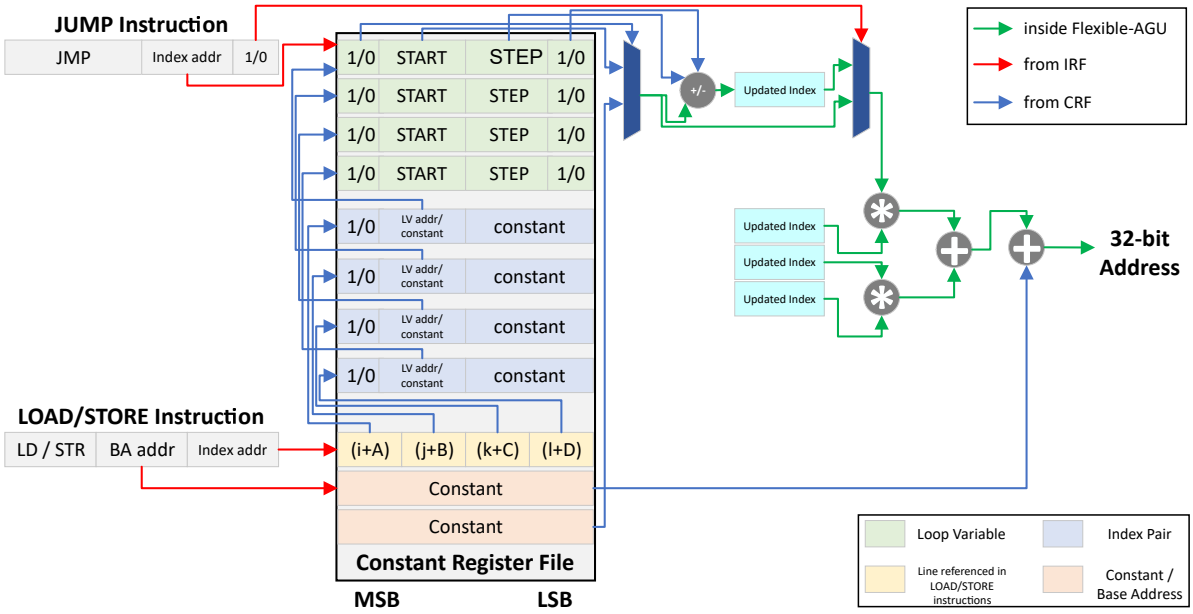



Figure 4.3: Flexible-AGU

stored at [15:11] of the Index Pair line.

3. CRF line referenced in the LOAD/STORE instruction contains 5-bit addresses of the Index Pair lines is represented as

| 5-bit | 5-bit | 5-bit | 5-bit |
|---|---|---|---|
| ($i$+A) address | ($j$+B) address | ($k$+C) address | ($l$+D)) address |

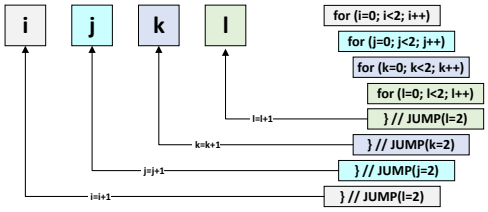
The values obtained from index-pairs are processed, and finally, Base Address is added to produce a 32-bit address. *JUMP* instructions are used to determine which loop-variable is either updated or reset to the initial value (i.e., START values).

Let's take a C-code example (Code 4.1) to understand the address generation in the Flexible-AGU.

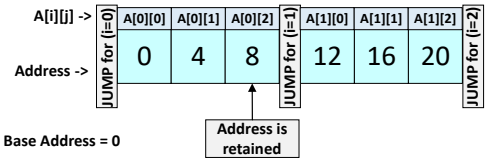
```
int i, j;
float A[2][3], B[2][3], C[2][3];
for (i = 0; i < 2; i++){
    for (j = 0; j < 3; j++){
        C[i][j] = A[i][j] * B[i][j];
    }
}
```

Code 4.1: Sample C code snippet

To keep it simple, let's focus on the address generation for A[i][j] with a Base Address of A=0 (See Figure 4.4b). Particularly, the address is calculated using the formula (for




(a) Loop Variables and associated JUMP instructions




(b) Addresses for 2D array using 2 loop variables

Figure 4.4: Address calculation in Flexible-AGU

a 2D array with row-major ordering).

$$\text{Address} = [\ \{\sum_{n=1}^{AD}(I * C + J)\} * \mathit{Address\ Space}\ ] + BA$$

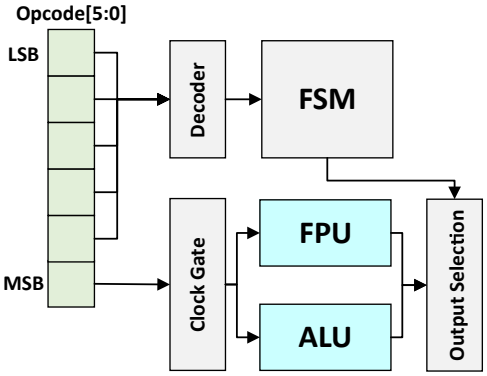
where,

$C =$ Number of Columns in array
$I =$ Row Number
$J =$ Column Number
$BA =$ Base Address
$AD =$ Array Dimension
$\mathit{Address\ Space} =$ Memory chip is divided into equal parts, called cell and *Address Space* is the data space in the cell.
TCDM is word addressable, so,
Address space = Word size (i.e., 4Bytes)

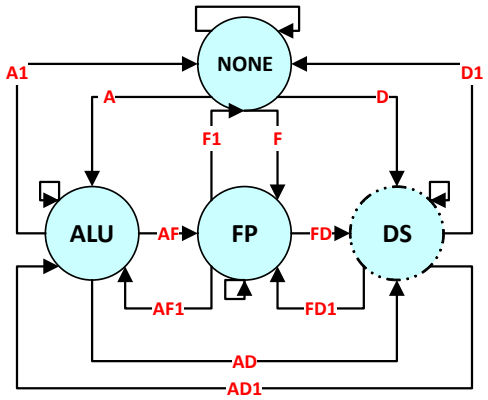
Derived equation from the above formula for calculating address of A[i][j] in *Flexible*-AGU is:

$$\begin{aligned}\text{Address} &= \text{Address of A[i][j] before JUMP for previous } i \text{ value} \\ &+ ([(i + 0) * (1 + 0)] + [(j + 0) * (1 + 0)])*4 \\ &+ \text{BA}\end{aligned}$$

An optimization technique to retain the value of A[i][j] before $JUMP$ for $i$ always produced correct address generation for variables in the target domain applications. Implementing this technique also kept the circuit of Flexible-AGU simple, which eventually improved the energy efficiency of CGRA. TCDM is word (i.e., 32-bit) addressable (See section 3.6), variables multiplied by 4. Address for A[0][0] is 0 as all values in the formula are 0, similarly, addresses for A[0][2] and A[1][0] are 8 and 12 respectively (See Figure 4.4b). This example can also be extended for address generation for variable using 4 loop variables e.g. A[(i+n1)*(j+n2)][(k+n3)(l+n4)], where i,j,k, and l are loop variables and n1, n2,n3, and n4 are the constants or variables (See Figure 4.4a).




(a) Organization of Instruction Synchronizer




(b) Instruction Synchronizer FSM

Figure 4.5: Instruction Synchronizer

### Instruction Synchronizer

To synchronize the inputs and outputs of multi-cycle operations from ALU and FP modules, an Instruction Synchronizer (IS) is introduced in PE (See Figure 4.5). Figure 4.5a shows the organization of IS, using last 5 bits of opcode[5:0] (i.e., excluding MSB) the FSM featured in IS can decode if it is an ALU operation (1 clock-cycle), FP operation (1/2 clock-cycles), or DS operation (5/8 clock-cycles). FSM also generates a signal used to select between the ALU or FP module or DS unit output. To optimize the design, there are two versions of IS, which are (1) FSM with ALU and FP states (implemented in the PEs featuring ALU and FPU only), and (2) FSM with ALU, FP, and DS states (implemented in the PEs featuring ALU, FPU, and DS Unit). Later in this chapter, a technique to clock-gate ALU and FP Units using the MSB (1-bit) of opcode and involving IS is presented. (See Figure 4.8).

Depending on the decoded opcode, FSM determines its state for synchronizing the operations in a PE. $NONE$ state represents no operation is executing in a PE, $ALU$ state represents ALU operations, $FP$ state represents FP module operations, and $DS$ state represents Divide-Square-root Unit operations. In Figure 4.5b, A/F/D and A1/F1/D1 represent the conditions for transitions between $ALU/FP/DS$ state to and from $NONE$ state, respectively. Similarly, AD/AF/FD and AD1/AF1/AD1 represent the conditions for transitions between $ALU$, $FP$, and $DS$ states, respectively. When the FSM is in $NONE$ state, PE is clock gated particularly, if the current state of FSM is in either $ALU$ or $FP$ or $DS$ state, then clock gate the other modules which are not represented by the current state in FSM.

## 4.1.1 Design 1: IEEE 754-2008 Standard compliant 4x2 PE Array

A 4x2 PE array is proposed in the first design optimization, and implicit support for IEEE 754-2008 FP computing is embedded in the CGRA. Figure 4.1 shows the CGRA

| Float Operator | Latency (cycles) | | | |
|---|---|---|---|---|
| | FPU (32-bit) | mSFU (16-bit) | mSFU (8-bit) | tSFU (16-bit) |
| Absolute | 1 | 1 | - | 1 |
| Less-Than | 1 | 1 | - | 1 |
| Multiply | 2 | 2 | 2 | 1 |
| Add | 2 | 2 | 2 | 1 |
| Subtract | 2 | 2 | 2 | 1 |
| Divide | 8 | 5 | - | 5 |
| Square-root | 8 | 5 | - | 5 |
| Fused-Multiply-Accumulate | - | - | - | 2 |

Table 4.1: Latency (cycles) of float operators

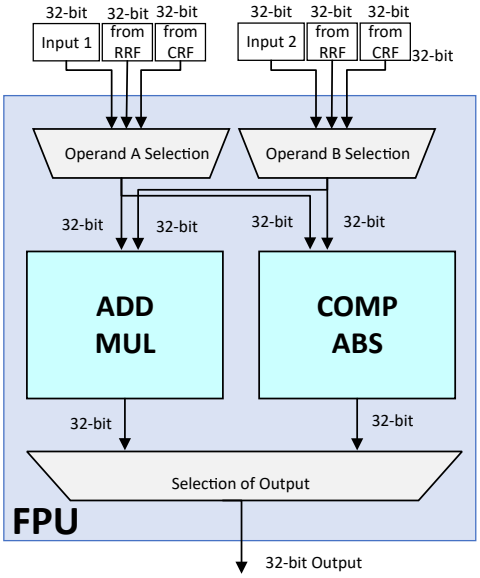



Figure 4.6: FPU and its underlying hierarchical blocks

integrated system, and Figure 4.2 shows the organization of a PE with IEEE 754-2008 compliant FP Unit (FPU), respectively. PEs are connected through a mesh torus network for sharing data with orthogonally adjacent PEs and a bus network to broadcast the context data. Each PE features an FPU, a Load Store Unit, and a Flexible-Address Generation Unit (Flexible-AGU).

### FPU

Figure 4.6 shows the hierarchical blocks present in the FPU. FPU consists of five IEEE 754-2008 compliant FP operators, i.e., float-Absolute, float-Less-Than, float-Multiply, float-Add, and float-Subtract. Division and Square-root operators are very costly w.r.t. other FP operators, so a shared Divide-Square-root (DS) Unit is included in the PE array, i.e., in PE_00. Table 4.1 shows the latency of featured FP operators in FPU.

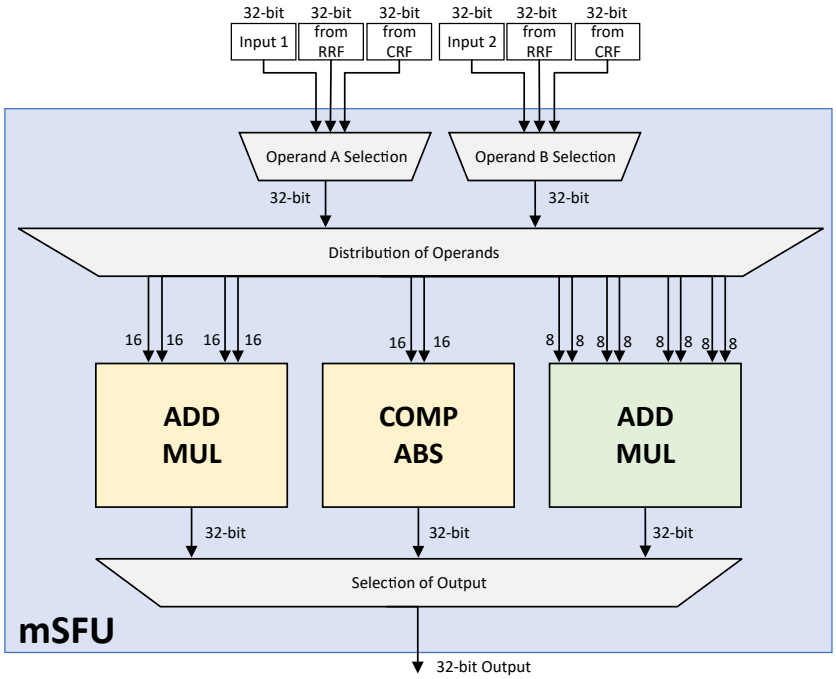



Figure 4.7: mSFU and its underlying hierarchical blocks

Estimation of latency of each FP operator is done based on IEEE 754-2008 compliant FP operators featured in shared-FPU-cluster on PULP-Cluster.

### 4.1.2 Design 2: Transprecision FP compliant 4x2 PE Array

This design optimization introduced *transprecision* computing in the CGRA. A 4x2 size is used for the second design optimization, and instead of FPU, a mini-SFU (mSFU) is introduced in the CGRA (Figure 4.1). As *transprecision* FP operators use 16-bit and 8-bit operands, SIMD is also introduced in the design to optimize resources on the CGRA. Due to the inclusion of more operators, ISA is extended by 1-bit (i.e., 21-bit ISA), and this design optimization required updating the IRF size from 20*64 to 21*64 due to ISA extension and CRF size from 32*16 to 20*32 due to introduction of Flexible-AGU (for storing more constants). Similar FP operators as of an FPU are featured in the mSFU. A simple clock-gating scheme is implemented between ALU and mSFU to improve the energy efficiency of CGRA further.

#### mini-SFU

Figure 4.7 shows the hierarchical blocks present in the mini-SFU (or mSFU). mSFU includes five FP operators, i.e., float-Absolute, float-Less-Than, float-Multiply, float-Add, and float-Subtract (shown in Table 4.1) for *transprecision* computing using variable FP datatype. With reduced datapath, it was feasible to introduce three *binary16alt* compatible DS Units in the first three PEs (i.e., PE_00, PE_01, and PE_02). Commonly

used *binary16alt* based FP operators (i.e., float-Multiply, float-Add, and float-Subtract) execute in 2-lane SIMD fashion (See Figure 4.9). Two *binary16alt* based FP operators float-Absolute and float-Less-Than, often occur in a control statement and can not execute in parallel, are single-lane FP operators (i.e., do not use SIMD). Costly and rarely occurred *binary16alt* based float-Divide and float-Square-root operators do not execute in SIMD. Divide and Square-root operators use an iterative non-restoring divider, and the required clock cycles are determined by the size of mantissa (i.e., precision bits). With reduced mantissa size (i.e., 7-bit) these two FP operators have less latency (i.e., 5-cycles) w.r.t. IEEE 754-2008 counterparts (i.e., 8-cycles).

*binary8* FP datatype features only 2 precision bits and can only be used to perform three fundamental FP operations (i.e., float-Multiply, float-Add, and float-Subtract). These three FP operators execute in a 4-lane SIMD fashion.

### 4.1.3 Design 3: Mixed FP based 4x2 PE Array

Design space exploration to implement a version of CGRA featuring a configurable FP Unit capable of executing 32-bit, 16-bit, and 8-bit FP format operations. The third design optimization is the aggregation of Design 1 and Design 2. The FP module in this design optimization includes both an FPU and an mSFU. This design choice increased the total area of CGRA and reduced the energy efficiency of CGRA due to the lack of fine-grained optimization in PEs.

### 4.1.4 Design 4: Transprecision FP compliant 4x4 PE Array

This configuration is used for designing a *transprecision* FP compliant heterogeneous cluster using CGRA and 8-cores based on a 4-stage in-order RISC-V CPUs, namely RI5CY [48]. Due to low PE Utilization in CGRA w.r.t. RI5CY cores, it was necessary to optimize the PEs further and integrate CGRA into PULP-Cluster. Following solutions were implemented in the CGRA sub-system to achieve higher performances (while executing a set of real-world DSP applications) from the proposed heterogeneous cluster using different configurations:

- 4x4 PE array featuring 8 LSUs is used to
  - keep the total number of LSU in CGRA equal to the total number of LSU in the RI5CY sub-system.
  - exploit cheap $MOVE$ operations due to simple mesh torus interconnect network in CGRA.
- Only 1 DS Unit was included in PE_00 because float-Divide and float-Square-root operations are rare and costly.

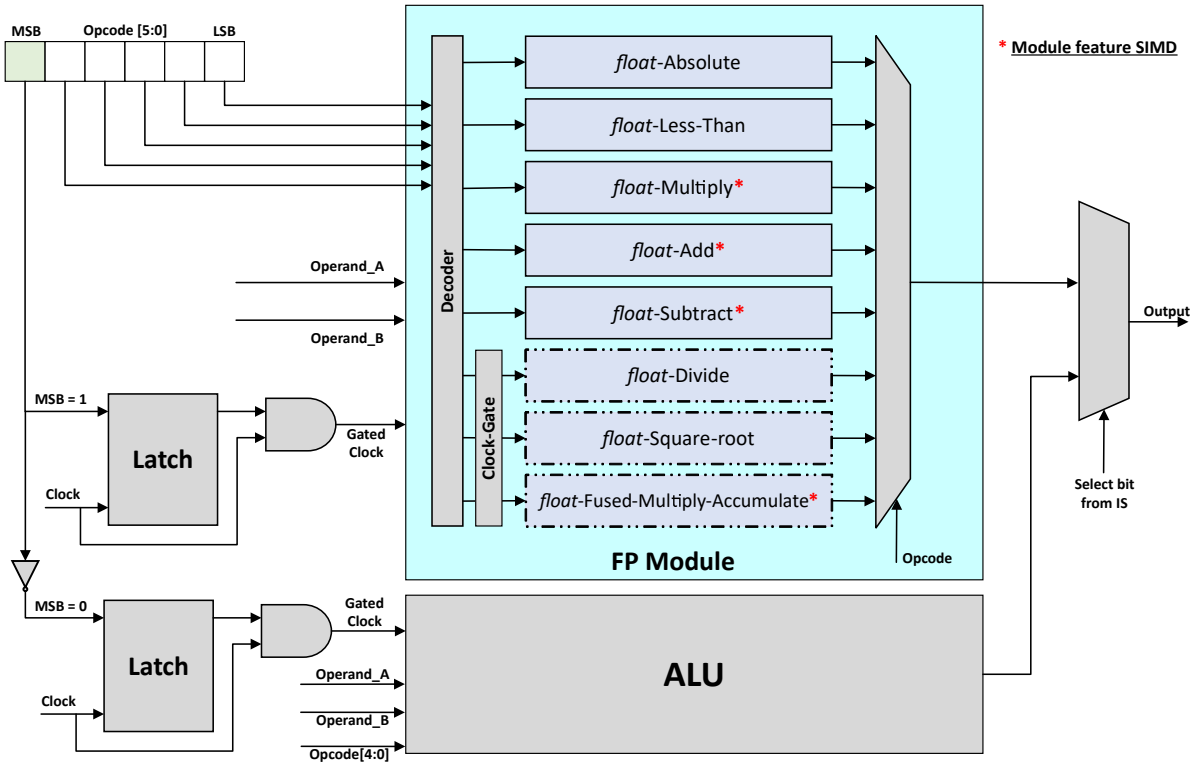



Figure 4.8: Clock-Gating Scheme between FP module and ALU

- 3 types of PE are featured to exclude the modules which are not required. These are
  - PE with an LSU and Flexible-AGU to generate corresponding addresses for memory operations (See Figure 4.2). Interleaving PE rows are used to place these PEs for optimal movement of loaded data from memory.
  - PE with ALU, tSFU, and DS Unit (i.e., PE_00 only)
  - PE with ALU and tSFU (i.e., all PEs but PE_00)
- CGRA can achieve a high PE utilization while executing a kernel but often fails to maintain such high PE utilization, so a hierarchical clock-gating scheme is implemented to improve the energy efficiency of CGRA further. These are
  - clock-gate PE, if End-of-Execution is reached.
  - clock-gate all modules if stall(s) occurred due to memory operation(s), i.e., LOAD/STORE or NOP(s).
  - clock-gate ALU, if FP operators are active and vice-versa (See Figure 4.8).
- To further exploit the cheap $MOVE$ operations in PE array, when a data is loaded

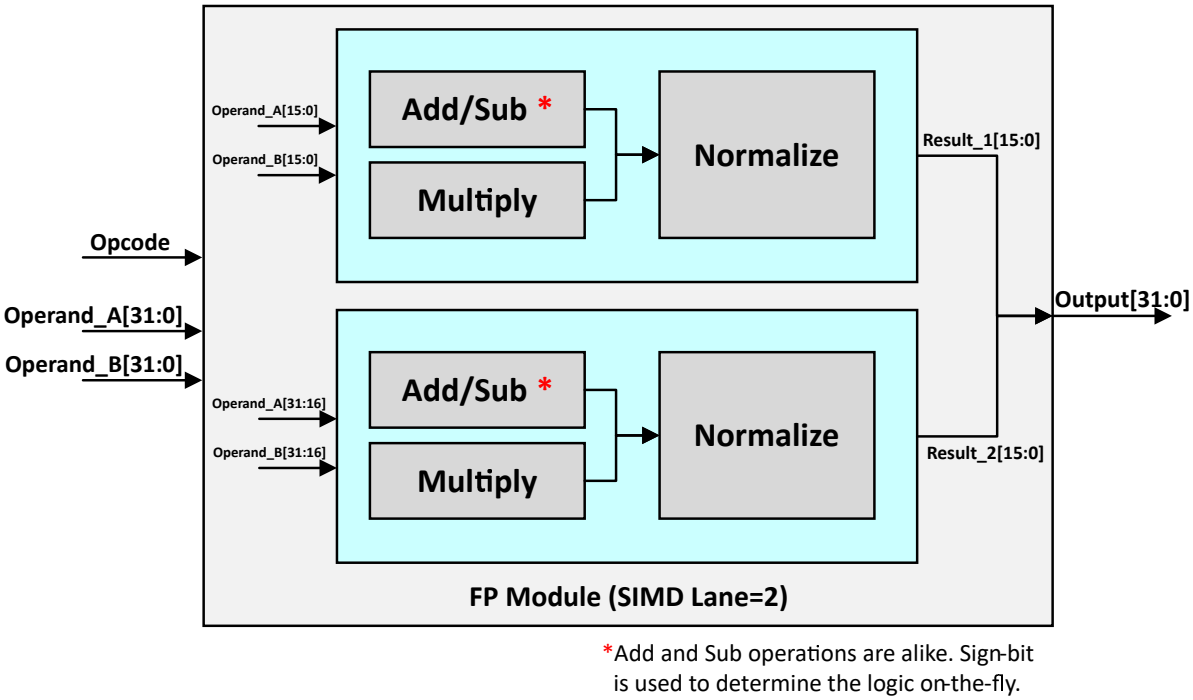


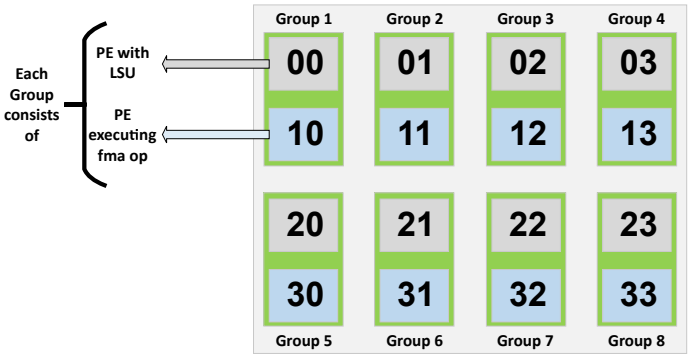

Figure 4.9: SIMD execution in FP module

from memory into a PE, broadcast the loaded data to its adjacent PEs (See Figure 4.10). A single such broadcast can save up to the execution of 4 parallel *MOVE* instructions to move loaded data into adjacent PEs, which resulted in a significant improvement of the latency performance of CGRA.

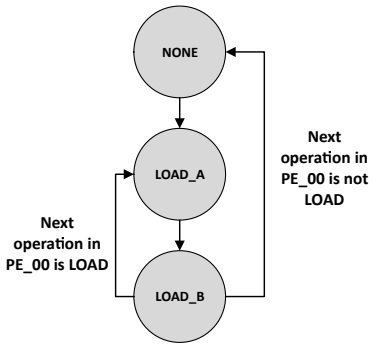
### tiny-SFU

*binary16alt* is widely adopted in mainstream architectures due to its high dynamic range (i.e., similar to IEEE 754-2008 *binary32*) with a smaller footprint, i.e., reduced precision bits (7-bit) [104]. It is apparent to exclude *binary8* in the next iteration of *transprecision* computing-based CGRA due to lack of real-world applications for *binary8* datatype. This design optimization features FP operators using *binary16alt* only. Implementation of only 16-bit operators further reduced the area of *transprecision* FP-based module (i.e., tiny-SFU (tSFU)) in the CGRA.

Considering that CGRA often fails to maintain a high PE utilization while executing a kernel, to increase the energy efficiency of CGRA, a fine-tuned clock-gating scheme is implemented between tSFU and ALU. A simple technique of comparing MSB of opcode to determine execution between an FP operation or an ALU operation is employed. This also aided the implementation of clock-gating using only 1-bit (See Figure 4.8) to reduce the unnecessary switching activities in a PE. In Figure 4.8, a 6-bit opcode is extracted from the 21-bit instruction. The MSB determines if the operation to be executed is an FP operation (MSB=1; clock-gate ALU) or an ALU operation (MSB=0;




(a) PE grouping for FP-FMA operation




(b) FSM for FP-FMA operation

Figure 4.10: Implementation of FP-FMA in CGRA

| Timestamp | Operation | FSM State |
|---|---|---|
| #0 in PE_00 | *LOAD* data A | NONE |
| #1 in PE_01 | • Fetch A from PE_00<br>• Perform accumulation<br>• Jump to **LOAD_B** State | LOAD_A |
| #1 in PE_00 | *LOAD* data B | |
| #2 in PE_01 | • Fetch B from PE_00<br>• Perform multiplication<br>• If there is next *LOAD* in PE_00, then jump to **LOAD_B** State else, jump to **NONE** | LOAD_B |

Table 4.2: State description of FSM for FP-FMA operation

clock-gate tSFU). The remaining 5-bits are then sent to their respective destination and decoded to determine the specific operation. If the FP operation is a SIMD (lane=2; 2x 16-bit) operation, then the final output is concatenated to align the incoming two 16-bit results from the FP normalize module to produce 32-bit data (See Figure 4.9).

Furthermore, a 3-state FSM-based FP-Fused-Multiply-Accumulate (FP-FMA) module is also implemented to optimize kernels that extensively employ FP multiplication and accumulation (See Figure 4.10 and Table 4.2). First, a group is formed using (1) PE with LSU and (2) PE without LSU as shown in Figure 4.10a and 8 such groups can be formed in the current 4x4 PE array. To understand the working of this FSM (See Figure 4.10b) based FP-FMA module, let us focus on the PE group formed with PE_00 and PE_10 (See Table 4.2). When an FP-FMA opcode is encountered in PE_10 (i.e., timestamp #0), at the same timestamp, a *LOAD* operation is also scheduled in PE_00. In this timestamp, the FSM is in NONE state (FP-FMA module is clock-gated while FSM is in NONE state) and set to jump to next state, i.e., LOAD_A. In the next timestamp (i.e., timestamp #1), PE_00 has fetched the required data from memory and has broadcast it to the adjacent neighboring PEs. Due to the cheap *MOVE* operation in CGRA, 8 such parallel FP-FSM can be executed without significant energy-wise overhead. Now, PE_10 will choose the incoming operand from PE_00 only as operand information is provided by

| MSB | | | | | LSB | |
|---|---|---|---|---|---|---|
| trigger | masked-bit | | bits used for decoding | | | |
| 1 | 0 | 0 | 0 | 0 | 0 | add |
| 1 | 0 | 0 | 0 | 0 | 1 | sub |
| 1 | 0 | 0 | 0 | 1 | 0 | mac |
| 1 | 0 | 0 | 0 | 1 | 1 | sumdotp |
| 1 | 0 | 0 | 1 | 0 | 0 | clip8 |

Table 4.3: ALU8 6-bit Opcodes. MSB is used to trigger ALU8, next 2-bit are masked, and remaining last 4-bit are used for decoding ALU8 operation.

fetched instruction in PE_10. Fetched data is then accumulated with the data present in a special register allocated for temporarily storing the accumulated values from FP-FMA operations. FSM is set to jump to the next state, i.e., LOAD_B. In the same timestamp (i.e., timestamp #1), PE_00 will fetch the second operand required by PE_10 in the next clock cycle. In the next timestamp (i.e., timestamp #2), if there is a load in PE_00, a trigger signal is also issued, which is used by FSM in PE_10 to decide the next state from the LOAD_B state. In timestamp #2, PE_10 will fetch the data from PE_00, and FSM is in LOAD_B state already. Data loaded in previous clock-cycle (i.e., operand A) and data fetched in current clock-cycle (i.e., operand B) are multiplied, and depending on the value of trigger signal issued by PE_00, FSM will either jump to LOAD_A state (if there is *LOAD* operation in PE_00 in timestamp #3) or jump to NONE state (in this case, result from a multiplication in PE_10 is added with the data in accumulator register and stored in a register in RRF of PE_10, this destination register is provided in the instruction fetched in timestamp #2).

### 4.1.5 Design 5: 4x2 PE Array featuring 8-bit integer operators

This design optimization includes 32-bit integer (i.e., ALU) and 8-bit integer (i.e., ALU8) datatype operators with 4x2 PE array (See Figure 4.2). Particularly, ALU8 features 8-bit integer (SIMD lane=4) based operators (See Table 4.3). These operators are implemented to accelerate the Deep Neural Network (DNN) algorithms because DNNs use quantization (i.e., a technique to reduce memory footprints) to reduce inputs and weights, and 8-bit formats are well suited for such techniques [105, 106, 107, 108].

Five 8-bit integer based operators are implemented in ALU8, these are:

1. **add**: performs SIMD style 4x8-bit addition
2. **sub**: performs SIMD style 4x8-bit subtraction
3. **mac**: performs SIMD style 4x8-bit multiply-accumulate
4. **sumdotp**: performs dot-product using two 4x8-bit inputs

5. **clip8**: called just before STORE operation to clip (i.e., rounding and shifting) the accumulated 32-bit value to 8-bit value

Decoding of operators (opcode width= 6-bit) in ALU8 is performed using 3-bit only, and the remaining 2-bit are masked (these bits are always zeros), and MSB is used to perform clock-gating between ALU and ALU8 (See Table 4.3). Clock-gating between ALU and ALU8 is implemented using a similar technique presented in Design 4 (i.e., clock-gating between FP module and ALU) (See Figure 4.8).

ALU8 features a dedicated register file to temporarily store the immediate values generated while executing a kernel. The 5x32-bit register file is sectioned into two parts to store the values from mac and sumdotp operators.

- 5x32-bit register file:
  - 4x32-bit registers are dedicated to temporarily store 4 accumulated values from 4 parallel 8-bit integer mac operations, and
  - 1x32-bit register is dedicated to temporarily store the dot-product result from sumdotp operator.

Code 4.2 is a sample code snippet (executed on RI5CY), including 2 loops.

```
// ** int8 operation in RI5CY **

// ** PULP built-in macro to perform dot-product on a.b **
// ** and result is stored in c **
#define SumDotp (a, b, c)    __builtin_pulp_sdotp4(a, b, c)

// ** PULP build-in macro to clip 32-bit value to 8-bit value **
// ** range is manually defined **
#define CLIP8(x)     __builtin_pulp_clip(x, -128, 127)

int8_t * pInBuffer;
int8_t * pWeight;
uint16_t channels, columns, out_shift;
int8_t * pOut;
int8_t *pOut2 = pOut + channels;

// ** PULP built-in macro to represent a vector of 4 chars **
// ** typedef signed char v4s __attribute__ ((vector_size (4))); **
v4s vecA;
v4s vecB;

for (int i = 0; i < channels >> 2; i++){
    int8_t *pB  = pInBuffer;
    int8_t *pB2 = (pB + columns);
    int8_t *pA  = pWeight;
    int sum, sum2;

    for (int j = 0; j < columns >> 2; j++){
        vecA  = *( (v4s*) pA  );
```

```
        vecB  = *( (v4s*) pB  );
        vecB2 = *( (v4s*) pB2 );

        sum  = SumDotP (vecA, vecB, sum);
        sum2 = SumDotP (vecA, vecB2, sum);

        pA  += 4;
        pB  += 4;
        pB2 += 4;
    }

    *pOut = (int8_t) CLIP8(sum >> out_shift);
    pOut++;
    *pOut2 = (int8_t) CLIP8(sum2 >> out_shift);
    pOut2++;
}
pOut += channels;
```

Code 4.2: C code snippet for matrix multiplication with PULP built-in macros

- Highlights of code snippet (Code 4.2) for RI5CY
  - *sum* and *sum2* stores the accumulated values over 2 loops
  - *SumDotP* and *CLIP8* are 3-input operand operators
  - *CLIP8* is called for rounding the final values
  - If same operators without dedicated register file in ALU8 are implemented in CGRA, then the inner loop requests *channles*×*columns* LOAD, STORE, and MOVE operations for sum and sum2 (such unnecessary operations are avoided by introducing dedicated 5x32-bit register file in ALU8)

```
// ** int8 operation in CGRA **
int i, j, channels, columns, out_shift;
int = pInBuffer[columns*channels];
int = pWeight[columns*channels];
int = pOut[columns*channels];

for (i = 0; i < channels >> 2; i++){
    int sum, sum2;

    for (j = 0; j < columns >> 2; j++){
        sum  = sumdotp (pInBuffer[i], pWeight[i], sum);
        sum2 = sumdotp (pInBuffer[i], pWeight[i+j], sum);
    }
    pOut[i]   = clip8(sum >> out_shift);
    pOut2[i+1] = clip8(sum2 >> out_shift);
}
```

Code 4.3: Modified C code snippet for matrix multiplication for CGRA

- Highlights of transformed code (Code 4.3) snippet for CGRA

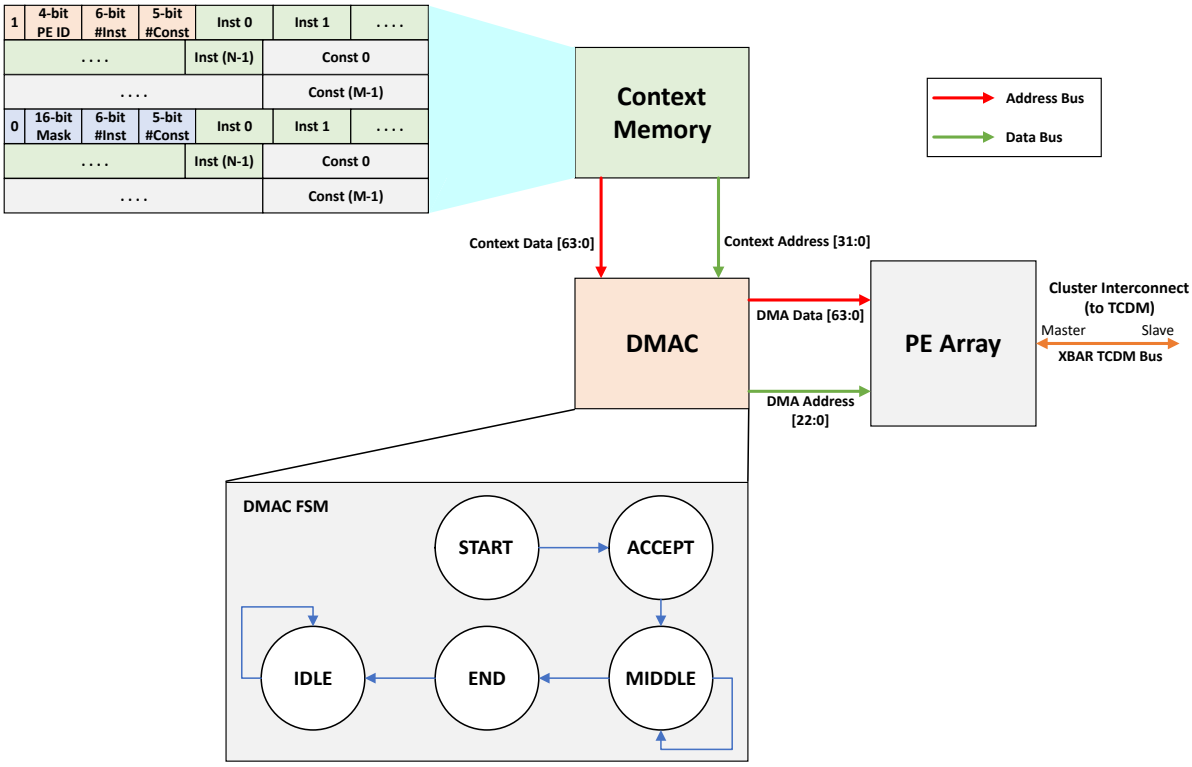



Figure 4.11: Configuration Network for context loading

- *sum* and *sum2* stores the accumulated values over 2 loops
- *sumdotp* and *clip8* are 2-input operand operators
- Due to implementation of dedicated register file in ALU8, the inner loop requests **0** LOAD, STORE, and MOVE operations for sum and sum2
- When *clip8* operation is executed, then 32-bit *sum* and *sum2* values stored in dedicated register file are rounded and stored back in memory (pOut[i] and Out[i+1])

Design optimization 5 presented a good exploration for accelerating DNNs on CGRA. However, SoA architectures (i.e., TPU [47], Tensorflow [102], and NVDLA [103]) are far superior architectures w.r.t. CGRA and thus restrained from further development of CGRA for accelerating DNNs.

## 4.2 Computation Model

The associated compiler compiles an application, and required assembly code is obtained (See chapter 5), then the assembler processes obtained assembly code using the

| State | Information |
|---|---|
| START | Sets flag for accepting inputs from Context Memory and jump to *ACCEPT* state. |
| ACCEPT | Detects the addressing mode and jumps to *MIDDLE* state. |
| MIDDLE | (1) Fetches a 64-bit context data from Context Memory, (2) detects number of instructions and constants for that particular PE, (3) directs the instructions into the IRF and the constants to CRF of that particular PE, and (4) jumps to *MIDDLE* state to repeat for next PE. (5) When done, jumps to *END* state. |
| END | Stops accepting inputs from Context Memory and jumps to *IDLE* state. |
| IDLE | DMAC enters sleep mode. |

Table 4.4: Description of FSM states in DMAC

ISA (See Table 4.5 and Table 4.7) of CGRA to generate the required instruction part of a context for each PE (i.e., program to be stored into the IRF). Assembler also employs a technique to eliminate the repeating constants in CRF for each PE by recursively checking if the constant to be allocated in the CRF of a PE exists already or not (See section 5.2). Finally, the context data (comprises of instructions and constants) is generated for each PE which is later pre-loaded (and prior to execution) in the Context Memory of CGRA. Corresponding instructions and constants are then loaded into their respective PEs using DMAC (See Table 4.4). If the context data size is larger to be fit into the PE array, then context data is split into multiple parts and executed sequentially.

## Load Context

Figure 4.11 shows the configuration network for loading context data in PEs, and Table 4.4 shows the description of each state in DMAC FSM. Once the context data has been loaded into the Context Memory, DMAC starts receiving the fetched context data and detects the number of instructions and constants for each PE using the configuration lines (See Figure 4.12). Each configuration line contains information, which are (1) PE ID, (2) instruction count, and (3) constant count, and once this line is decoded, DMAC redirects the bits to their respective PEs.

A full-sized series of instructions for a PE that can fit in the IRF is 64x21-bit, and similarly, 32x20-bit of constants can fit into a CRF of a PE. So, the configuration line dedicates 6-bit to represents the instruction counts (i.e., 6-bit are required to represent numbers between 0 to 63 in binary representation) and 5-bit to represent a constant count (i.e., 5-bit are required to represent numbers between 0 to 31 in binary representation). There can be either 8 or 16 PEs in a PE array, so 4-bit is dedicated to

MSB LSB

| 5-bit | 1-bit | 5-bit | 1-bit | 3-bit | 6-bit |
|---|---|---|---|---|---|
| Input_2 Addr | Type (1/0) | Input_1 Addr | Type (1/0) | DRA | Opcode |
| Index Addr | Type (1) | BA Addr | Type (1) | DRA | LOAD |
| Index Addr | Type (1) | BA Addr | Type (1) | DRA | STORE |

| 1-bit | 2-bit | 5-bit | 1-bit | 6-bit | 6-bit |
|---|---|---|---|---|---|
| (1=BB)/(0=LV) | 00 | LV Addr | Type (1) | BB Addr | JUMP |

| 3-bit | 6-bit | 6-bit | 6-bit |
|---|---|---|---|
| 000 | False Path Addr | True Path Addr | CJUMP |

| 10-bit | 5-bit | 6-bit |
|---|---|---|
| 00 0000 0000 | #Consecutive NOPs | NOP |

DRA : Destination Register Address
BB : Basic Block
LV : Loop Variable
Type=1 : Source is in CRF
Type=0 : Source is either in RRF or in neighbouring PE

Table 4.5: 21-bit ISA Table

representing the PE ID. Figure 4.13 shows the format of the data and address bus in the configuration network and Table 4.6 represents the structure of Context memory segments. First bit is dedicated to determine the addressing modes, that are

- *Broadcast Addressing Mode*: This addressing mode is used to broadcast a set of instructions and constants to the PEs requiring identical instructions and constants in the CGRA.

- *Normal Addressing Mode*: This addressing mode is used to load non-identical instructions and constants to the PEs in the CGRA.

| **#bits** | **Information** |
|---|---|
| 1-bit | Addressing Mode |
| 4-bit or 16-bit | Normal Address or Mask |
| 6-bit | Instruction count for current PE |
| 5-bit | Constants count for current PE |
| N*21-bit | Total Instructions bits |
| M*20-bit | Total Constants bits |

Table 4.6: Structure of Context Memory segments

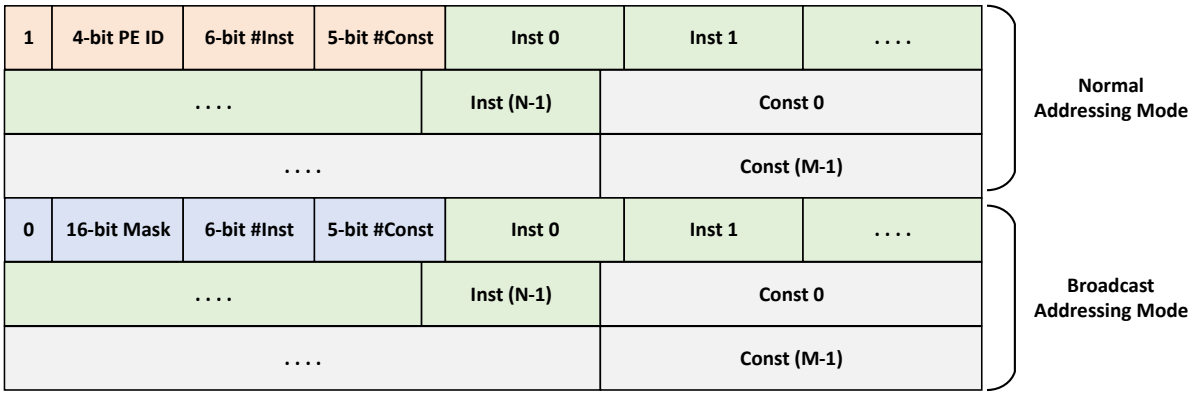



Figure 4.12: Segments of Context Memory

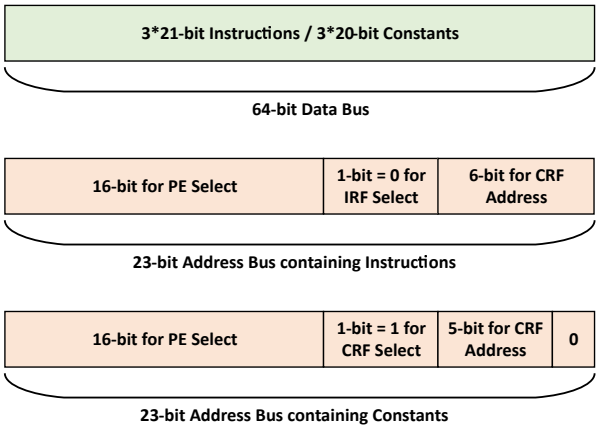



Figure 4.13: Data and Address Bus in the configuration network

In the case of normal addressing mode, the following 4-bit are dedicated to identify the PE in the PE array. The succeeding 6-bit are dedicated to detect the number of instructions for the identified PE, and the following 5-bit are dedicated to detect the number of constants for the identified PE. The rest of the abutting bits that are dedicated to the identified PE are instructions and constants of the identified PE.

In the case of broadcast addressing mode, the following (masked) 16-bit are dedicated to identify the set of PEs in the PE array. The succeeding 6-bit are dedicated to detect the number of instructions for the set of PEs, and the following 5-bit are dedicated to detect the number of constants for the set of PEs. The rest of the abutting bits that are dedicated to the set of PEs are instructions and constants.

Figure 4.13 shows the Data and Address bus in the configuration network. Each 64-bit context data line that is fetched from the Context Memory through the data bus is either 3x21-bit instructions or 3x20-bit constants. If the context data line contains the instruction for PE, then the 23-bit address bus is segmented by dedicating (1) first 16-bit to either mask or address of target PE, (2) 1-bit (= 0 ) dedicated to select IRF, and (3) 6-bit dedicated for addresses in IRF. If the context data line contains the data for PE, then the 23-bit address bus is segmented by dedicating (1) first 16-bit to either mask or

address of target PE, (2) 1-bit (= 1 ) dedicated to select CRF, (3) 5-bit dedicated for addresses in CRF, and (4) 1-bit (= 0 ) for zero-padding the extra space.

## Execution

During execution, in each clock cycle, PE fetches a 21-bit instruction from their respective IRF and decodes it according to the ISA format shown in Table 4.5. Particularly, the opcode (See Table 4.7 for a list of opcodes) is decoded, and it is determined if the operation is an integer operation or an FP operation. The sources for operands and destination of output are determined.

| Mnemonic | Datatype | Description | Operation |
|---|---|---|---|
| – | – | No Operation | – |
| EoE | – | End-of-Execution | – |
| U_ADD | int | Unsigned Add | R = (U)In1 + In2<br>C = 0 |
| S_ADD | int | Signed Add | R = In1 + In2<br>C = 0 |
| SUB | int | Subtract | R = In1 - In2<br>C = 0 |
| MUL | int | Multiply | R = In1 * In2<br>C = 0 |
| LSHIFT | int | Left Shift | R = In1 << In2<br>C = 0 |
| RSHIFT | int | Right Shift | R = In1 >> In2<br>C = 0 |
| LOAD | int | Load | – |
| STORE | int | Store | – |
| MOV | int | Move | R = In1<br>C = 0 |
| AND | int | Bitwise AND | R = In1 & In2<br>C = 0 |
| OR | int | Bitwise OR | R = In1 \| In2<br>C = 0 |

| | | | |
|---|---|---|---|
| NOT | int | Bitwise NOT | R = ~ In1<br>C = 0 |
| XOR | int | Bitwise XOR | R = In1 ^ In2<br>C = 0 |
| LTE | int | Less-than or Equal | if (In1 $\leq$ In2) , C = 1<br>else C = 0 |
| GTE | int | Greater-than or Equal | if (In1 $\geq$ In2) , C = 1<br>else C = 0 |
| NE | int | Not Equal | if (In1 $<$ In2) , C = 1<br>else C = 0 |
| DIV | int | Divide | R = In1 $\div$ In2<br>C = 0 |
| ABS | int | Absolute Value | R = \| In1 \|<br>C = 0 |
| fABS | float | float Absolute | R = \| In1 \|<br>C = 0 |
| fLT | float | float Less-Than | if (In1 $<$ In2) , C = 1<br>else C = 0 |
| fMUL | float | float Multiply | R = In1 * In2<br>C = 0 |
| fADD | float | float Add | R = In1 + In2<br>C = 0 |
| fSUB | float | float Subtract | R = In1 - In2<br>C = 0 |
| fDIV | float | float Divide | R = In1 $\div$ In2<br>C = 0 |
| fSQRT | float | float Square-Root | R = $\sqrt{In1}$<br>C = 0 |

Table 4.7: Summary of Opcodes (R = Result, C = Condition bit)
Gray colored cells represent newly added opcodes.

## 4.3 Summary and Concluding Remarks

In this chapter, CGRA architecture and different design optimizations to implement FP support under the ULP power budget are presented. Particularly, four different design optimizations for FP computation and one exploration to support NN computation are shown. The chapter begins with a brief motivation for the different design choices with optimizations adopted for our proposed CGRA. Later follows a brief description of each component in the CGRA sub-system with a detailed description of each design optimization made during the design space exploration. Finally, the computation model of the proposed CGRA is discussed.

# Chapter 5

# Compiler Support

## Contents



Typically the algorithms used to compile codes for CGRAs feature any or all of the three common characteristics, which are (1) support for static scheduling, (2) dynamic reconfiguration, and (3) either limited to specific hardware properties or use a special language or both. This is because most compiler research has evolved from algorithms used for placement and routing of FPGAs [109] in combination with generating code for VLIW processors like hyperblock formation [110] and modulo scheduling [111, 112]. Apart from those characteristics, a compiler must preserve the meaning of the source code (i.e., correctness), produce efficient target code in terms of space and time (i.e., maximum output execution speed and minimum memory footprint), ease of use (i.e., user-friendly front-end), easily debugged, correct and understandable optimizations, and does all of those tasks quickly (i.e., quick compilation).

## Contribution and Outline of the Chapter

This chapter presented the compilation flow for the proposed CGRA and preceded by, as part of motivation, two problem statements, i.e., (1) efficiently mapping of kernels with multi-cycle operations onto proposed CGRA and (2) encoding of the required data for address generation by Flexible-AGU (See section 4.1) within the LOAD/STORE instructions using current compiler are presented. The outline of this chapter is as follows:

- Problem statements.

- Walk-through of the *Compilation Flow* for proposed CGRA.
- Mapping of multi-cycle operations onto CGRA using Static Mapping approach.
- Development of a 5-stage pass to efficiently decouple the address generation branches from DFGs for Flexible-AGU.
- Assembler and optimization techniques used.
- As part of exploration, a solution to auto-partition the large kernels for the proposed CGRA is presented.

Finally, a summary and concluding remarks of this chapter are provided.

## Problem Statement

Change propagation is a foremost element to consider during the development of any (sub-)system. Any change(s) in architecture or to its element(s) must be propagated through to the toolchain to keep consistency throughout the design cycle. The two main problems to be addressed in the current compiler w.r.t. current design optimizations in the proposed CGRA are:

### Statement 1

Efficiently mapping a multi-cycle operation onto the same PE due to the implementation of FP operators in the proposed CGRA.

### Statement 2

Decoupling of address generation branches from DFGs and encoding required data for address generation to be used by Flexible-AGU (See section 4.1) within the LOAD/STORE instructions.

Next in this chapter, a walk-through of the current compilation flow and then proposed solutions to the aforementioned problem statement are presented.

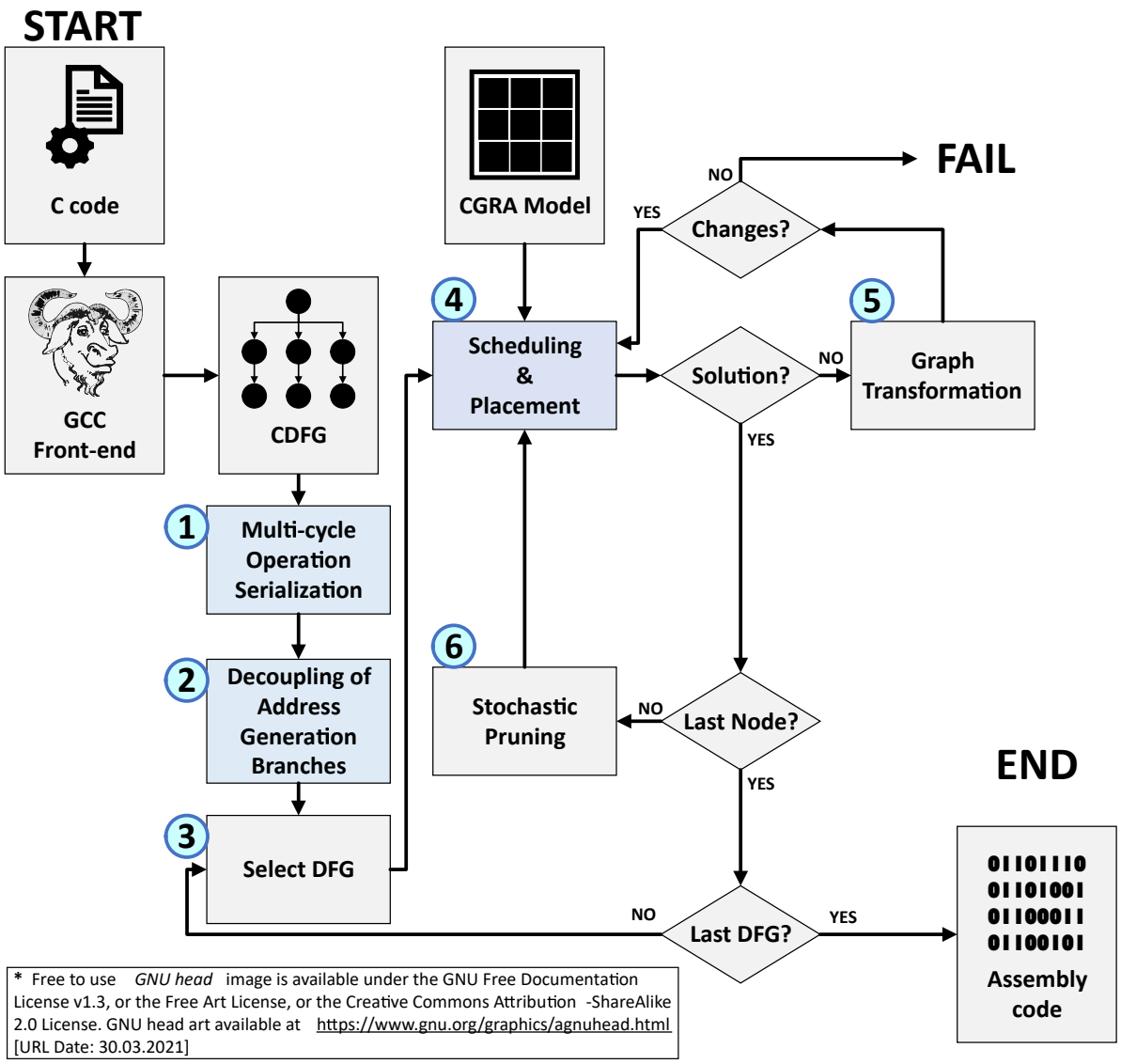



Figure 5.1: Compilation Flow

## 5.1 Compilation Flow

Figure 5.1 shows the compilation flow used for mapping CDFG of kernels onto proposed CGRA. The mapping process is automated through a software tool implemented using an Eclipse-based modeling framework and code generation solution called Eclipse Modeling Framework (EMF) [85, 86]. The internal representation (i.e., CDFG) is generated from the GCC front-end using a dedicated GCC plugin [113]. The dedicated GCC plugin is introduced to transform the C syntax tree into a machine-independent intermediate representation, i.e., GIMPLE [114] representation. The obtained CDFG is interpreted and used to generate assembly code for the target CGRA by the compiler.

Figure 5.2 shows the EMF model of CDFG implemented in the compiler. Figure 5.2 includes all the Activity Nodes represented as classes along with their type of relationship

with other classes. Particularly, *EndLoad* and *BeginLoad* classes have a two-way reference, and *ConditionalJump* and *UnconditionalJump* inherits from *JumpInstruction* classes, respectively.

The Java model of CGRA implemented in the compiler and it includes all the definitions of modules of CGRA represented as classes along with there type of relationships with other classes. Particularly, *CGRA* class contains (1) *ConfigMemory* (i.e., Instruction Memory), (2) *DataMemory* (i.e., Data Memory), (3) *Link* (i.e., represents all types of connections in the CGRA), and *Tile* (i.e., PE) classes. *Tile* class includes all the classes which represents the components of a PE in the CGRA. *Tile* class contains *MemorizationUnit* and *ComputationUnit*. Furthermore, *Memorization* class represents the register part of a PE which are *RF* (i.e., Regular Register File), *ConstantRegisterFile* (i.e., Constant Register File), and (3) *ConfigReg* (i.e., Instruction Register File). *ComputationUnit* class represents the computation part of a PE which are (1) *LoadStore* (i.e, Load Store Unit) , (2) *ALU* (i.e., Arithmetic Logic Unit) and, (3) *FPU* (i.e., Floating Point Unit).

DFGs (or Basic Blocks) that constitute a CDFG are mapped onto the PE array of the CDFG because of the homomorphism between the two graphs (i.e., (1) DFG and (2) CGRA model).

An existing compilation framework is used in this thesis which uses a register allocation approach to map a DFG onto a PE array [88]. First, the DFGs are queued in a series. This ordering is done either based on a DFG allocating the highest number of registers to share temporary data with other DFGs, e.g., loop variables, etc., or based on their occurrence in the CDFG from top to bottom [80]. Then for each DFG, a set of mappings are found which are compatible with the already mapped DFGs by setting the constraints, which are:

- Target location constraints are related to data that is used within the mapping stage of a basic block.
- Reserved location constraints restrict the use of some resources due to the data which are not used in the basic block but need to be considered during the mapping stage of a basic block.

If no mapping is found, then the compiler transforms the graph to ease the mapping [89]. If there can not be any transformation applied to a DFG, then there is no valid mapping solution for the selected DFG. Then a backtracking mechanism is used to find another compatible set of mapped DFGs to map the current DFG. All the valid mappings found for the current DFG are stored in a mapping bank. The mapping approach described in [90] is used in the compiler for the proposed CGRA. The compilation flow presented in this chapter extends to add support for multi-cycle operations and decouple the address generation branches in the DFGs due to the implementation of Flexible-AGU in the proposed CGRA. The compilation flow consists of six stages i.e., (1) *Multi − cycle Operation Serialization* (new contribution), (2)

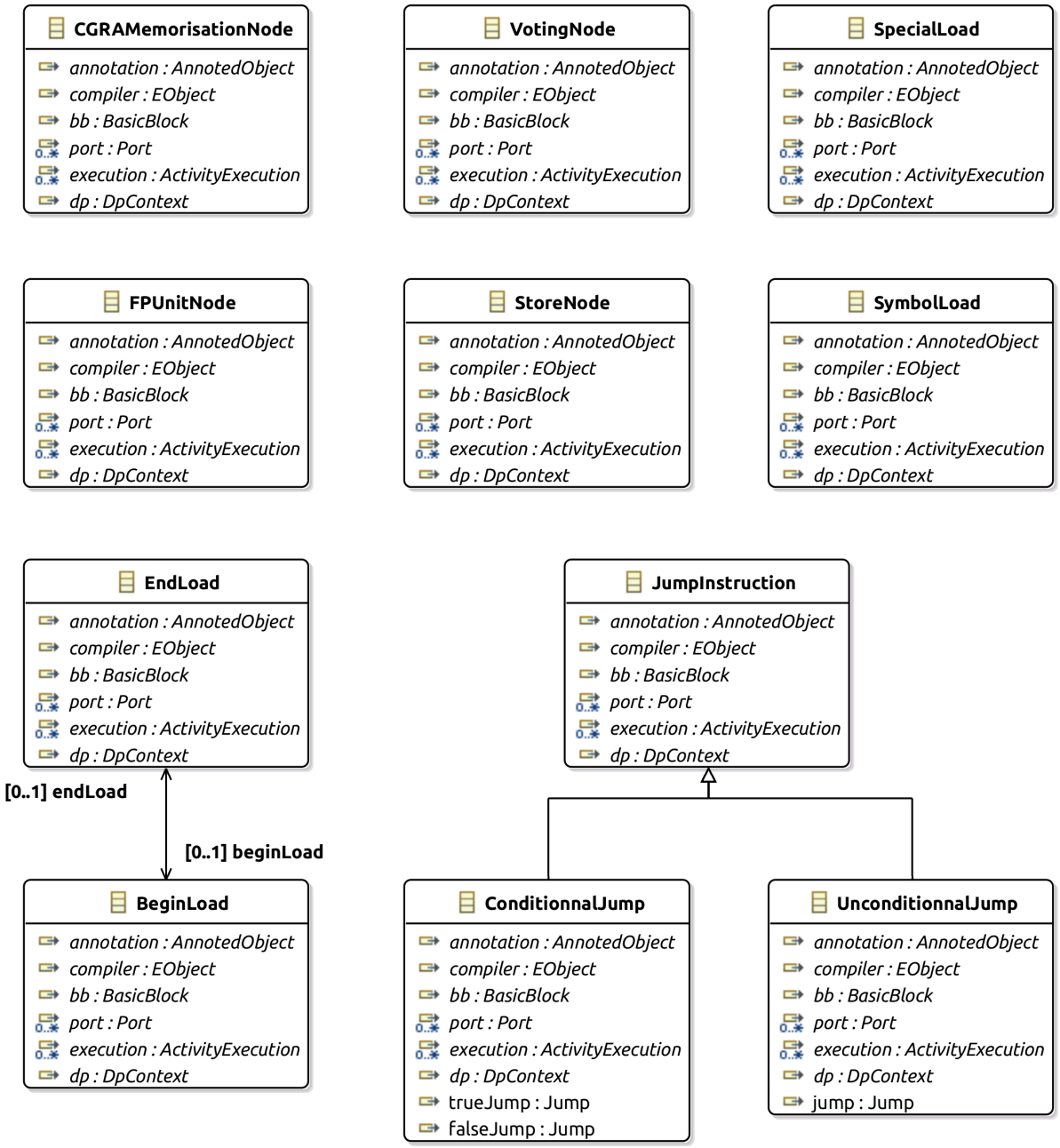



Figure 5.2: EMF model of CDFG

*Decoupling of Address Generation Branches* (new contribution), (3) Select DFG, (4) *Scheduling and Placement* (modified step), (5) Graph Transformation, and (6) Stochastic Pruning. In this section, steps involved in addressing the aforementioned two problems are discussed.

| Keyword | Description |
|---|---|
| fABS$$ | float Absolute |
| fLT$$ | float Less-Than |
| fmul$$ | float Multiply |
| fadd$$ | float Add |
| fsub$$ | float Subtract |
| fdiv$$ | float Divide |
| fsqrt$$ | float Square-Root |

Replace $$ with *32*, *16*, *16alt*, and *8* to represent IEEE 754-2008 *binary32*, IEEE 754-2008 *binary16*, *binary16alt*, and *binary8* datatype respectively.

Table 5.1: Reserved keywords for FP datatype

### 5.1.1 DFG Mapping with multi-cycle operations

The compiler shares a common framework with a High-Level Synthesis (HLS) tool called GAUT [87]. While compiler can easily detect primitive datatype i.e., *int* and *float* however, it is difficult to detect custom datatype (i.e., *binary16alt* or *binary8*), as compilation flow depends on CDFG provided by the GCC interpretation of the C code. Some keywords (See Table 5.1) are reserved and supplied in the C code to facilitate the SIMD feature for the custom float data. For simplicity, same technique is also applied for IEEE 754-2008 *binary32* and *binary16* datatype. The compiler detects these keywords and generates instructions. These instructions are later interpreted by the assembler in the assembly code to generate the bit-stream with specified opcodes. Code 5.1 and Code 5.2 are C code examples using reserved keywords for custom float datatype to represent the ease of use to write kernels at the GCC front-end:

```
// ** binary16alt operation **
int i;
float A[10], B[10], C[10];
float OUT = 0.0f;
for (i = 0; i < 10; i++){
        // OUT = (A[i] * B[i]) + C[i];
        OUT = fadd16alt ( fmul16alt(A[i] , B[i]) , C[i] );
}
```

Code 5.1: Modified C code for binary16alt operations

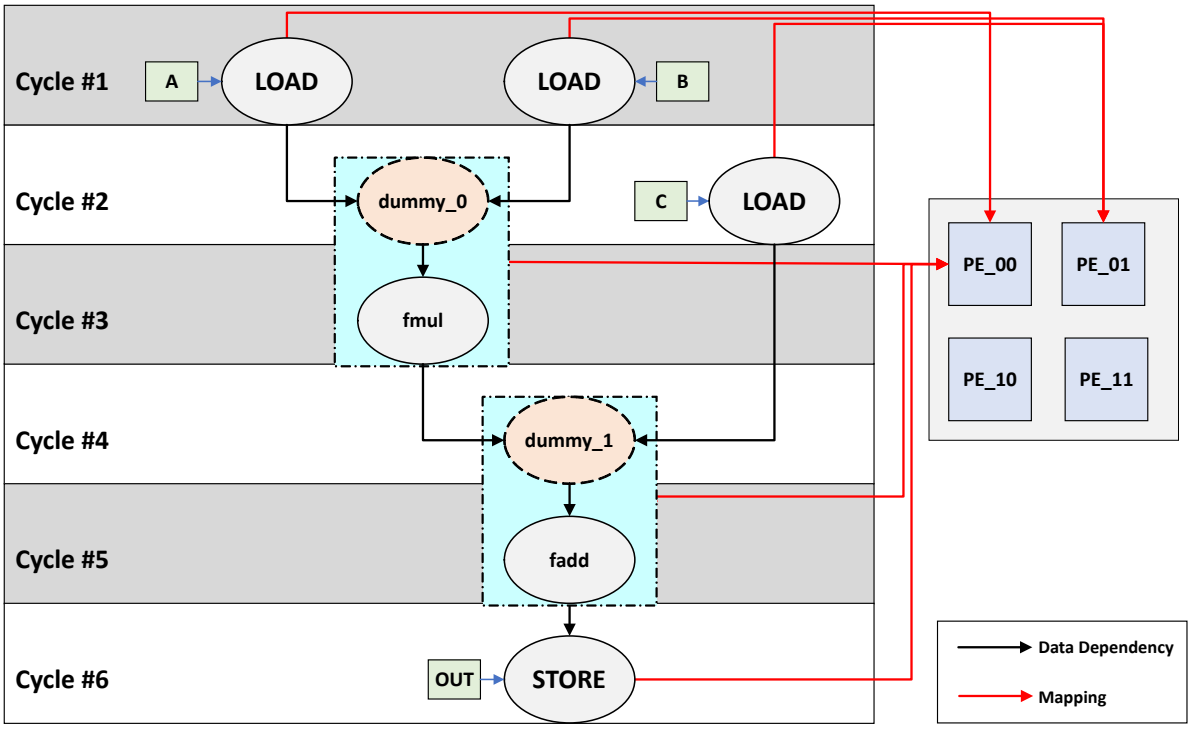



Figure 5.3: Graph Transformation to add dummy nodes and Mapping of DFG nodes onto PEs

```
// ** binary8 operation **
int i;
float A[10], B[10], C[10];
float OUT = 0.0f;
for (i = 0; i < 10; i++){
        // OUT = (A[i] * B[i]) + C[i];
        OUT = fadd8 ( fmul8(A[i] , B[i]) , C[i] );
}
```

Code 5.2: Modified C code for binary8 operations

### Multi-cycle Operation Serialization

The compilation flow detects the multi-cycle operations (i.e., FP operations) in the CDFG. It transforms those operation nodes by adding dummy nodes equal to the number of clock cycles required to perform those operations. Figure 5.3 shows such graph transformation to add dummy nodes and mapping for 2 clock-cycle FP operations (i.e., *fmul* and *fadd*) by adding a dummy node to each FP operation node (i.e., *dummp_0* and *dummy_1*). The transformed DFG is then passed to the next step in the compilation flow.

### Select DFG

In this step, a DFG is selected from the CDFG using forward traversal-Breadth First Search (BFS) [80], and in the next step, nodes in selected DFG are mapped onto CGRA. Once all DFGs are mapped, the compilation flow reaches the end of compilation, and assembly code is generated.

### Graph Transformation

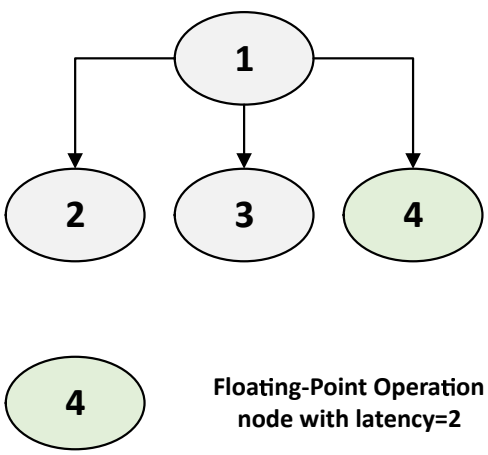
Graph transformation performs the dynamic transformation of a DFG during the mapping process. Figure 5.4 shows the different types of graph transformation which the compiler can perform on the fly. Figure 5.4a shows a sample DFG where node 4 is a floating-point operation with a latency of 3 cycles. Figure 5.4b shows multi-cycle serialization. In this step, the compiler detects the multi-cycle operations (i.e., FP operations) and adds dummy nodes according to their latency. In Figure 5.4b two dummy nodes are added, i.e. *D1* and *D2*. In case the binding step does not find any solution for a node, the compiler transforms the graph by either re-routing or distributes the fan-out to find a valid mapping solution by satisfying the available connectivity constraints of the CGRA. Figure 5.4c shows the re-routing and Figure 5.4d shows the re-computing steps, respectively.

### Scheduling and Placement

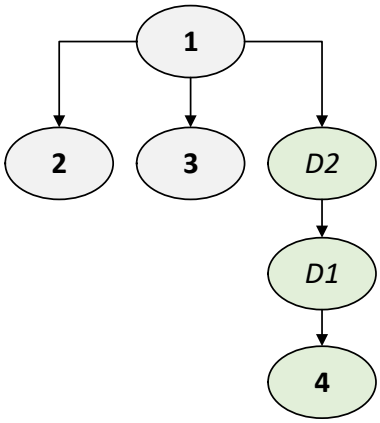
In this step, a backward traversal list scheduling algorithm [89] is used to schedule the nodes in each DFG, where schedulable operations are listed by priority order. The priority of a node depends on its weight, followed by mobility and total fan-outs. The weight of an operation is decided by its latency of execution in clock cycles. A node is schedulable in a backward traversal list scheduling algorithm if and only if all of its child nodes are scheduled already. The difference decides mobility between ALAP and ASAP schedule length. These make the FP operations to be scheduled with the highest priority order.

An incremental version of Levi's [97] sub-graph matching algorithm is used for the binding step in the compilation flow. The proposed algorithm is exact in nature and adds the currently scheduled operation node (except FP operations) and its associated data node to the sub-graph, composed of previously scheduled and bound nodes. Only the previously retained set of solutions are used to find possible mapping solutions to add this new set of scheduled nodes (i.e., operation and data nodes). In this step, future nodes, i.e., yet to be scheduled nodes, are not considered.

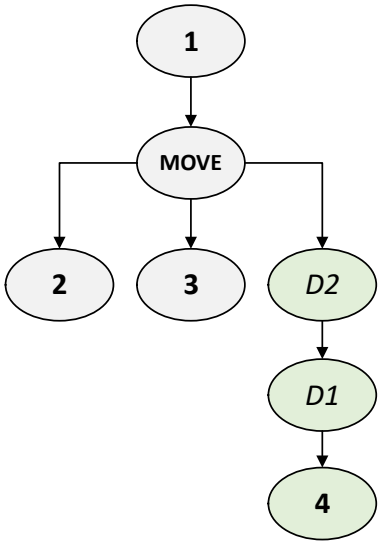
The binding of FP operation follows a static mapping approach, where the algorithm skips the dummy nodes and finds a binding solution for the first operation node among all the FP nodes (nodes $fmul$ and $fadd$ in Figure 5.3). The same binding solutions are



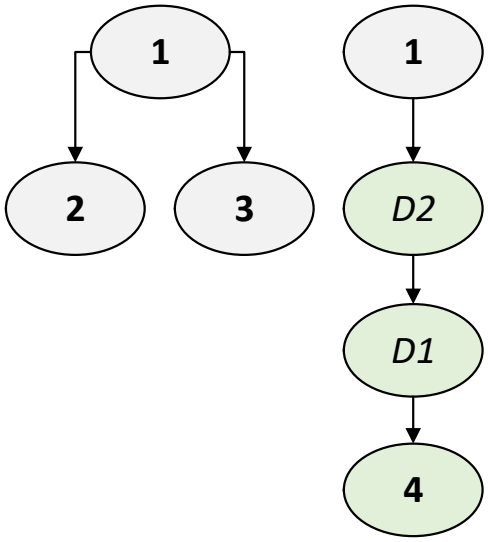

(a) Sample DFG with FP operation




(b) Multi-cycle Operation Serialization




(c) Re-routing




(d) Re-computing

Figure 5.4: Graph Transformation

then copied to the rest of the dummy nodes, i.e., the binding solution of $fmul$ is copied to *dummy_0*, and the binding solution of $fadd$ is copied to *dummy_1*. Such binding of FP operations enables the execution of multi-cycle operations onto the proposed CGRA with static scheduling. In static scheduling, only the first encountered node of a multi-cycle operation is mapped. The same mapping solution is then copied to the rest of the nodes of that particular multi-cycle node, where the timestamp of the dummy nodes, which copy the mapping solution of the already mapped operation node, is adjusted accordingly.

### 5.1.2 Decoupling of address generation branches for Flexible-AGU

To decouple the address generation branches from a DFG, a 5-stage pass has been implemented in the compiler. Implementation of Flexible-AGU can save up to 9 clock

cycles (in case of a 2D variable, see assembly code below). In this section, a typical 2D variable compatible with the Global Index representation (See section 4.1) is used to explain decoupling address generation branches for Flexible-AGU and how the required data is encoded in the memory instructions to be used in the hardware unit.

```
// ** Assembly code for address generation of **
// ** Input[(i+A)*(j+B)][(k+C)*(l+D)] **
#1 PE_00 ADD R2 i A
#2 PE_00 ADD R3 j B
#3 PE_00 ADD R4 k C
#4 PE_00 ADD R5 l D
#5 PE_00 MUL R2 R2 R3
#6 PE_00 MUL R4 R4 R5
#7 PE_00 ADD R2 R2 R4
#8 PE_00 MUL R3 R2 4
#9 PE_00 ADD R3 R3 BA // ** 32-bit address is stored in R3 **
#10 PE_00 LOAD Input // ** Data LOAD **
```

Code 5.3: Sample Assembly code snippet for CGRA

Code 5.3 represents assembly code of the address generation for a typical 2D variable (i.e., Input[(i+A)*(j+B)][(k+C)*(l+D)] where, i,j,k,l are the loop variables and A,B,C,D are the variables / constants) that must be executed before a memory operation. The use of Flexible-AGU allows eliminating such a series of execution of assembly code lines before any memory operation (i.e., LOAD/STORE). The required data for address generation is encoded within memory operation instruction (See section 4.1).

Figure 5.5 represents a possible DFG for address generation for Input[(i+A)*(j+B)] [(k+C)*(l+D)]. To carefully detect the address generation branches in a DFG, a 5-stage pass is implemented, which detects the address generation branches and incrementally decomposes the branches into a series of data nodes attached to the memory operation node while eliminating the unwanted operation nodes.

Detection of address generation branches for LOAD operation (See Figure 5.6a) is relatively straightforward w.r.t. the detection for STORE operation. There is no computation branch whose target node is attached to a LOAD operation node (i.e., the LOAD operation node does not have source nodes). In the case of LOAD operation, the algorithm recursively detects the data nodes and eliminates the encountered operation nodes. However, each data node is evaluated for correct alignment for the LOAD operation node. To detect the Base Address (BS), the detection pattern is as follows; first, the immediate source node (i.e., ADD node) of the LOAD operation node has 2 source nodes, if one of the data nodes is evaluated as an integer value and has zero source nodes, then the current data node contains the BA value. Then, the algorithm recursively searches and finally ends up with data nodes that do not have any source operation nodes. In between, the algorithm also detects the first index (i.e., Index1 = [(i+A)*(j+B)]) and second index (i.e., Index2 = [(k+C)*(l+D)]) and finally aligns them as source nodes to LOAD operation node in a series (See Figure 5.6b) like

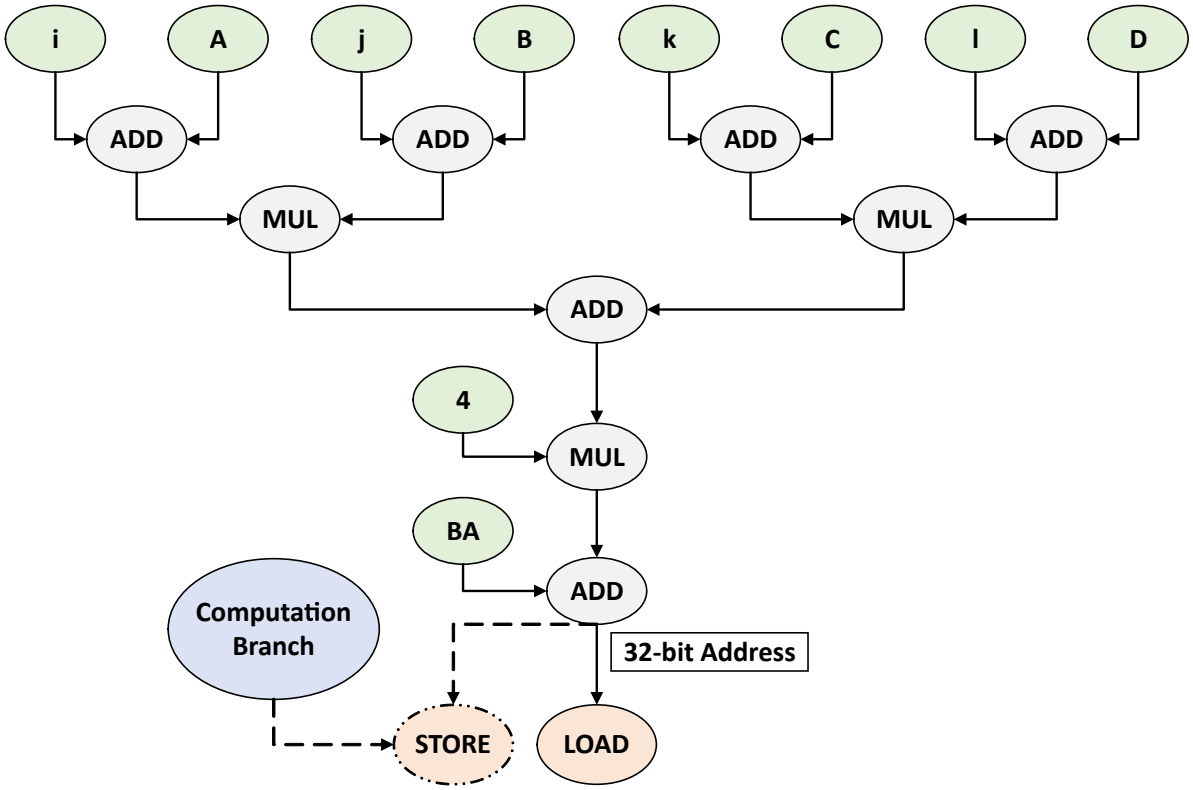



Figure 5.5: DFG with address generation branches

$$\text{Index1} \rightarrow \text{BA} \rightarrow \text{Index2}$$

and the LOAD instruction is as follows

| timestamp | #PE | **LOAD** | Reg | Index1 | BA | Index2 |
|---|---|---|---|---|---|---|

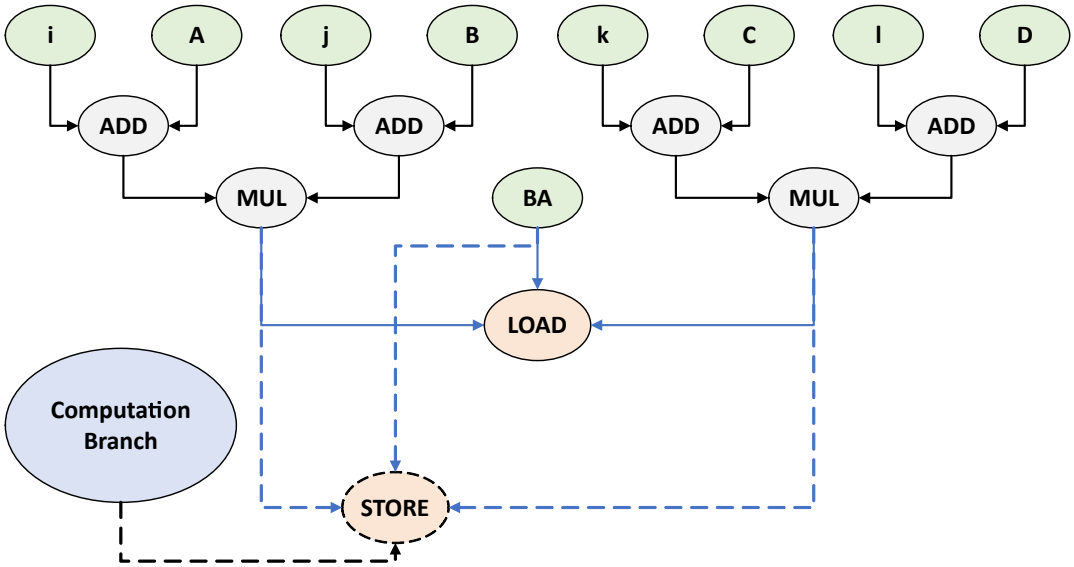
A similar approach is applied for the STORE operation node, but the algorithm also detects and avoids pruning of the computation branch (See Figure 5.6). This has been done after exhaustively analyzing each possible DFG pattern for any combination of variable representations that fall under the predefined Global Index representation. The alignment of required data nodes for the STORE operation node is like

$$\text{Computation Branch} \rightarrow \text{Index1} \rightarrow \text{BA} \rightarrow \text{Index2}$$

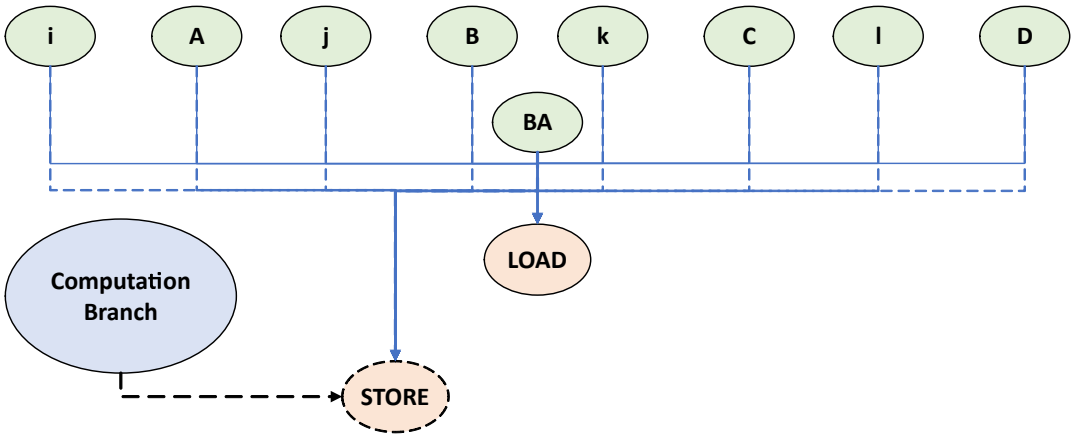
and the STORE instruction is as follows

| timestamp | #PE | **STORE** | Reg | Index1 | BA | Index2 |
|---|---|---|---|---|---|---|

The Assembler section explains how the assembler encodes the obtained assembly code lines for memory operations to generate 21-bit instructions for the proposed CGRA.




(a) DFG with sorted address generation branches




(b) DFG with address generation data

Figure 5.6: Decoupling of address generation branches in DFG

## 5.2 Assembler

The Toolchain for the proposed CGRA has been split into (1) compiler (to efficiently map kernels onto CGRA) and (2) assembler (to further optimize the generated assembly code to produce the required bit-stream). The assembler takes the ASCII text assembly generated by the compiler during the mapping process. The Instruction Set Architecture (ISA) of the proposed CGRA produces the required machine code for PE array configuration. The generated machine code consists of instructions and constants that will be broadcast to the IRF and CRF of PEs. Memory instructions (i.e., LOAD/STORE instructions) also include the addresses of the indexes stored in the CRF, which Flexible-AGU will use for address generation. Mainly, the assembler adds the following optimizations and encoding techniques (as a part of new contribution ) while generating

the required machine code:

- removes instructions generated for dummy nodes used to map multi-cycle operations (See dummy_0 and dummy_1 in Figure 5.3).
- eliminates any duplicate entry of constants in the CRF of PEs.
- encodes the information required by the memory instructions (explained below).

Alongside the assembly code file, compiler also generates a configuration file which consists of information about the loop variables used in the application. These information are (1) START, (2) STEP, and (3) END values of each loop variables. A parser is implemented to manipulate these values and generate the required 20-bit data for each loop variables which represents the loop variable line in CRF (See Section 4.1) and the structure of such lines are as follow:

| MSB | 10-bit | 8-bit | LSB |
|---|---|---|---|
| 1/0 | START | STEP | 1/0 |

where, MSB determines if START (10-bit) value is a loop variable (MSB=1) or a constant (MSB=0) and LSB determines if the STEP (8-bit) value should be subtracted (LSB=1) or added (LSB=0) form the START value in the next loop iteration.

After this step, a second parser is implemented to manipulate the memory instructions present in the assembly code file. This step formats each memory instructions in the following structure

timestamp #PE **LOAD/STORE** Reg i A j B BA k C l D

where, i,j,k,l are the loop variables and A,B,C,D are constants/variables.

If any of these values are missing, the parser will produce the memory instruction lines accordingly. Few examples are as follows:

- if the variable is *Input[i]*, then instruction line would be

  timestamp #PE **LOAD/STORE** Reg i 0 1 0 BA 0 0 0 0

- if the variable is *Input[i][j]*, then instruction line would be

  timestamp #PE **LOAD/STORE** Reg i 0 1 0 BA j 0 1 0

- if the variable is *Input[i+A]*, then instruction line would be

  timestamp #PE **LOAD/STORE** Reg i A 1 0 BA 0 0 0 0

- if the variable is *Input[i][A]*, then instruction line would be

  timestamp #PE **LOAD/STORE** Reg i 0 1 0 BA 0 A 1 0

Next, the assembler interprets these memory instructions and produce Index Pair lines (i.e., ($i$+A) or ($j$+B) or ($k$+C) or ($l$+D)) to be stored in CRF. The structure of these lines are as follows:

| MSB | 9-bit | 10-bit |
|---|---|---|
| 1/0 | LV address/Constant | Constant |

where, MSB determines if *i,j,k or l* are loop variables (MSB=1) or a constant value (MSB=0). In the case of MSB=1, then the 5-bit address of the loop variable is stored at [15:11] of the Index Pair line.

The assembled line (i.e., Index Address line) in CRF, which represents the addresses of such Index Pair lines (which is referenced in the memory instructions), has a structure like this

| 5-bit | 5-bit | 5-bit | 5-bit |
|---|---|---|---|
| ($i$+A) address | ($j$+B) address | ($k$+C) address | ($l$+D)) address |

Finally, the assembler takes the modified memory instructions and generates the 21-bit instruction to be included in IRF. The structure of 21-bit memory instructions are as follows:

| MSB | | | | | LSB |
|---|---|---|---|---|---|
| 5-bit | 1-bit | 5-bit | 1-bit | 3-bit | 6-bit |
| Index Addr | Type (1) | BA Addr | Type (1) | DRA | LOAD |
| Index Addr | Type (1) | BA Addr | Type (1) | DRA | STORE |

DRA : Destination Register Address
BB : Basic Block
LV : Loop Variable
Type=1 : Source is in CRF

A PE decodes (See section 4.1) such encoding when a memory instruction is encountered to calculate the required address for fetching/storing data from/in the memory.

## 5.3 Summary and Concluding Remarks

This chapter presents the contribution in the compilation flow for the proposed CGRA. The chapter begins with a general introduction of the compiler support for CGRAs and then follows the background work. The focus of this chapter is on the compilation

flow for the proposed CGRA. The chapter discusses the compilation flow and how the homomorphism between the Application model and CGRA model made the mapping problem a sub-graph finding problem. Finally, the main contribution of this chapter, mapping of multi-cycle operation and decoupling of address generation branches from the DFGs, are discussed. Next chapter 6 presents a heterogeneous cluster featuring *transprecision* computing, including both the RI5CY sub-system and the CGRA sub-system.

# Chapter 6

# Heterogeneous Platform for Transprecision Computing

## Contents



This chapter discusses the design and implementation of a heterogeneous cluster, including the *transprecision* computing-based CGRA sub-system. The CGRA design is explained in detail in section 4.1.4, and this chapter mainly focuses on the integration of CGRA into the PULP-Cluster of the PULP SoC [2] (See section 3.6).

## Contribution and Outline of the Chapter

The outline of this chapter is as follows:

1. Design and implementation of a heterogeneous platform, particularly,
    - integration of a *transprecision* computing-based CGRA into PULP-Cluster and implementation of optimization techniques to optimize the performance of the proposed heterogeneous cluster,
    - description of the software infrastructure employed to enable the heterogeneous platform,

- memory map of PULP SoC and dedicating address space for the CGRA,
- workload synchronization between CGRA and RI5CY sub-systems, and
- manual mapping approaches adopted for mapping kernels on CGRA for obtaining high PE utilization.

Finally, a summary and concluding remarks are provided at the end of this chapter.

## 6.1 Heterogeneous Platform

PULP-SoC [2] (See Figure 3.7 and Figure 3.8) exploits the flexible aspect of the RISC-V ISA to provide a multi-core RI5CY sub-system and also allows coupling of a programmable parallel processing engine (CGRA sub-system) for flexible near-sensor processing (i.e., audio, image, bio-signals, and embedded ML). The parallel computing heterogeneous cluster (Figure 6.1) contains 8 RI5CY cores supporting the RV32IMC instruction set [99], with an extension supporting SIMD-style vectorization and targeting energy-efficient digital signal processing (*Xpulp*) [48].

The heterogeneous cluster presented is based on the parallel compute cluster domain (aka PULP-Cluster) of the PULP Architecture (See section 3.6). The CGRA sub-system includes a 4x4 PE array, and the architecture is presented in section 4.1.4. This section focuses on the integration of the CGRA sub-system in PULP-Cluster and the implementation of optimization techniques.

### 6.1.1 CGRA Integration

Figure 6.1 shows the organization of CGRA and RI5CY in the parallel compute cluster domain (aka PULP-Cluster) of PULP-SoC. In the set-up of the parallel compute cluster domain, the global wires to the CGRA Integration module are connected to

- Cluster Interconnect for transferring data between TCDM and CGRA sub-system,
- Cluster Bus for transferring data between CGRA and L2 memory,
- Event Unit to interact with the cluster events for CGRA (including events for clock-gating between CGRA and RI5CY),
- Cluster Timer to access the cluster timer unit for input triggers (i.e., Frequency-Locked Loop (FLL) clock, Prescaler to FLL clock, reference clock, and external event), counting modes, interrupt request, and

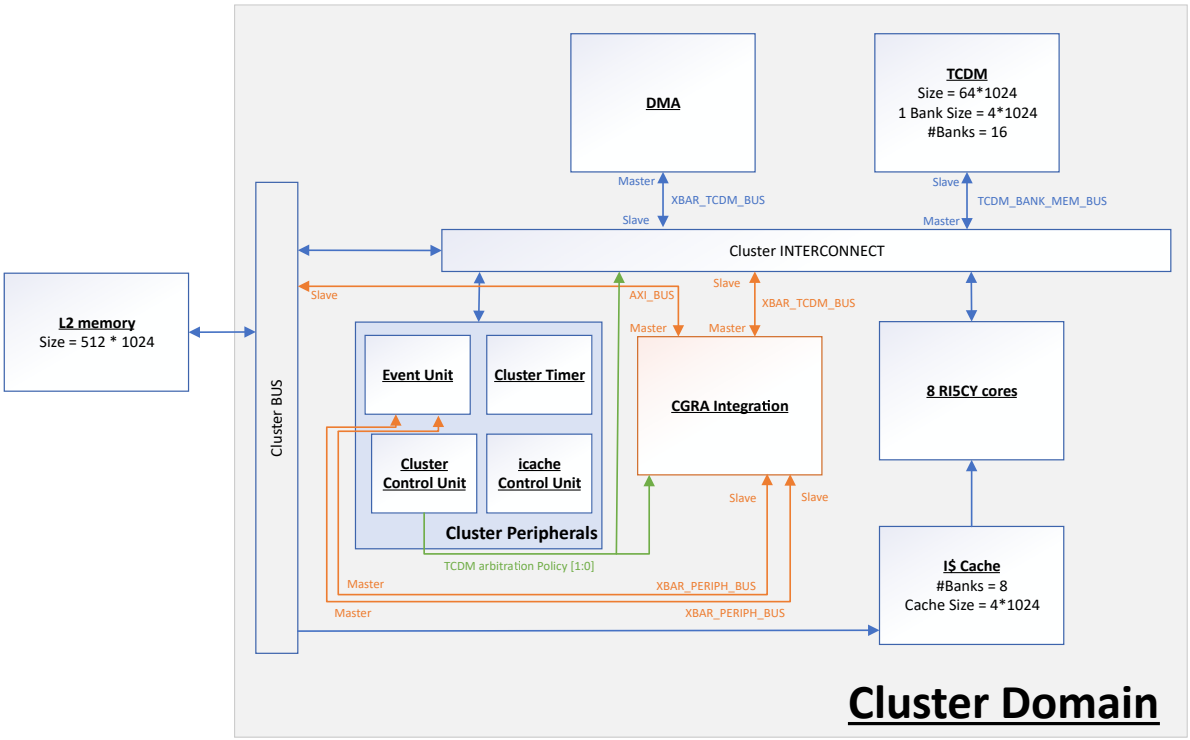



Figure 6.1: Organization of CGRA and RI5CY in heterogeneous cluster

- Cluster Control Unit to access the memory map of cluster domain.

Figure 6.2 shows the connections of wires inside the CGRA Integration module. CGRA Integration is a wrapper for three components which are the key blocks of the CGRA sub-system, (1) CGRA Wrapper which constitutes the PE array and DMA CGRA controller, (2) Global Context Memory, which store the context data, and (3) DMA controller dedicated for CGRA sub-system. Figure 6.3 shows the continuation of connections of the global wires of the CGRA Integration module.

### Clock-gating between CGRA and RI5CY sub-systems

A clock-gating scheme between CGRA and RI5CY sub-system is implemented to further improve the energy efficiency of the heterogeneous cluster. The working of the clock-gating scheme is as follows:

- **When RI5CY cores are executing:** CGRA sub-system detects that no task has been issued for the CGRA and issues a CLK_GATE_CGRA signal to clock-gate the PE array of CGRA.

- **When the CGRA sub-system is executing:** CGRA sub-system issues a CGRA_BUSY signal which is used to clock-gate the RI5CY sub-system. Particularly,

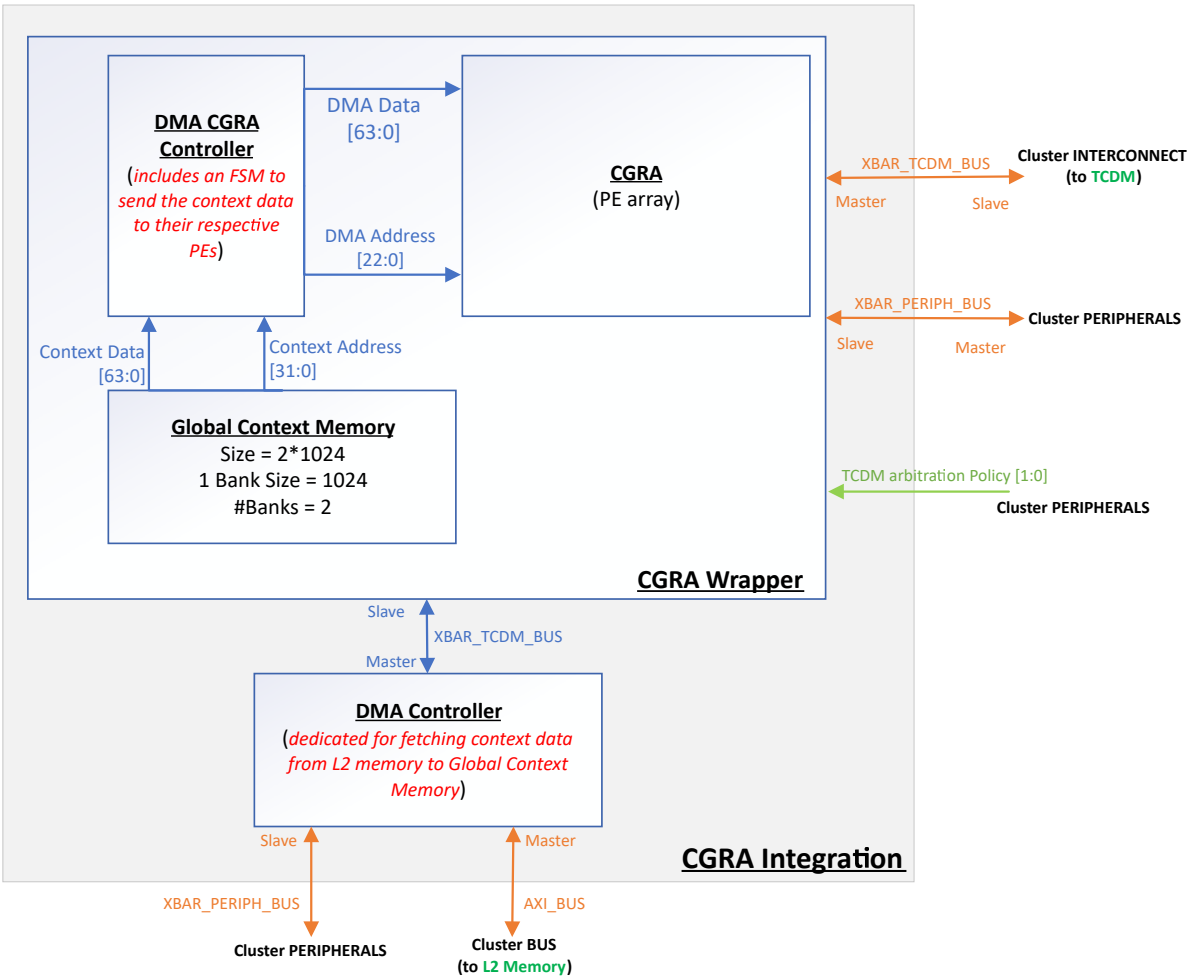



Figure 6.2: CGRA integration

1. clock-gate FPU Cluster

2. clock-gate all RI5CY cores but CORE[0] (CORE[0] is always active (1) to catch the interrupt signals within the heterogeneous cluster, and (2) to issue functions for Cluster Peripherals and DMA)

### 6.1.2 Software Infrastructure

CGRA also features a dedicated memory mapping of operations, and PEs access these control registers to offload works to the CGRA and synchronize the execution. There are two control registers, i.e., (1) command register and (2) status register, and then with the help of a simple Application Programming Interface (API), CGRA performs the offloading and synchronization of tasks in the cluster domain. Table 6.1 describes the main functions of this API. Before the initialization of execution on CGRA, the context and data from the L2 memory are loaded into GCM and TCDM to program the DMA

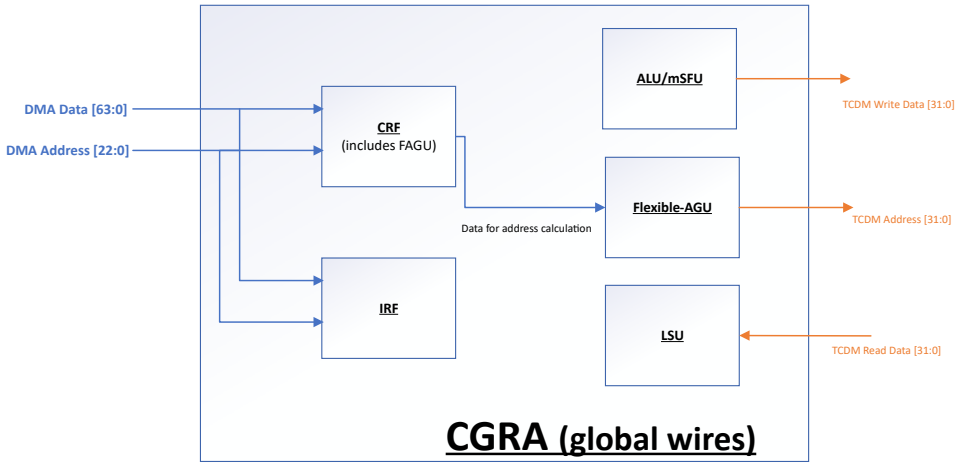



Figure 6.3: CGRA global wires

| Function | Details |
|---|---|
| void DATA_LOAD_L2toTCDM() | writes data from L2 memory to TCDM through DMA |
| void CGRA_LOAD_CONTEXT_L2toTCDM() | writes context data from L2 memory to the GCM through DMA CGRA Controller |
| int CGRA_START_EXECUTION() | Begin CGRA execution by writing in the command register |
| int CGRA_CHECK_STATUS(int ID) | CGRA synchronization |
| int CGRA_FREE_CGRA(int ID) | Release CGRA |

Table 6.1: API for controlling CGRA

(dedicated for CGRA sub-system) and DMA CGRA Controller. The context of the CGRA consists of instruction and constants for each PE (generated by the compilation flow described in 5.1). The API functions CGRA_LOAD_CONTEXT_L2toTCDM() and DATA_LOAD_L2toTCDM() execute a set of routines to write context and data into GCM and TDCM, respectively. The CGRA_START_EXECUTION() writes the execute command into the command register of the CGRA. Upon completion of the execution of workload in CGRA, the status register is updated. The status of the CGRA execution is updated by calling CGRA_CHECK_STATUS() by checking the status register. Finally, CGRA_FREE_CGRA() is called, and a signal is sent to the Event Unit in the cluster domain to notify the availability of CGRA and scheduling of the next task in the cluster domain.

### 6.1.3 PULP SoC Memory Map

Figure 6.4 represents the memory map of PULP SoC, mainly highlighting the reserved and non-reserved addresses. All area in the memory map is addressable from any RI5CY cores. Aliased addresses have a specific meaning when addressed from either Fabric Con-

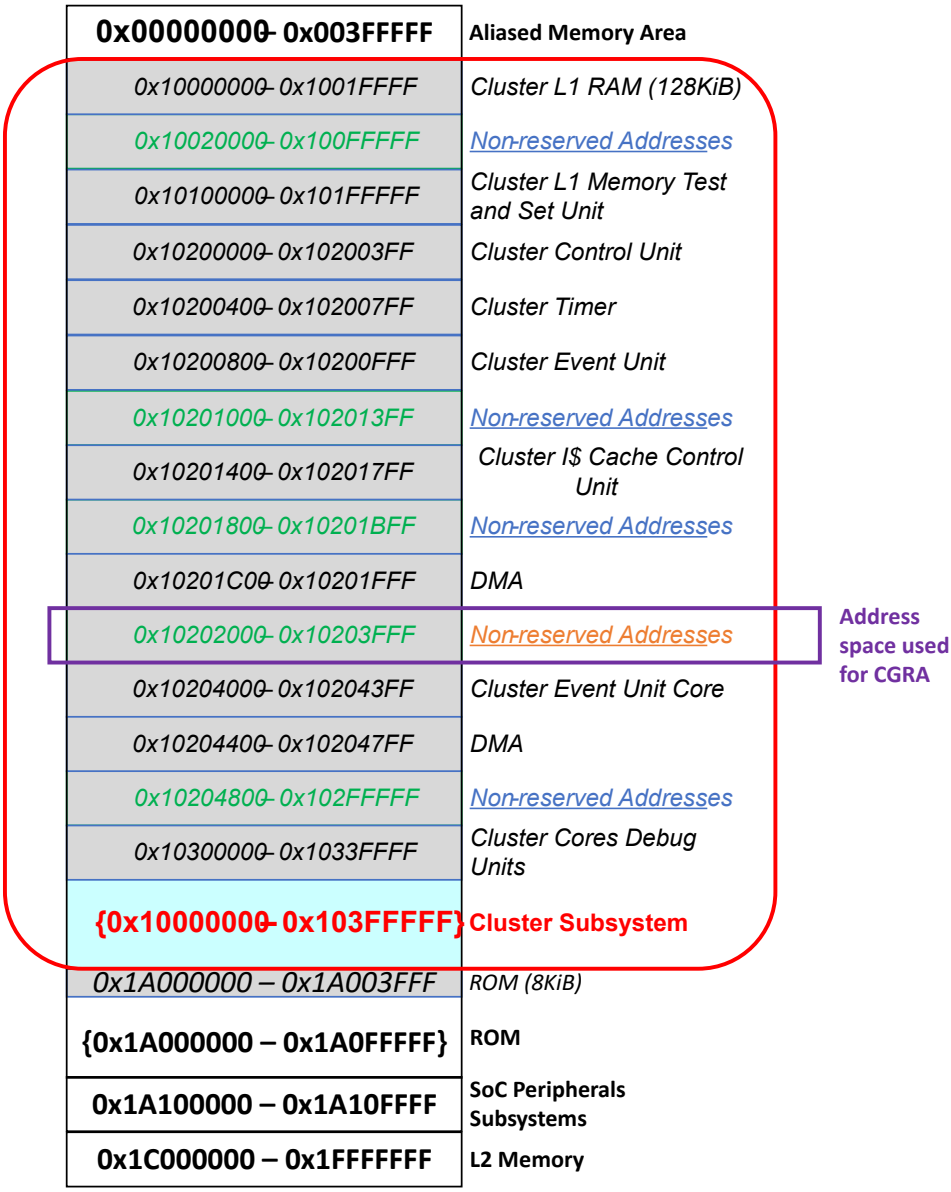



Figure 6.4: PULP SoC Memory Map

troller (FC) (i.e., a microcontroller like core for control, communications, and security functions in the PULP SoC) or RI5CY cores in the PULP-Cluster. Aliased addresses are preferred over standard addresses because they provide faster access that is typically 1 or 2 clock cycles.

### Reserving address for CGRA Events in PULP-Cluster components

Table 6.1 shows a set of functions in the API used for controlling and triggering CGRA events in the PULP-Cluster. Two main actions are required for efficient control of the CGRA sub-system using the API.

1. Position the CGRA_DMA peripheral on the slave port of the Cluster Interconnect to interact with the heterogeneous cluster peripherals.

    - For CGRA DMA events, position number 8 is reserved, which corresponds to address 0x10202000 in the memory map in Figure 6.4. This design choice resulted in CGRA_DMA COMMAND_REGISTER address to 0x10202000, and CGRA_DMA STATUS_REGISTER to 0x10202004. Command and Status registers are used to control and synchronize the CGRA DMA events using the APIs in Table 6.1. Depending on the number of tasks that can be serialized in the CGRA at once, the value of CGRA_DMA STATUS_REGISTER is incremented by 4 to get the next register address.

2. Position the CGRA peripherals on the slave port of the Cluster Interconnect to interact with the heterogeneous cluster peripherals.

    - For CGRA events, position number 9 is reserved, which corresponds to address 0x10202400 in the memory map in Figure 6.4. This design choice resulted in CGRA COMMAND_REGISTER address to 0x10202400 and the CGRA STATUS_REGISTER address to 0x10202404. These registers are used to control and synchronize CGRA events using the API in Table 6.1. Depending on the number of tasks that can be serialized in the CGRA at once, the value of CGRA STATUS_REGISTER is incremented by 4 to get the next register address.

### 6.1.4 Workload Synchronization between CGRA and RI5CY sub-systems

In this section, workload synchronization between the CGRA sub-system and the RI5CY sub-system is explained using the help of an example program. Code 6.1 presents the sample C code:

```
int i, k;
float A[10], B[10], C[10];
float OUT[10], OUT_ORDERED[10];

// ** execute in CGRA **
for (i = 0; i < 10; i++){
        // OUT[i] = (A[i] * B[i]) + C[i];
        OUT[i] = fadd8 ( fmul8(A[i] , B[i]) , C[i] ); }

// ** execute in RI5CY **
for (k = 9; k <= 0; k--){
        OUT_ORDERED[k] = OUT[9-k]; }
```

Code 6.1: Sample C code showing workload split between CGRA and RI5CY sub-systems

The *for* loop with $i$ is executed on CGRA sub-system and the *for* loop with $k$ is executed on RI5CY sub-system. First, the C code part that is executed on the CGRA sub-system is compiled, and the corresponding bit-stream is obtained. Code 6.2 presents the C function showing the workload synchronization between two sub-systems.

```
// ** inclusion of header files and definition of input/output data **

#define CGRA_execute 1
#define RI5CY_execute 1

static int cluster_entry()
{
    // ** RI5CY part is executed on CORE[0] **
    if (get_core_id() == 0)
    {
        if(CGRA_execute)
        {
            // ** Data is loading from L2 to TCDM **
            DATA_LOAD_L2toTCDM ();

            // ** Context is loading from L2 to GCM **
            CGRA_LOAD_CONTEXT_L2toGCM();

            // ** CGRA execution started **
            int id = 0;
            id = CGRA_START_EXECUTION();
            CGRA_CHECK_STATUS(id);
            CGRA_FREE_CGRA(id);

            // ** CGRA Execution completed **
        }

        if(RI5CY_execute)
        {
            // ** RI5CY execution started **
            int k;
            for (k = 9; k <= 0; k--)
            {
                OUT_ORDERED[k] = OUT[9-k];
            }
            // ** CPU Execution completed **
        }
    }
    return 1;
}
```

Code 6.2: C function showing workload synchronization between CGRA and RI5CY sub-systems

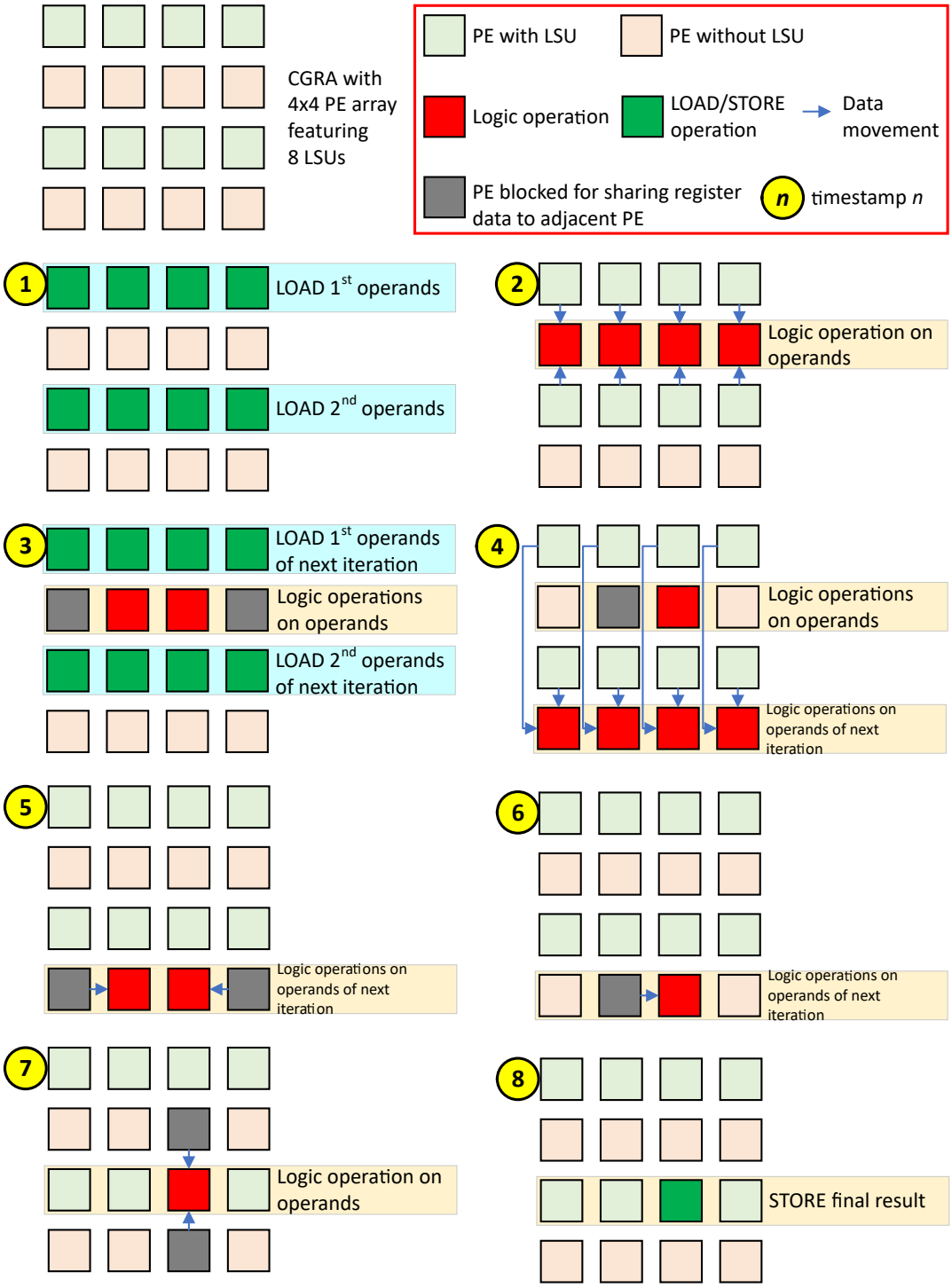



Figure 6.5: Manual Mapping Approach 1

### 6.1.5 Manual Mapping approaches

The proposed compiler for CGRA is unable to fully exploit the available resources on CGRA during the mapping step, resulting in low PE utilization. So, to reach a higher PE utilization, mapping of the applications used for benchmarking heterogeneous cluster

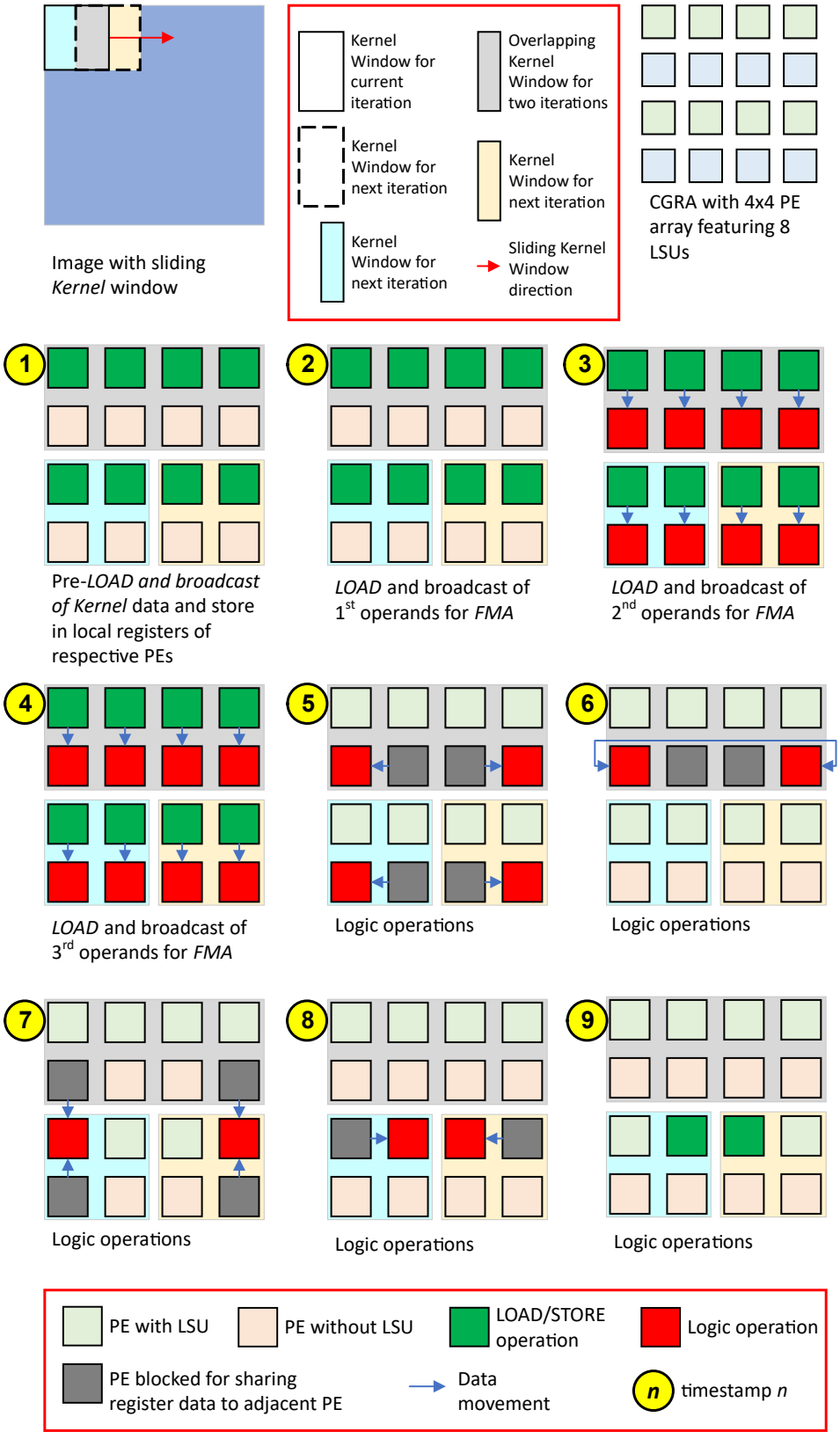



Figure 6.6: Manual Mapping Approach 2 (used in *k-means*)

where mainly CGRA is executing are performed manually. There are particularly two

manual mapping approaches employed. These are:

1. **Manual mapping Approach 1**: Figure 6.5 shows this approach and is used for the applications where,
   - majority of data required for computation are loaded during each loop iteration, and
   - majority of computed results are stored back to memory (i.e., TCDM) either before or at the end of each loop iteration.
2. **Manual mapping Approach 2**: Figure 6.6 shows this approach and is used for the applications where,
   - majority of data required for computation are pre-loaded and used over several loop iterations, and
   - majority of computed results that can be temporarily stored in the local registers of CGRA to avoid repeated storing and loading of same data over several loop iterations.

### Manual Mapping Approach 1

Figure 6.5 shows the Manual Mapping Approach 1. To explain this approach, Code 6.3 presents a simple multiply-accumulate kernel to explain each step in this mapping approach.

```
int j;
float A[8], B[8];  // ** Input Data **
float OUT;  // ** Output Data **

float tempA = 0.0;
float tempB = 0.0;

// ** loop unrolled **
// for (j = 0; j < 8; j+=8)
// {
    float temp0 = A[j+0] * B[j+0];
    float temp1 = A[j+1] * B[j+1];
    float temp2 = A[j+2] * B[j+2];
    float temp3 = A[j+3] * B[j+3];

    tempA = temp0 + temp1 + temp2 + temp3;

    float temp4 = A[j+4] * B[j+4];
    float temp5 = A[j+5] * B[j+5];
    float temp6 = A[j+6] * B[j+6];
    float temp7 = A[j+7] * B[j+7];
```

```
    tempB = temp4 + temp5 + temp6 + temp7;
// }

OUT = tempA + tempB;
```

Code 6.3: A simple multiply-accumulate C code

Following are the steps considered for mapping the kernel onto a 4x4 PE array with 8 LSUs.

1. At timestamp #1, addresses are calculated in the Flexible-AGU of PEs with LSU. A[0], A[1], A[2], A[3] are loaded in first row, and B[0], B[1], B[2], B[3] are loaded in third row, respectively.

2. At timestamp #2, PEs in the second row perform logic operations, i.e., multiplication on the fetched data. PEs in the first and third rows broadcast the loaded data to their adjacent PEs.

3. At timestamp #3, A[4], A[5], A[6], A[7] are loaded in the first row, and B[4], B[5], B[6], B[7] are loaded in the third row, respectively. While data is being loaded in the first and third rows, PEs in the second row performs the accumulation of calculated results.

4. At timestamp #4, PEs in the fourth row perform logic operations, i.e., multiplication on the fetched data, and PEs in the second row perform accumulation of the calculated results.

5. At timestamp #5, PEs in the fourth row performs accumulation.

6. At timestamp #6, PEs in the fourth row performs accumulation.

7. At timestamp #7, PE in the third row performs the final addition of the accumulated results from the third and fourth rows.

8. At timestamp #8, finally accumulated result is stored back in the memory.

### Manual Mapping Approach 2

Figure 6.6 shows the Manual Mapping Approach 2. To explain this approach, Code 6.4 presents a sample C code to explain each step in this mapping approach.

```
int i0, i, j;
float Image[200][200], A[200], B[9]; // ** Input Data **
float Image_Out[200][200]; // ** Output Data **
```

```
for (i0 = 0; i0 < 200; i0++)
{
    // ** loop unrolled **
    for (i = 0; i < 200; i+=2)
    {
        float tempA = 0.0;
        float tempB = 0.0;

        float temp0 = B[0] + A[i0+0] * Image[i0][i];
        float temp1 = B[1] + A[i0+1] * Image[i0][i];
        float temp2 = B[2] + A[i0+2] * Image[i0][i];
        float temp3 = B[3] + A[i0+3] * Image[i0][i];

        tempA = temp0 + temp1 + temp2 + temp3;

        float temp4 = B[4] + A[i0+4] * Image[i0][i+1];
        float temp5 = B[5] + A[i0+5] * Image[i0][i+1];
        float temp6 = B[6] + A[i0+6] * Image[i0][i+1];
        float temp7 = B[7] + A[i0+7] * Image[i0][i+1];

        tempB = temp4 + temp5 + temp6 + temp7;

        Image_Out[i0][i+0] = tempA * B[8];
        Image_Out[i0][i+1] = tempB * B[8];
    }
}
```

Code 6.4: A sample C code. Text highlights correspond to the color coding in Figure 6.6. Pre-loaded Data, Distinct Data 1, Distinct Data 2, Common Data

Following are the step considered for mapping the kernel onto a 4x4 PE array with 8 LSUs.

1. First, the code is analyzed to determine the common and distinct calculations to be performed over multiple loop iterations. For example, the sliding kernel window in Figure 6.6 has both common and distinct calculations over two loop iterations.

2. At timestamp #1, elements of the B[8] array are pre-loaded into their respective PEs. This step is done before starting execution of the loops in the kernel, as these values are stored in the local registers of the PEs and used over several loop iterations.

3. At timestamp #2, elements of the Image[200][200] array are loaded in the first row and the third row, respectively.

4. At timestamp #3, elements of the A[200] array are loaded in the first row and the third row, respectively. PEs in the second row begins logic operations, i.e., fused-multiply-accumulate (FMA).

5. At timestamp #4, if B[8] array data is not present in PEs to perform FMA, then those data are loaded in adjacent PE in either first or third rows and broadcast to the PEs second and fourth rows, respectively. Otherwise, the second and fourth rows perform FMA operations with the preloaded data.

6. At timestamp #5, PEs in the second and fourth rows perform accumulation.

7. At timestamp #6, PEs in the second row perform accumulation.

8. At timestamp #7, PEs in the third row perform accumulation.

9. At timestamp #8, PEs in the third row perform multiplication.

10. At timestamp #9, finally, the results are stored back in the memory.

## 6.2 Summary and Concluding Remarks

This chapter presents a heterogeneous cluster featuring a *transprecision* computing-based CGRA sub-system and a multi-core RI5CY sub-system. Particularly, highlighting the (1) integration of CGRA into the PULP-Cluster, (2) clock-gating between CGRA and RI5CY sub-systems, (3) software infrastructure, (4) PULP SoC memory map, (5) reserving addresses for CGRA events into the PULP-Cluster components, (6) workload synchronization between two sub-systems, and (7) manual mapping approaches to map applications onto CGRA. Next chapter 7 presents an extensive exploration of the design space and evaluation of proposed CGRA and the heterogeneous cluster using a set of real-world applications which implements the fundamental algorithms for the applications used in two domains relevant for ultra-low-power systems, i.e., near sensor computing and embedded machine learning.

# Chapter 7

# Experimental Framework and Performance Evaluation

## Contents



This chapter analyses the implementation of proposed CGRA, mainly, (1) Design optimization 1: IEEE 754-2008 Standard compliant 4x2 PE Array (See section 4.1.1), (2) Design optimization 2: Transprecision FP compliant 4x2 PE Array (See section 4.1.2), and (3) Design optimization 4: Transprecision FP compliant 4x4 PE Array (See section 4.1.4), are presented. A standard Power, Performance, and Area (PPA) analysis and other experiments are performed to evaluate proposed CGRA featuring *transprecision* computing, and the obtained results are compared with different architectures supporting either *transprecision* FP or SoA IEEE 754-2008 standard FP computations.

The evaluation of the CGRA designs is performed and presented in two parts.

- In the first part, Design optimization 2: Transprecision FP compliant 4x2 PE Array is in focus and compared with (1) Design optimization 1: IEEE 754-2008 Standard compliant 4x2 PE Array, to provide an overview of comparing *transprecision* computing-based CGRA design with SoA FP computing-based CGRA, (2) a highly optimized RISC-V based ASIP extended with custom instructions for accelerating DSP applications, namely RI5CY [48], supporting *tranprecision* computing, and (3) RI5CY supporting SoA FP computing.

- In the second part, an extensive exploration of the design space with different configurations to pull off the highest performances from the heterogeneous cluster containing both CGRA sub-system and multi-core RI5CY sub-system (See chapter 6), in terms of latency, power, and area is presented.

## Contribution and Outline of the chapter

The outline of this chapter is as follows:

1. Analyses of implementation of the proposed CGRA, particularly,

    - description of the experimental setup, including a brief description of the applications used for PPA analysis and the different architectures used for comparison in this chapter,

    - evaluation of accuracy deviation of *binary16alt* with respect to SoA FP datatype i.e., IEEE 745-2008 *binary32*,

    - implementation in 28nm process node and total cell area ($\mu m^2$) comparing the proposed GCRA with different architectures,

    - latency performance (cycles) and evaluation of different architectures executing kernels using different FP datatype,

    - energy consumption ($\mu$J) and comparison with different architectures, and

    - energy-efficiency (Million Operations Per Second Per mW or MOPS/mW) of different architectures.

2. Analyses of implementation of the heterogeneous cluster, particularly,

    - setup of the experiments, including a brief description of the applications used for Power, Performance, and Area (PPA) analysis,

    - implementation in 22nm process node (design sign-off) and total area ($\mu m^2$) breakdown of the heterogeneous cluster,

- latency performance (cycles) and evaluation of different configurations while executing kernels,
- energy consumption ($\mu$J) of the sub-systems and heterogeneous cluster, and
- exposition of the correlation between PE utilization of CGRA and variations in the power consumption.

Finally, a summary of the results and concluding remarks are provided at the end of this chapter.

## 7.1 Analyses of implementation of the proposed CGRA

### 7.1.1 Evaluation Methodology

A set of applications, which implements the fundamental algorithms for the applications used in two domains relevant for ultra-low-power systems, i.e., near sensor computing and embedded machine learning, are chosen for performing the experiments. Below is a brief description of the selected applications.

- PCA performs Principal Component Analysis, and this algorithm is used for seizure detection, which is used for a wide range of applications for processing Electroencephalography (EEG) signals. PCA itself consists of 5 sequentially executed kernels, namely, (1) Mean Covariance, (2) Householder, (3) Accumulate, (4) Diagonalize, and (5) Principal Component. These kernels exhaustively perform FP computations and also require a wide dynamic range of FP data representation. Particularly, *binary16alt* and IEEE 754-2008 *binary32* FP datatype are used.
- CONV implements a 5x5 convolution kernel, and this algorithm is used for image and audio processing applications. Particularly, *binary8* FP datatype is used.
- DWT computes the Discrete Wavelet Transform, and this algorithm is used for applications performing Electrocardiography (ECG) analysis. Particularly, *binary8* FP datatype is used.
- SVM is the prediction stage of a Support Vector Machine, and this algorithm is used as a classifier for predicting traffic data, ECG, etc. Particularly, *binary8* FP datatype is used.

Table 7.1 shows the complexity of these kernels in terms of (1) total number of operations executed, (2) highest loop iteration, and (3) input data size (in bits).

| Kernel | Operations executed | Highest loop iteration | Input Data size (bits) |
|---|---|---|---|
| mean_covariance | 397,348 | 47,104 | 94,208 |
| Householder | 35,632 | 1,360 | 9,216 |
| Accumulate | 106,298 | 1,240 | 8,704 |
| Diagonalize | 74,987 | 2,368 | 9,216 |
| PC | 168,738 | 11,776 | 102,400 |
| CONV | 766,728 | 4,096 | 131,072 |
| DWT | 39,456 | 448 | 16,384 |
| SVM | 15,630 | 896 | 72,000 |

Table 7.1: Complexity of Kernels

Here, Design 2 (See section 4.1.2) is referred to as TRANSPIRE. TRANSPIRE is compared against 3 different architectures featuring *binary16alt*, *binary8*, and IEEE 754-2008 *binary32*. Below is a brief description of the three selected architectures.

- RI5CY_FPU [48] is a single-core in-order 4-stage RISC-V CPU with support for IEEE 745-2008 *binary32* FP datatype. This architecture will provide a comparison of TRANSPIRE with SoA architecture.

- RI5CY_SFU [61] is a single-core in-order 4-stage RISC-V CPU with enhanced ISA supporting SIMD-style vectorization and supports *binary16alt* and *binary8* FP datatype. This architecture will provide a comparison of TRANSPIRE with SoA architecture with similar features.

- TRANSPIRE_FPU (See section 4.1.1) is version of TRANSPIRE featuring IEEE 754-2008 *binary32* FP datatype. This architecture will compare TRANSPIRE with the same architecture featuring SoA IEEE 754-2008 *binary32* with no degradation on quality of results. TRANSPIRE_FPU features the same FP operators as TRANSPIRE.

All the configurations of different architectures have been carefully chosen to ensure a fair comparison. Mainly, RI5CY [48] is an in-order 4-stage RISC-V CPU that supports SIMD extensions, custom instructions, and misaligned load support. Due to these features, the bandwidth requirements for data memory are extensively reduced, and computational efficiency increases. RI5CY is highly optimized for DSP applications, making it a good candidate for a fair comparison with TRANSPIRE.

### 7.1.2 Quality of Results

One of the main concerns of using an FP format with fewer precision bits is the accuracy degradation w.r.t. SoA FP datatype. Table 7.2 shows the accuracy deviation of the results calculated using *binary16alt* and *binary8* w.r.t. IEEE 754-2008 *binary32* and

| Kernel | Average deviation (%) | Data-type |
|---|---|---|
| mean_covariance | 4.80 | *binary16alt* |
| Householder | 0.33 | *binary16alt* |
| Accumulate | 9.03 | *binary16alt* |
| Diagonalize | 5.49 | *binary16alt* |
| PC | 1.54 | *binary16alt* |
| CONV | 2.32 | *binary8* |
| DWT | 6.98 | *binary8* |
| SVM | 7.11 | *binary8* |

Table 7.2: Accuracy Performance

*binary16* respectively. It must be noted that *binary16alt* and IEEE 754-2008 *binary32* have same dynamic data range and *binary8* and IEEE 754-2008 *binary16* have same dynamic data range. So, the results of these pairs are compared with each other to ensure a fair comparison. In the Table 7.2, it can be observed that the accuracy deviation is always below 10%. Particularly, Accumulate kernel shows a 9.03% accuracy loss due to extensive computations on sub-normal FP numbers (i.e., FP numbers between -1.0 to +1.0). The least accuracy loss is observed in the Householder kernel due to fewer operations involving sub-normal FP numbers.

### 7.1.3 Implementation Results

All four architecture designs have been synthesized (for gate-level simulation) at 28nm Ultra-Thin Body and Buried oxide Fully Depleted Silicon On Insulator (UTBB-FD-SOI) [115] process node. Parameters used for synthesis are configured to test setup time violation (because zero wire model is used during logic synthesis), and they are as follows:

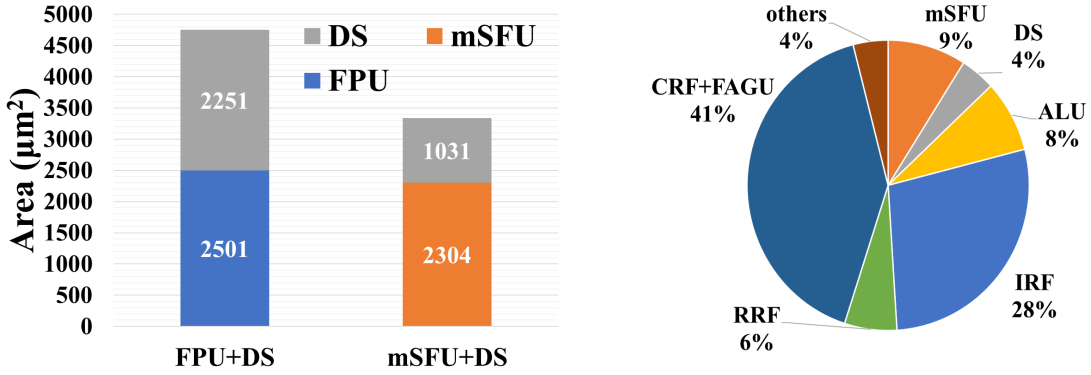
- 50 Hz frequency
- 0.6V operating voltage
- worst-case analysis corner (i.e., slow NMOS, slow PMOS)
- 125°C temperature
- low power low V$t$ transistors

Configuration of TRANSPIRE is as follows:

- Context Memory is sized at 4KiB, to fit two full-size configuration data (i.e., instructions and constants)

| | TRANSPIRE | RI5CY SFU | TRANSPIRE FPU | RI5CY FPU |
|---|---|---|---|---|
| **DMA Controller** | 593 | 4 KiB Instruction Cache | 593 | 4 KiB Instruction Cache |
| **Interconnect** | 6,273 | | 6,273 | |
| **Context memory** | 9,345 | | 9,345 | |
| **TCDM** | 65,164 | | 65,164 | |
| **PE Array** | 186,407 | | 174,230 | |
| **Total** | 267,784 | 213,371 | 255,605 | 185,812 |

Table 7.3: Total cell area ($\mu m^2$) breakdown and comparison




(a) Area comparison of FPU, mSFU, and DS

(b) Area breakdown of PE

Figure 7.1: Total cell area ($\mu m^2$) breakdown and comparison

- TCDM is sized at 32KiB
- 4x2 PE array size
- Instruction Register File (IRF) is sized at 21x64-bit, featured in each PE
- Constant Register File (CRF) is sized at 20x32-bit, featured in each PE
- Regular Register File (RRF) is sized at 7x32-bit, featured in each PE
- OutPut Register (OPR) is sized at 1x32-bit, featured in each PE

TRANSPIRE_FPU shares the same configuration as TRANSPIRE, with mSFU and DS unit replaced with IEEE 754-2008 *binary32* compliant FPU and DS unit, respectively. Both RI5CY_SFU and RI5CY_FPU also feature a 4KiB instruction cache and 32KiB data memory (i.e., TCDM) for the sake of fair comparison.

Table 7.3 shows the total cell area ($\mu m^2$) breakdown analysis of all four architectures (synthesized in 28nm process node). It can be observed that TRANSPIRE has 1.25× larger total cell area than RI5CY_SFU, 1.05× larger total cell area than TRANSPIRE_FPU, and 1.44× larger total cell area than RI5CY_FPU. RI5CY sub-systems have a comparable total cell area ($\mu m^2$) w.r.t. TRANSPIRE sub-system (See Table 7.3),

| Kernel | PE Utilization (%) |
|---|---|
| **PCA** | 72% |
| **CONV** | 63% |
| **DWT** | 87.5% |
| **SVM** | 47% |

Table 7.4: Average PE Utilization of kernels

| Kernel | TRANSPIRE binary8 (cycles) | RI5CY_SFU binary8 (cycles) | Gain |
|---|---|---|---|
| **CONV** | 268,179 | 1 455,097 | 5.43× |
| **DWT** | 11,140 | 16,912 | 1.52× |
| **SVM** | 11,408 | 114,747 | 10.06× |

Table 7.5: Latency Performance (cycles) of kernels (*binary8*)

so single-core RI5CY featuring either SFU or FPU is used for comparison in this chapter.

Figure 7.1b shows the total cell area ($\mu m^2$) breakdown of a PE featuring mSFU and DS unit, and it can be observed that mSFU takes 9% and DS unit takes 4% of total cell area ($\mu m^2$) of a PE. This allowed an mSFU to be included in each PE in TRANSPIRE. Figure 7.1a represents the total cell area ($\mu m^2$) comparison of *transprecision* computing compliant mSFU and DS with SoA IEEE 754-2008 *binary32* compliant FPU and DS unit. It can be observed that the later combination has 1.42× larger total cell area ($\mu m^2$) than *transprecision* compliant units. Particularly, FPU is 1.09× larger than mSFU and IEEE 754-2008 *binary32* compliant DS unit is 2.18× larger than *binary16alt* compliant DS unit.

### 7.1.4 Latency Performance

This section presents and compares the latency performance of each architecture executing the set of kernels discussed in section 7.1.1. The latency performance is shown in the number of clock cycles consumed in executing a kernel.

Table 7.4 shows the average PE utilization of TRANSPIRE for each kernel. Same PE utilization numbers are also applicable for TRANSPIRE_FPU, as both architectures use the same compilation flow. TRANSPIRE features a 32-bit wide datapath, so TRANSPIRE can support SIMD lanes=2 with 16-bit FP datatype (i.e., *binary16alt*) and SIMD lanes=4 with 8bit FP datatype (i.e., *binary8*) but TRANSPIRE_FPU only uses 32-bit FP datatype (i.e., IEEE 754-2008 *binary32*). Thus, fewer cycles are consumed for executing the same set of kernels in TRANSPIRE w.r.t. TRANSPIRE_FPU. Figure 7.2 shows the latency performance (cycles) of four architectures executing PCA kernels featuring different FP formats.

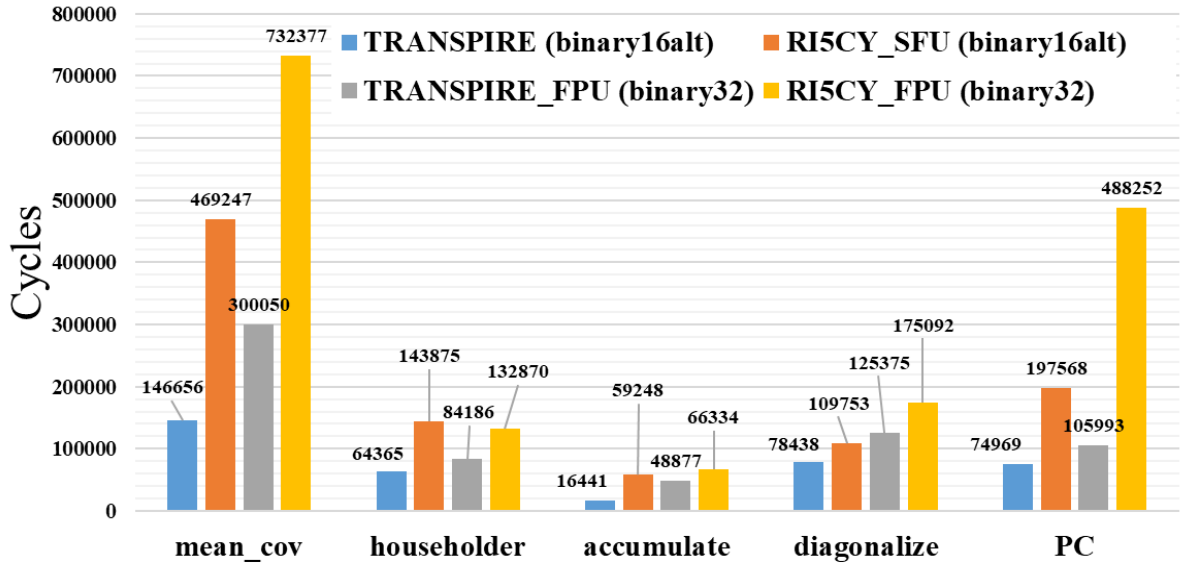



Figure 7.2: Latency performance (cycles) of PCA kernels (*binary16alt* and *binary32*)

| Kernel | TRANSPIRE binary8 ($\mu$J) | RI5CY_SFU binary8 ($\mu$J) | Gain |
|---|---|---|---|
| **CONV** | 3.036 | 21.506 | 7.08× |
| **DWT** | 0.124 | 0.256 | 2.07× |
| **SVM** | 0.123 | 1.588 | 12.91× |

Table 7.6: Energy Consumption ($\mu$J) of kernels (*binary8*)

At 100% PE utilization, TRANSPIRE can execute 8x2 parallel 16-bit FP operations and 8x4 parallel 8-bit FP operations, while RI5CY_SFU can execute 1x2 parallel 16-bit FP operations and 1x4 parallel FP operations. It must be noted that RI5CY cores can efficiently execute memory operations (i.e., LOAD and STORE) by exploiting its in-order 4-stage pipeline architecture, which gives RI5CY cores an architectural advantage to perform well in kernels demanding extensive memory operations w.r.t. TRANSPIRE. It can be observed that RI5CY_SFU is unable to surpass the latency performance of TRANSPIRE with low average PE Utilization. Moreover, TRANSPIRE (while executing *binary8* kernels) outperforms RI5CY_SFU (while executing *binary8* kernels) by 10.06×. Table 7.5 shows latency performance (cycles) TRANSPIRE and RI5CY_SFU executing kernels featuring *binary8* FP format. Unlike RI5CY_SFU, RI5CY_FPU neither supports SIMD nor surpasses the latency performance of TRANSPIRE with the average PE utilization(See Figure 7.2).

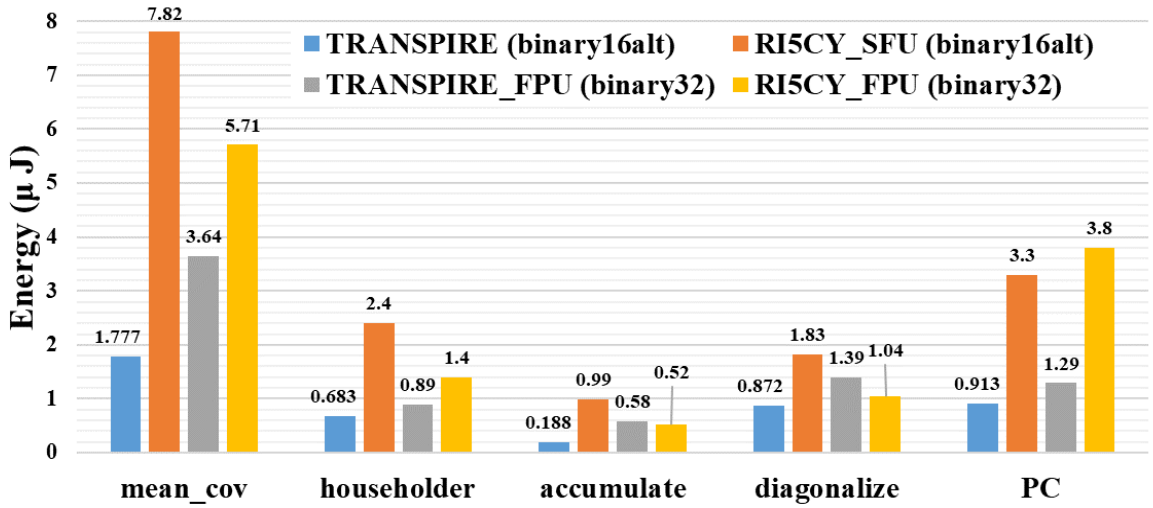



Figure 7.3: Energy consumption ($\mu$J) of PCA kernels (*binary16alt* and *binary32*)

### 7.1.5 Energy Consumption

Total cell area ($\mu m^2$) of TRANSPIRE sub-system is 1.25× and 1.44× larger than RI5CY_SFU and RI5CY_FPU sub-systems, respectively. However, implementing power-saving techniques in TRANSPIRE lowered the power consumption in TRANSPIRE, such that the obtained results are comparable with RI5CY sub-systems. Energy consumption ($\mu$J) is directly proportional to latency (seconds) which can be deduced from the following equation:

$$Total\_Energy\ (\mu J) \quad = (avg\_Power\ (mW)\ *\ cycles)\ *\ 1\_clk\_cycle\ (ns) * 10^{-6} \tag{7.1}$$

where,

$cycles$ = total number of cycles executed on RI5CY/ TRANSPIRE

$avg\_Power$ = average Power consumption (mW) of RI5CY/ TRANSPIRE sub-system

Using a proper clock-gating scheme, a better energy-saving scheme can be implemented; the previous chapter discusses such implementation (See chapter 6)

Figure 7.3 shows the total energy consumption ($\mu$J) of each sub-system while executing 16-bit or 32-bit FP datatype. Latency performance (cycles) of TRANSPIRE (See Figure 7.2) is better than the rest of the three architectures in comparison. This resulted in

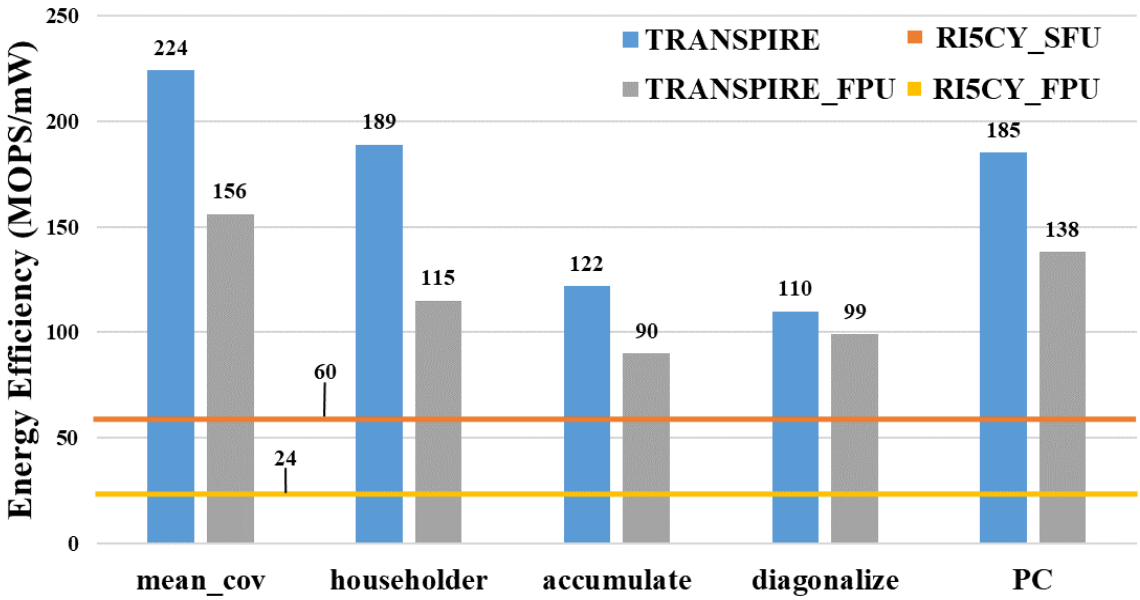



Figure 7.4: Energy-Efficiency (MOPS/mW) comparison of 4 architectures

the least energy consumption ($\mu$J) of TRANSPIRE w.r.t. RI5CY_SFU, RI5CY_FPU, and TRANSPIRE_FPU (See Equation 7.1). Similarly, TRANSPIRE (while executing *binary8* kernels) consumes less energy than RI5CY_SFU (while executing *binary8* kernels) by 12.91× (See Table 7.6).

Particularly, TRANSPIRE_FPU performed non-vectored FP operations on 32-bit operands and executed more instructions, resulting in more energy consumption. Due to the complex architecture of the RI5CY_SFU core, its energy consumption is higher than TRANSPIRE. Lastly, RI5CY_FPU performs non-vectored FP operations on 32-bit operands and features a complex core w.r.t. TRANSPIRE, which consumes more energy than TRANSPIRE.

### 7.1.6 Energy-Efficiency

Figure 7.4 shows the energy efficiency (MOPS/mW) of four architectures while executing 16-bit or 32-bit FP operations. IEEE 754-2008 *binary32* is comparable with *binary16alt*, PCA kernels are considered for this comparison. TRANSPIRE outperforms all architectures used in the comparison due to simple and efficient architecture. Particularly, Householder and Diagonalize kernels have low performance in TRANSPIRE w.r.t. other kernels because these kernels are high control intensive due to complex control flow constructs and hence, causes low ILP. TRANSPIRE reaches a maximum of 224 MOPS/mW, and TRANSPIRE_FPU reaches a maximum of 156 MOPS/mW, while RI5CY_SFU shows 60 MOPS/mW and RI5CY_FPU shows 24 MOPS/mW. In RI5CY cores, the energy efficiency is constant because these cores can efficiently execute the

complete PCA application at once.

## 7.2 Analyses of implementation of the heterogeneous cluster

### 7.2.1 Evaluation Methodology

#### Applications

A set of applications employed in different near-sensor processing application fields (i.e., audio, image, bio-signals, and embedded ML) is considered for conducting the experiments. These applications include different execution patterns and stress the flexibility of CGRA. Table 7.7 shows the complexity of these kernels for CGRA in terms of (1) number of operations executed, (2) highest loop iteration (after optimization), (3) size of the global context data (Bytes), and (4) input data size (Bytes). These kernels are executing 16-bit *binary16alt* operations on both sub-systems (i.e., CGRA and RI5CY). Following is a brief description of the applications used to conduct experiments, along with some of their real-life usage:

- **FFT**: Implements radix-2 Fast Fourier Transform and is widely used in voice recognition, telecommunication, medical imaging, and Magnetic Resonance Imaging (MRI) as well as Computerized Tomography (CT) scan, optics, military, and geology.

- **FIR**: Implements Finite Impulse Response and is used where linearity has to be ensured. FIR is preferred for its adaptive design, and some of its applications are linear predicting coding, linear interpolation, spatial beamforming (sensor arrays), adaptive filters (ECG), speech analysis and modeling, averaging filters (counter noisy signal), and multi-rate signal processing (Digital-Analog Converters).

- **IIR**: Implements Infinite Impulse Response and is one of the two primary types of filters used in audio and digital signal processing.

- **k-means**: Implements k-means clustering and is used in machine learning for quantization and classification.

- **matMUL**: Implements matrix multiplication is used to describe any linear system for equations (modern physics).

- **CONV**: Implements 5x5 convolution and is widely used in image and audio processing applications.

| Kernel | #Operations Executed | Highest loop iteration (after optimization) | Global Context Data size (Bytes) | Input Data size (Bytes) |
|---|---|---|---|---|
| FFT | 341,586 | 1,792 | 8,688 | 6,144 |
| FIR | 9,222 | 32 | 3,720 | 528 |
| IIR | 171,984 | 416 | 4,752 | 5,056 |
| k-means | 1,202,920 | 15,400 | 8,954 | 7,488 |
| matMUL | 1,514,676 | 4,096 | 5,520 | 16,384 |
| CONV | 352,080 | 1,024 | 5,712 | 4,196 |
| DWT | 77,902 | 384 | 3,840 | 1,040 |
| SVM | 147,562 | 308 | 10,992 | 11,704 |
| CCA | 460,199 | 900 | 41,856 | 3,800 |

Table 7.7: Kernel Complexity for CGRA

- **DWT**: Implements Discrete Wavelet Transform and is mainly used in applications for ECG analysis.
- **SVM**: Implements the prediction stage of Support Vector Machine and is widely used as a classifier for predicting ECG, traffic data, etc.
- **CCA**: Implements Canonical Correlation Analysis and is a multivariate statistical ordination analysis (Brain-Computer Interfaces).

### Heterogeneous Cluster

Figure 6.1 shows the organization of CGRA and RI5CY in the heterogeneous cluster. Both architectures load their respective context data from a 512KiB L2 memory through Cluster Bus. Both architectures share data using a 64KiB TCDM, implemented using 16 1024x32-bit SRAM banks. Peripherals on the cluster, particularly, Event Unit (controls the cluster event generation by issuing CGRA and RI5CY, clock-gating between CGRA and RI5CY, and DMA events), are shared by both architectures. Below are a brief description of both sub-systems.

- **CGRA** is a 4x4 PE array connected through a simple mesh torus interconnect network (See section 4.1.4). Each PE features 1*32-bit integer as well as 2*16-bit *binary16alt* FP datatype operators. Each PE has a 21x32-bit instruction register file, 20x32-bit constant register file, and 8x32-bit local register file. CGRA sub-system features an 8KiB Context Memory implemented using 2 1024x32-bit SRAM banks to hold 2 maxed size configuration data of 4KiB each (16*[21*64 + 64 + 20*32] = 4KiB, here extra 16*64-bit hold 16 configuration lines used to identify the PE, its number of instructions, and its number of constants).
- **RI5CY** is a 4-stage in-order RISC-V CPU with an enhanced ISA supporting

SIMD-style vectorization. Sub-system features a 4KiB shared instruction cache implemented using a standard cell (or latch) based approach. RI5CY 8-cores share an FPU cluster consisting of 4 SFU [61, 56] cores. Scheduling of FP operations is deterministic and based on the scheme described in [17].

### 7.2.2 Implementation Results

PPA analysis is considered for evaluating the performance of heterogeneous clusters for different configurations using CGRA and RI5CY sub-systems. Additionally, PPA analysis is also used for comparing CGRA and RI5CY at the sub-system level.

#### CAD Tools

All experiments are conducted using a post-synthesis netlist obtained after digital sign-off in *GLOBALFOUNDARIES* 22FDX® (22nm FD-SOI Technology) using a low-threshold 8-cell library at low voltage. Synopsys Design Compiler is used for logic synthesis, and place-and-route (PnR), Cadence Innovus, is used. Particularly, PnR is done using multi-mode-multi-corner analysis i.e., best-case (FFG, -40°C & +125°C, 0.72V), worst-case (SSG, -40°C & +125°C, 0.59V), and typical-case (TT, +25°C, 0.65V) constraints. Finally, Synopsys PrimeTime is used for power estimation using typical-case constraints, where the cluster is running at 200MHz frequency.

#### PULP Platform

PULP is a silicon-proven Parallel Ultra Low Power platform targeting ULP application domains [46]. An approximation for conducting experiments is to run the parts of kernels on the best performing architecture available on the heterogeneous cluster. Particularly, to measure performances of FFT and DWT on CGRA, a major part of the kernel is executed on CGRA, and parts consisting of only memory operations (i.e., LOAD/ STORE operations to arrange the final output data in memory) between TCDM and CGRA are executed on the RI5CY sub-system.

#### Area Results

Table 7.8 shows the total area ($\mu m^2$) breakdown of CGRA and RI5CY sub-system, and Figure 7.5 shows the post-PnR layout of the heterogeneous cluster. CGRA sub-system is 2.19× smaller than RI5CY sub-system, which indicates that CGRA can be included alongside RI5CY without bringing any major area-wise overhead to cluster.

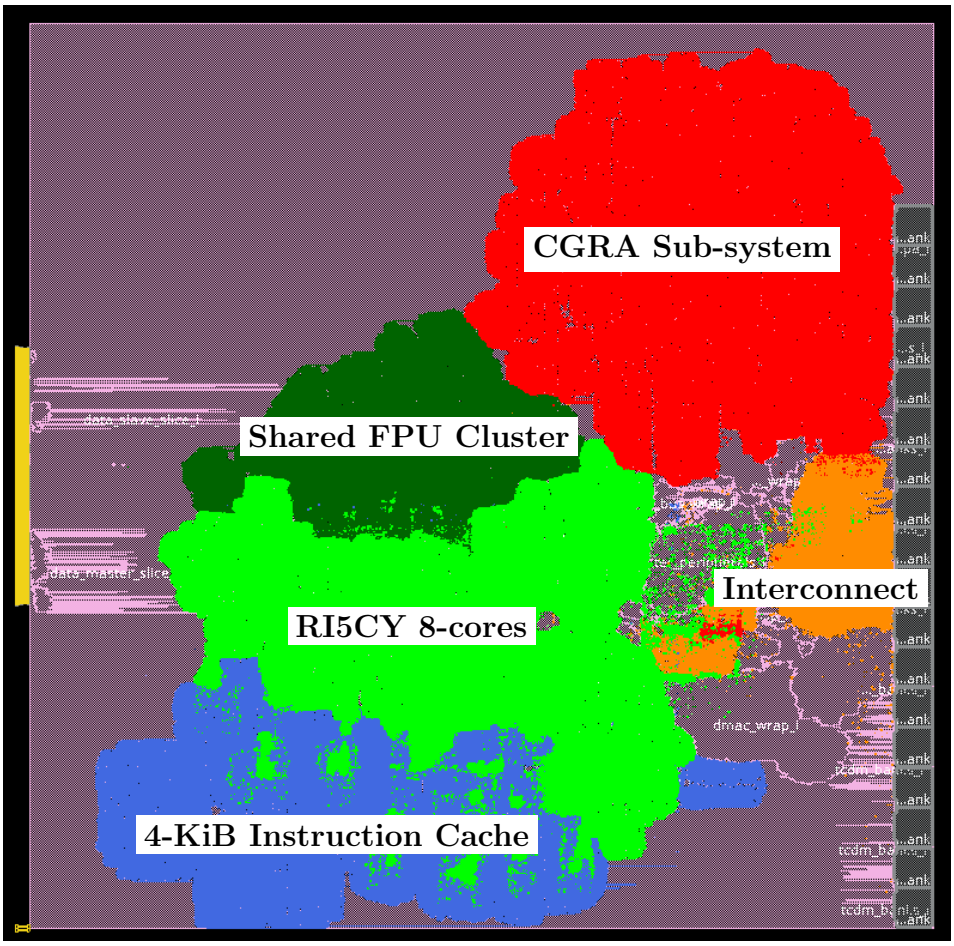



Figure 7.5: Post-PnR view of heterogeneous cluster

### 7.2.3 Latency Performance

#### CGRA vs. RI5CY 8-cores

Table 7.9 shows the latency performance of two architectures while executing different kernels. CGRA can efficiently exploit the ILP of k-means, FIR, IIR, CCA, and SVM than RI5CY due to highly parallelizable inner loops. However, FFT, matMUL, CONV, and DWT have low performance on CGRA due to (1) a high number of memory operations because CGRA requires at least 2 clock-cycles to perform a memory operation or (2) the inner loops have less parallelism resulting in low PE utilization in CGRA or (3) a kernel featuring high control intensive code constructs because all PEs in the CGRA is synchronized before every $JUMP$ instruction, which further restricts CGRA to exhibit low PE utilization.

| | **CGRA** ($\mu m^2$) | **RI5CY** ($\mu m^2$) |
|---|---|---|
| DMA Controller | 1,114 | FPU_Cluster (161,738) |
| CGRA Interconnect | 363 | + Interconnect (65,989) |
| Context Memory | 14,608 | + 4-KiB I$ Cache (234,026) |
| PE Array | 344,058 | + RI5CY 8-cores (326,605) |
| **Total** | **360,143** | **788,358** |

Table 7.8: Area comparison of CGRA sub-system and RI5CY sub-system

| **Kernel** | **CGRA (cycles)** | **RI5CY (cycles)** | **CGRA vs. RI5CY** |
|---|---|---|---|
| FFT | *34,349 | 24,753 | 0.72x |
| FIR | 1,393 | 3,806 | 2.73x |
| IIR | 16,348 | 30,214 | 1.85x |
| k-means | 102,586 | 431,144 | 4.20x |
| matMUL | 72,556 | 71,028 | 0.98x |
| CONV | 22,108 | 16,112 | 0.73x |
| DWT | **6,108 | 1,961 | 0.32x |
| SVM | 9,516 | 11,421 | 1.20x |
| CCA | 40,203 | 50,641 | 1.26x |

* 92% of the workload of FFT is executing on CGRA (31,579 cycles) while the rest 8% is executing on RI5CY (2,770 cycles).
** 87% of the workload of DWT is executing on CGRA (5,321 cycles) while the rest 13% is executing on RI5CY (787 cycles).

Table 7.9: Latency comparison of CGRA and 8-cores RI5CY

#### CGRA (with SIMD disabled) vs. RI5CY 8-cores (with SIMD disabled)

The number of processing units in the CGRA sub-system are [16 PEs * 2 SIMD lanes] = 32 processing units, and in the RI5CY sub-system are [8 cores * 2 SIMD lanes] = 16 processing units. CGRA sub-system has 2× more processing units than RI5CY sub-system, but CGRA is unable to (always) maintain 100% PE utilization while RI5CY can efficiently maintain a high utilization. So, to explore if the latency numbers for CGRA are influenced by any architectural advantage, particularly due to SIMD w.r.t. RI5CY, Table 7.10 shows the comparison between two architectures with SIMD disabled while executing 6 kernels (whose optimized non-SIMD version are available). It can be observed that the gain pattern in Table 7.10 is similar with Table 7.9. Thus, it is concluded that both architectures are efficiently utilizing the SIMD feature.

| Kernel | CGRA (cycles) | RI5CY (cycles) | CGRA vs. RI5CY |
|---|---|---|---|
| FFT | *63,612 | 41,377 | 0.65x |
| FIR | 1,961 | 4,190 | 2.14x |
| IIR | 31,752 | 47,772 | 1.51x |
| matMUL | 243,340 | 150,733 | 0.62x |
| CONV | 41,972 | 21,802 | 0.52x |
| DWT | **11,607 | 3,733 | 0.32x |

* 92% of the workload of FFT is executing on CGRA (60,833 cycles) while the rest 8% is executing on RI5CY (2,779 cycles).
** 87% of the workload of DWT is executing on CGRA (10,277 cycles) while the rest 13% is executing on RI5CY (1,330 cycles).

Table 7.10: Latency comparison of CGRA and 8-cores RI5CY. Both architectures are executing with SIMD disabled.

### 7.2.4 Energy Consumption

The gate count in a module is directly proportional to its power consumption (i.e., internal power, switching power, and leakage power), thus, its energy consumption. Even though the total area of the CGRA sub-system is 2.19× less than the RI5CY sub-system, the RI5CY sub-system can efficiently clock-gate 98% of core registers [91]. To further explore the energy consumption of both architectures and the effects on the performance of PULP-Cluster with the integration of CGRA, three different experiments are conducted, particularly, (1) comparison of energy consumption between two sub-systems, (2) comparison of energy consumption of heterogeneous cluster for different configurations, and (3) if the inclusion of CGRA does lower the energy-efficiency of PULP-Cluster due to added leakage current by CGRA sub-system when not active.

#### CGRA vs. RI5CY 8-cores

Table 7.11 shows the comparison of energy consumption of two sub-systems. Due to the simple design of PEs in CGRA and the efficient clock gating scheme, the energy consumption of CGRA is less w.r.t RI5CY. Even, latency-wise less performing kernels on CGRA exhibits similar or lesser energy consumption w.r.t. RI5CY. Particularly, CGRA consumes 1.49× less energy w.r.t. RI5CY while executing matMUL (latency performance of CGRA is 0.98× w.r.t. RI5CY) and 0.96× energy w.r.t. RI5CY while executing DWT (latency performance of CGRA is 032× w.r.t. RI5CY).

| **Kernel** | **CGRA ($\mu$J)** | **RI5CY ($\mu$J)** | **CGRA vs. RI5CY** |
|---|---|---|---|
| FFT | *0.8824880 | 0.6703261 | 0.76x |
| FIR | 0.0286958 | 0.1254629 | 4.38x |
| IIR | 0.3636613 | 0.9959894 | 2.74x |
| k-means | 2.5856801 | 20.1751680 | 7.80x |
| matMUL | 1.5936925 | 2.3779464 | 1.49x |
| CONV | 0.4691318 | 0.4491526 | 0.96x |
| DWT | **0.1387076 | 0.0889372 | 0.86x |
| SVM | 0.1901297 | 0.3007069 | 1.58x |
| CCA | 1.0011935 | 1.3878727 | 1.39x |

* 92% of the workload of FFT is executing on CGRA (0.8074750 $\mu$) while the rest 8% is executing on RI5CY (0.0750130 $\mu$).
** 87% of the workload of DWT is executing on CGRA (0.1030146 $\mu$) while the rest 13% is executing on RI5CY (0.0356930 $\mu$).

Table 7.11: Energy consumption comparison of CGRA sub-system and 8-cores RI5CY sub-system

| **Kernel** | **CGRA ($\mu$J)** | **RI5CY ($\mu$J)** | **CGRA vs. RI5CY** |
|---|---|---|---|
| FFT | *1.625019 | 0.897049 | 0.55x |
| FIR | 0.061027 | 0.162212 | 2.66x |
| IIR | 0.749065 | 1.287721 | 1.72x |
| k-means | 4.963111 | 26.670570 | 5.38x |
| matMUL | 3.185208 | 3.051363 | 0.96x |
| CONV | 0.967888 | 0.614512 | 0.64x |
| DWT | **0.266665 | 0.112640 | 0.42x |
| SVM | 0.400148 | 0.417095 | 1.04x |
| CCA | 1.896455 | 1.714678 | 0.91x |

* 92% of the workload of FFT is executing on CGRA (1.524634 $\mu$) while the rest 8% is executing on RI5CY (0.100385 $\mu$).
** 87% of the workload of DWT is executing on CGRA (0.221460 $\mu$) while the rest 13% is executing on RI5CY (0.045205 $\mu$).

Table 7.12: Energy consumption of heterogeneous cluster

| Kernel | RI5CY without CGRA ($\mu$J) | RI5CY with CGRA ($\mu$J) | Difference |
|---|---|---|---|
| FFT | 0.824399 | 0.897049 | 8% |
| FIR | 0.151022 | 0.162212 | 7% |
| IIR | 1.198892 | 1.287721 | 7% |
| k-means | 25.405160 | 26.670570 | 5% |
| matMUL | 2.842896 | 3.051363 | 7% |
| CONV | 0.567142 | 0.614512 | 8% |
| DWT | 0.106865 | 0.112640 | 5% |
| SVM | 0.383517 | 0.417095 | 8% |
| CCA | 1.569849 | 1.714678 | 9% |

Table 7.13: Energy consumption of heterogeneous cluster executing on RI5CY with and without CGRA

### Energy consumption of heterogeneous cluster

Table 7.12 shows the energy consumption of the heterogeneous cluster while executing kernels with different configurations, i.e., (1) executing a kernel using CGRA sub-system only, (2) executing a kernel using RI5CY sub-system only, or (3) executing a kernel using both sub-systems. Furthermore, the overall gain pattern in Table 7.12 is similar to Table 7.11, which supports that CGRA does not bring or add a significant overhead to the heterogeneous cluster. A detailed exploration of this effect is performed in the next sub-section.

### Energy consumption of heterogeneous cluster with and without CGRA

Energy-wise overhead due to leakage current of CGRA sub-system is unavoidable, and the effect of such overhead on the heterogeneous cluster is evaluated in this sub-section. In order to analyze the overhead due to leakage current, two configurations of PULP-Cluster are considered, i.e., (1) heterogeneous cluster featuring CGRA sub-system, and (2) PULP-Cluster without CGRA sub-system. This experiment empirically shows that the energy consumption differences between the two configurations are low enough to include both sub-systems in PULP-Cluster and still be within the ULP domain. Table 7.13 shows the energy consumption differences of executing kernels between two configurations. It can be observed that the overhead is always below 10%, which concludes that the integration of the CGRA sub-system into the PULP-Cluster does not bring any significant overhead or downgrades the overall performance of the PULP-Cluster.

| Kernel | Static PE Utilization | Mean PE Utilization | Peak PE Utilization | avg. Power Consumption ($\mu$W) |
|---|---|---|---|---|
| FFT | 7% | 42% | 50% | 5.114 |
| FIR | 25% | 66% | 100% | 4.120 |
| IIR | 13% | 56% | 100% | 4.449 |
| k-means | 7% | 36% | 75% | 5.041 |
| matMUL | 50% | 82% | 100% | 4.393 |
| CONV | 7% | 44% | 100% | 4.244 |
| DWT | 25% | 50% | 75% | 3.872 |
| SVM | 7% | 32% | 75% | 3.996 |
| CCA | 7% | 35% | 100% | 3.856 |

Table 7.14: Correlation between PE Utilization and average Power consumption in CGRA

| Kernel | CGRA PE Utilization | RI5CY core Utilization |
|---|---|---|
| FFT | 42% | 92% |
| FIR | 66% | 61% |
| IIR | 56% | 71% |
| k-means | 36% | 80% |
| matMUL | 82% | 87% |
| CONV | 44% | 75% |
| DWT | 50% | 52% |
| SVM | 32% | 50% |
| CCA | 35% | 44% |

Table 7.15: Dynamic PE Utilization of CGRA and dynamic core utilization of RI5CY

### 7.2.5 Utilization

Table 7.14 shows the correlation between average power consumption and PE utilization of CGRA. Here, (A) static PE utilization is the minimum number of PEs that are always active during the execution of a kernel, (B) mean PE utilization is the average number of active PEs during an entire execution of a kernel, and (C) peak PE utilization is the highest number of active PEs at any timestamp during the execution of a kernel. The varying PE utilization is resulting in the variation of the average power consumption of CGRA. Such effect is due to the implementation of a hierarchical clock-gating scheme in CGRA, which is done at 3 different levels:

1. clock-gate PE, if End-of-Execution is reached.

2. clock-gate all modules, if stalls(s) encountered due to LOAD/STORE operations(s) (due to issue of stall of 1 clock-cycle by a cluster interconnect for each data load or store from a distinct memory bank in TCDM [94]) or if NOP operation is

encountered.

3. clock-gate ALU, if FP operators are executing and vice-versa.

Furthermore, Table 7.15 shows the dynamic PE utilization of CGRA and dynamic core utilization of RI5CY. Although CGRA exhibits a relatively low PE utilization w.r.t. RI5CY, CGRA can demonstrate similar or better performances as a highly optimized multi-core ASIP, i.e., RI5CY. This is mainly due to the simple ISA choice of CGRA, which resulted in the simple (no pipeline) design of PEs. Thus, CGRA can efficiently exploit the parallelization of kernels (in terms of lesser energy consumption) w.r.t. complex 4-stage pipeline design of RI5CY.

## 7.3 Summary and Concluding Results

In the first part of this chapter, it is empirically presented that TRANSPIRE achieves better performance w.r.t. TRANSPIRE_FPU, RI5CY_SFU, and RI5CY_FPU. Particularly, TRANSPIRE optimally exploits instruction parallelism and data parallelism in the applications presented to achieve a maximum of 10.06× better performance and consumed 12.91× less energy w.r.t. RI5CY core with an area overhead of 1.25× only.

In the second part of this chapter, the heterogeneous cluster is implemented (design sign-off) in a 22nm process node and is evaluated using a set of kernels used in different near-sensor processing applications fields, particularly audio, image, bio-signal, and embedded ML. With the help of an extensive exploration of the design space with different configurations to pull off the highest performances from the heterogeneous cluster in terms of latency, power, and area is presented. Finally, it is concluded that the integration of CGRA into the PULP-Cluster, does not bring and significant (energy-wise) overhead or degrades the overall performance of the PULP-Cluster.

# Chapter 8

# Conclusion & Future Work

## Summary of the Research Work

The presented thesis is divided into three parts. (1) First part shows the architecture of an energy-efficient programmable hardware accelerator with native Floating-Point (FP) support capable of efficiently executing a complete kernel. (2) Second part mainly addresses the challenges of implementing multi-cycle support in the compiler and decoupling of address generation computation from the associated compilation flow. (3) Finally, the third part introduces a heterogeneous platform featuring a CGRA subsystem and a RI5CY sub-system. The heterogeneous platform makes the most different configurations to pull off the highest performances while executing multiple real-world algorithms employed in various near-sensor processing application fields (i.e., audio, image, bio-signals, and embedded Machine Learning). An exhaustive set of results are also presented, with the help of a silicon-proven Parallel Ultra Low Power (PULP) platform targeting Ultra-Low-Power (ULP) application domain, to empirically show that the proposed CGRA, when integrated into a System-on-Chip (SOC) targeting ULP application domain, boosts the performance of the SoC.

### Energy-Efficient Programmable Hardware Accelerator

The first part of the thesis presents the design space exploration of the CGRA with the help of five design optimizations.

- *Design 1* presents a CGRA with native support for State-of-the-Art (SoA) IEEE 754-2008 standard single-precision (32-bit) FP support (i.e., *binary32*). The experiment results show that design optimization 1 can achieve a maximum of 6.5× energy-efficiency compared to a RISC-V based GPP featuring *binary32* FP support while executing kernels used in Electroencephalography (EEG) signal processing application, with an area overhead of 1.9×. Design optimization 1 formed the basis for the implementation of design optimization 2.

- *Design 2* presents a CGRA with *transprecision* computing capabilities. Two custom FP datatype with same dynamic range as that of SoA FP counterparts with less precision bits (i.e., *binary16alt* to substitute *binary32*, and *binary8* to substitute *binary16*, respectively) are featured in this design optimization. This design optimization is first to present a CGRA which integrates (1) *transprecision* computing, (2) Single Instruction Multiple Data, and (2) target ULP application domain. The CGRA can achieve a maximum of 10.06× performance gain and consumes up to 12.91× less energy w.r.t. a RISC-V based GPP with an enhanced ISA supporting SIMD-Style vectorization and FP datatype, while executing a set of near-sensor processing applications, with an area overhead of 1.25× only.

- *Design 3* combines all FP related features of *Design 1* and *Design 2* to present a CGRA featuring both SoA FP datatype and *transprecision* FP datatype. This design optimization was part of an exploration. No additional experiments are performed on *Design 3*.

- *Design 4* is the incremental configuration of *Design 2*. This design optimization featured a 4x4 PE array with support for 32-bit integer and 16-bit *binary16alt* only. Implementation of a hierarchical clock-gating scheme and manual generation of assembly codes with high PE utilization enabled this design optimization to be compared with a multi-core RISC-V-based GPP with an enhanced ISA supporting SIMD-Style vectorization and FP datatype. This design optimization formed the basis of the third part of this thesis.

- *Design 5* presents an exploration and implementation of an 8-bit integer-based ALU for accelerating Neural Network applications. The design includes 32-bit integer (i.e., ALU) and 8-bit integer (i.e., ALU8) operators with a 4x2 PE array. ALU8 features SIMD with SIMD lanes=4. To reduce the memory footprints, ALU8 features a dedicated register file to store the immediate values generated while executing kernels. This design presents a good exploration for accelerating DNNs on CGRA. However, SoA architectures are far superior architectures w.r.t. CGRA and thus restrained from further development of CGRA for accelerating DNNS.

### Compiler Support

The second part of the thesis presents the associated compilation flow of the proposed CGRA. The walk-through of the compilation flow and how homomorphism between the Application and CGRA models made the mapping problem a sub-graph finding problem. Mainly, this part focuses on the two main contributions in the compilation flow. First is the challenges in implementing a technique of mapping multi-cycle operations (i.e., static mapping approach). The second is the decoupling of address generation branches from the application Data Flow Graph (DFG). Both tasks require in-depth knowledge of the compilation flow and CGRA architecture and helped identify a major limitation of executing a large kernel on CGRA.

### Heterogeneous Platform for Transprecision Computing

The point of presenting an energy-efficient ULP CGRA as a hardware accelerator is to improve the overall performance of the SoC. This part of the thesis presents the system-level integration of the proposed CGRA into an SoC to present a heterogeneous platform. *Design 4* presented in the first part of the thesis is integrated into the cluster domain of an SoC, where both the CGRA sub-system and the RISC-V based GPP multi-core sub-system features *transprecision* computing. The heterogeneous cluster is implemented (Design Sign-off) in a commercial 22nm FD-SOI technology. All the experiments presented in this section are obtained from this Placed-and-Routed netlist. Particularly, CGRA can reach a maximum of 4.20× latency gain and consumes a maximum of 7.80× less energy w.r.t. RISC-V based GPP multi-core sub-system without degrading the cluster performances while executing a set of real-world algorithms employed in various near-sensor processing application fields (i.e., audio, image, bio-signals, and embedded Machine Learning).

This thesis presented an energy-efficient CGRA targeting the ULP application domain and its associated compilation flow. To envision the CGRA as a hardware accelerator, the CGRA is integrated in the SoC. A set of real-world algorithms used in the near-sensor processing application fields are considered for benchmarking the proposed CGRA or heterogeneous cluster. All the experiments were conducted using a post-synthesized netlist of the CGRA or the heterogeneous cluster implemented using commercial FD-SOI process nodes. Finally, multiple challenges and future research directions have been identified.

## Concluding Remarks

The research work presented in this thesis can be inferred into a number of objectives. These objectives also lay the basis for possible future work directions.

The research work presented in this thesis can be promptly described as:

- to add support for Floating-Point computations in a programmable architecture targeting ultra-low-power domain applications, and
- to add support for multi-cycle operations in a programmable architecture (previously supporting only single-clock-cycle operations) and its associated toolchain.

The research work presented in this thesis is able to recognize a number of continuing

objectives. These objectives also helped in identifying future work directions. These objectives are as follows.

- Further optimize the proposed CGRA to further clock-gate or datapath-gate available hardware blocks or even individual operators. This must be carefully implemented, and a highly exhaustive design space exploration must be performed to achieve even higher energy efficiency than presented in this thesis.

- Versatility of the proposed CGRA can be tested with even more complex DSP applications, especially (1) applications with unequal loop iterations (w.r.t. number of PEs in the CGRA), (2) inconsistent memory accesses during the execution, and (3) executing very large applications, even larger than Principal Component Analysis (PCA) or Canonical Component Analysis (CCA).

- Include more SoA architectures into consideration while benchmarking the proposed CGRA, especially CGRAs targeting the Digital Signal Processing applications domain.

- Provide a silicon-proven architecture of the proposed CGRA to make it more appealing in the research community and motivate researchers to include this architecture while performing the evaluation of certain reconfigurable or programmable architectures.

The research work presented in this thesis also identified some objectives which can be taken further to extend the versatility of the proposed CGRA architecture. These objectives are as follows.

- Extend support for accelerating Deep Neural Networks (DNNs), especially performing online training of the DNNs at the edge.

- Extend support for accelerating cryptography algorithms with efficient implementation of hardware loops in the proposed CGRA with possibly different configurations.

- Perform design space exploration to increase the number of PEs in the CGRA up to two orders of magnitude (100×). Such large CGRA will result in an increase in power consumption, and to compensate for such overhead, integrating the resulted CGRA into a single-core microcontroller system like Pulpino featuring zero-RI5CY [98] would be an imminent logical step to consider.

## Future Work

This work help in identifying multiple directions for future work in the proposed CGRA.

- Execution model of proposed CGRA synchronizes all PEs before any $JUMP$ or Conditional-$JUMP$ statements. Due to such an execution model, the design space exploration to implement hardware loops in each PE requires PEs to be active all the execution time. This eventually degraded the energy efficiency of CGRA, as none of the PEs can be temporarily clock-gated.

- Results obtained from the proposed heterogeneous platform indicate that the exploration of executing Stencil kernels on this tiny CGRA could be a success and extend the applicability of the proposed CGRA.

- Another exploration can be done with applications used in the field of cryptography. The proposed CGRA demonstrated to efficiently execute complex real-world applications and to extend the application field of the proposed CGRA. Cryptography can also be considered in future exploration work.

## Auto-Partition

Proposed CGRA features limited memory storage for context and data. To execute a large kernel in the CGRA, that kernel needs to be segregated into smaller chunks of sizes that can fit into the tiny memories of the proposed CGRA. To partition any kernel requires an in-depth knowledge of the application field and the application segregation process is also time-consuming. All the data dependencies in the kernel need to be identified to perform any partitioning of the kernel. Exploration has been done addressing this limitation, where all DFGs of the application are analyzed. Local groups of the nodes with data dependencies are identified, and such local groups are coarsened. Then, each coarsened graph is then analyzed to identify the final instruction and constant counts. If the obtained coarsened graph is small enough to fit into the memories of CGRA, then the final bitstream is generated. If the coarsened graph is larger than the required size, then the process is repeated by uncoarsening the already coarsened graph and analyzing further segregation. The proposed technique is implemented in the assembler where the large application has been mapped already. The same technique can be replicated in the compiler during the mapping process and is left as future work.

## Hardware Loops

An exploration is made to efficiently implement hardware loops in each PE of CGRA. However, the execution model of CGRA to synchronize each PE before any $JUMP$ or Conditional-$JUMP$ operation requires all PEs to be active. This degraded the overall energy efficiency of the proposed CGRA, as clock-gating of PEs can not be executed in case of NOP or stalls. The implementation of the hardware loop was performed as an extension of the already implemented Flexible-AGU. This limited the exploration to implement the hardware loop as a separate module. If a separate hardware loop module

is to be explored, the implementation of execution of multiple threads must be taken into consideration.

## Acceleration of Stencils Kernels

Interesting performance figures are obtained from the proposed CGRA while executing real-world algorithms employed in the various near-sensor processing application fields. This gives an incentive to further extend the applicability of the proposed tiny CGRA by including complex Stencil kernels.

## Cryptography

Another application field that can be included in the list of exploration is Cryptography. Cryptography applications are always evolving, and due to this, these applications are mainly realized on FPGAs. ASIC implementations are rarely seen for these applications in the research field as they tend to go obsolete very soon. In this scenario, CGRAs can be a boon as they provide both flexibility and energy efficiency.

# Appendix A

# Evaluation of Architectures

A simple dot-product example is used to demonstrate where CGRA stands when compared to a RISC-V-based highly optimized multi-core ASIP, namely RI5CY. To maintain consistency in this experiment, SIMD is disabled on all architectures, so that one might not get added advantage over the other, and common overheads are also omitted from the comparison. All architectures execute an optimized version of the following C-code:

```
\\ ** SIZE = Pre-defined Vector size **
int i; float A[SIZE], B[SIZE], temp, RESULT;
for (i = 0; i < SIZE; i++){
  temp += A[i] * B[i];
}
RESULT = temp;
```

Table A.1 shows the architectural features of RI5CY and CGRA sub-systems. Some important points to be noted are:

- At 100% PE utilization, CGRA can execute a total of 16 parallel FP operations with SIMD disabled while 1-core and 8-cores RI5CY can issue only 1 and 4 parallel FP operations with SIMD disabled, respectively.

- While maintaining 100% PE utilization in CGRA is rare, RI5CY can efficiently maintain a high utilization. On top of that, RI5CY cores fail to maintain an Instruction Per Clock-cycle (IPC) of 1, but these cores benefit from an in-order 4-stage pipeline scheme and can efficiently partition a kernel and execute them separately.

- RI5CY cores can handle LOADs / STOREs much efficiently w.r.t. CGRA. Moreover, RI5CY can also efficiently use the fused-multiply-accumulate operator for calculating dot-product, while CGRA can efficiently exploit local registers in each PE and cheap *MOVE* operations to shift data around the PEs to avoid duplicate LOADs from memory.

Table A.2 shows bare-bones versions of latency (in cycles) and Table A.3 shows the energy consumption (in $\mu$J) numbers from each architecture i.e., CGRA with 4x4 PE

| Features | 1-core RI5CY Sub-system | 8-cores RI5CY Sub-system | 4x4 PE array CGRA Sub-system |
|---|---|---|---|
| Pipeline | Yes | Yes | No |
| SIMD | Yes | Yes | Yes |
| ISA | 32-bit | 32-bit | 21-bit |
| #Cores | 1 | 8 | 16 |
| #LSU | 1 | 8 | 8 |
| #FP Ops. | 1 | 8 | 32 |
| Communication | through TCDM | through TCDM | Mesh Torus network |

#LSU = Total number of Load-Store Units
#FP Ops. = Maximum number of parallel *binary16alt* FP operations (with SIMD) that can be issued
Communication = Shifting/Movement of data among cores in the sub-system

Table A.1: Architectural features of RI5CY and CGRA (4x4 PE)

| Vector Size | CGRA 4x4 PE | RI5CY 1-core | RI5CY 8-cores |
|---|---|---|---|
| 8 | 13 | 60 | 64 |
| 800 | 213 | 3,225 | 462 |
| 8,000 | 1,994 | 32,024 | 4,062 |

Table A.2: Latency comparison of Dot-Product (in cycles)

array, 1-core RI5CY, and 8-cores RI5CY, respectively. It can be observed that, in case of vector size = 8, the ratio between CGRA and RI5CY is higher than vector size = 800 or 8,000 because of the execution of instructions required to setup the variables in kernels outweighs the instructions required for FP calculation in RI5CY. This overhead in RI5CY becomes negligible as the vector size is increased. From Table A.2, it can also be observed that as the ratio between the number of cycles from CGRA and RI5CY architectures are maintained as the vector size is increased (i.e., CGRA : 8-cores RI5CY $\simeq$ 2 ; CGRA : 1-core RI5CY $\simeq$ 16 ) because CGRA can issue a maximum of 16 FP operations while 8-cores RI5CY can issue 4 FP operations and 1-core RI5CY can issue 1 FP operation.

| Vector Size | CGRA 4x4 PE | RI5CY 1-core | RI5CY 8-cores |
|---|---|---|---|
| 8 | 0.000276 | 0.000989 | 0.001784 |
| 800 | 0.004520 | 0.053153 | 0.012879 |
| 8,000 | 0.042313 | 0.527804 | 0.113236 |

Table A.3: Energy consumption comparison of Dot-Product (in $\mu$J). A single netlist processed at 22 nm FD-SOI technology and running at 200 MHz is used for the energy calculation.

# Bibliography


[1] G. Tagliavini et al. "A transprecision floating-point platform for ultra-low power computing." In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. Dresden, Germany: IEEE, 2018, pp. 1051–1056. DOI: 10.23919/DATE.2018.8342167.

[2] A. Pullini et al. "Mr.Wolf: An Energy-Precision Scalable Parallel Ultra Low Power SoC for IoT Edge Processing." In: *IEEE Journal of Solid-State Circuits* 54.7 (2019), pp. 1970–1981. DOI: 10.1109/JSSC.2019.2912307.

[3] E. Azarkhish et al. "Neurostream: Scalable and Energy Efficient Deep Learning with Smart Memory Cubes." In: *IEEE Transactions on Parallel and Distributed Systems* 29.2 (2018), pp. 420–434. DOI: 10.1109/TPDS.2017.2752706.

[4] F. Conti, P. D. Schiavone, and L. Benini. "XNOR Neural Engine: A Hardware Accelerator IP for 21.6-fJ/op Binary Neural Network Inference." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2940–2951. DOI: 10.1109/TCAD.2018.2857019.

[5] F. Conti et al. "An IoT Endpoint System-on-Chip for Secure and Energy-Efficient Near-Sensor Analytics." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.9 (2017), pp. 2481–2494. DOI: 10.1109/TCSI.2017.2698019.

[6] S. Das et al. "A Heterogeneous Cluster with Reconfigurable Accelerator for Energy Efficient Near-Sensor Data Analytics." In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. Florence, Italy: IEEE, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351749.

[7] Leibo Liu et al. "A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications." In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3357375. URL: https://doi.org/10.1145/3357375.

[8] A. Podobas, K. Sano, and S. Matsuoka. "A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective." In: *IEEE Access* 8 (2020), pp. 146719–146743. DOI: 10.1109/ACCESS.2020.3012084.

[9] Bjorn De Sutter, Praveen Raghavan, and Andy Lambrechts. "Coarse-Grained Reconfigurable Array Architectures." In: *Handbook of Signal Processing Systems*. Boston, MA: Springer US, 2010, pp. 449–484. ISBN: 978-1-4419-6345-1. DOI: 10.1007/978-1-4419-6345-1_17. URL: https://doi.org/10.1007/978-1-4419-6345-1_17.

[10] S. Kim et al. "Flexible video processing platform for 8K UHD TV." In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. Cupertino, CA, USA: IEEE, 2015, pp. 1–1. DOI: 10.1109/HOTCHIPS.2015.7477475.

[11] Chris Nicol. "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing." In: *Wave Computing White Paper* . (2017), pp. 1–9. URL: https://wavecomp.ai/wp-content/uploads/2018/12/WP_CGRA.pdf (visited on 02/09/2021).

[12] L. Duch et al. "HEAL-WEAR: An Ultra-Low Power Heterogeneous System for Bio-Signal Analysis." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.9 (2017), pp. 2448–2461. DOI: 10.1109/TCSI.2017.2701499.

[13] STMicroelectronics. "STM32L4 MCU series: Excellence in ultra-low-power with performance." In: *STM32 Ultra Low Power MCUs* . (2018), pp. 1–23. URL: https://www.st.com (visited on 02/09/2021).

[14] Fabio Montagna, Simone Benatti, and Davide Rossi. "Flexible, Scalable and Energy Efficient Bio-Signals Processing on the PULP Platform: A Case Study on Seizure Detection." In: *Journal of Low Power Electronics and Applications* 7.2 (2017), pp. 1–3. ISSN: 2079-9268. DOI: 10.3390/jlpea7020016. URL: https://www.mdpi.com/2079-9268/7/2/16.

[15] Arm Limited. *Arm Cortex-M55 Processor Devices Generic User Guide Revision r0p1*. URL: https://developer.arm.com/documentation/101273/0001/Cortex-M55-Processor-level-components-and-system-registers---Reference-Material/Floating-point-and-MVE-support (visited on 06/04/2021).

[16] Arm Limited. *Cortex-M4 Processor Datasheet*. URL: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4 (visited on 06/04/2021).

[17] Fabio Montagna et al. *A transprecision floating-point cluster for efficient near-sensor data analytics*. 2020. arXiv: 2008.12243 [cs.DC].

[18] Gerald Estrin. "Organization of Computer Systems: The Fixed plus Variable Structure Computer." In: *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*. IRE-AIEE-ACM '60 (Western). San Francisco, California: Association for Computing Machinery, 1960, pp. 33–40. ISBN: 9781450378697. DOI: 10.1145/1460361.1460365. URL: https://doi.org/10.1145/1460361.1460365.

[19] R.W. Hartenstein et al. "A novel ASIC design approach based on a new machine paradigm." In: *IEEE Journal of Solid-State Circuits* 26.7 (1991), pp. 975–989. DOI: 10.1109/4.92017.

[20] D.C. Chen and J.M. Rabaey. "A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths." In: *IEEE Journal of Solid-State Circuits* 27.12 (1992), pp. 1895–1904. DOI: 10.1109/4.173120.

[21] Bingfeng Mei et al. "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix." In: *Field Programmable Logic and Application*. Ed. by Peter Y. K. Cheung and George A. Constantinides. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 61–70. ISBN: 978-3-540-45234-8. DOI: 10.1007/978-3-540-45234-8_7.

[22] H. Singh et al. “MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications.” In: *IEEE Transactions on Computers* 49.5 (2000), pp. 465–481. DOI: 10.1109/12.859540.

[23] Karthikeyan Sankaralingam et al. “TRIPS: A Polymorphous Architecture for Exploiting ILP, TLP, and DLP.” In: *ACM Trans. Archit. Code Optim.* 1.1 (Mar. 2004), pp. 62–93. ISSN: 1544-3566. DOI: 10.1145/980152.980156. URL: https://doi.org/10.1145/980152.980156.

[24] Volker Baumgarten et al. “PACT XPP—A self-reconfigurable data processing architecture.” In: *The Journal of Supercomputing* 26 (Sept. 2003), pp. 167–184. DOI: 10.1023/A:1024499601571.

[25] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it).” In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[26] G. Theodoridis, D. Soudris, and S. Vassiliadis. “A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools.” In: *Fine- and Coarse-Grain Reconfigurable Computing*. Ed. by Stamatis Vassiliadis and Dimitrios Soudris. Dordrecht: Springer Netherlands, 2007, pp. 89–149. ISBN: 978-1-4020-6505-7. DOI: 10.1007/978-1-4020-6505-7_2. URL: https://doi.org/10.1007/978-1-4020-6505-7_2.

[27] Leibo Liu et al. “HReA: An Energy-Efficient Embedded Dynamically Reconfigurable Fabric for 13-Dwarfs Processing.” In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.3 (2018), pp. 381–385. DOI: 10.1109/TCSII.2017.2728814.

[28] Leibo Liu et al. “An Energy-Efficient Coarse-Grained Reconfigurable Processing Unit for Multiple-Standard Video Decoding.” In: *IEEE Transactions on Multimedia* 17.10 (2015), pp. 1706–1720. DOI: 10.1109/TMM.2015.2463735.

[29] Loris Duch. “Hardware / Software Architectural and Technological Exploration for Energy-Efficient and Reliable Biomedical Devices.” In: 2018. URL: https://infoscience.epfl.ch/record/261219/files/EPFL_TH8917.pdf (visited on 06/04/2021).

[30] Tony Nowatzki et al. “Stream-dataflow acceleration.” In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 416–429. DOI: 10.1145/3079856.3080255.

[31] Raghu Prabhakar et al. “Plasticine: A reconfigurable architecture for parallel patterns.” In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 389–402. DOI: 10.1145/3079856.3080256.

[32] DARPA. *Defense Advanced Research Projects Agency*. URL: https://www.darpa.mil/ (visited on 05/05/2021).

[33] Rick Bahr et al. “Creating an Agile Hardware Design Flow.” In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218553.

[34] SAMSUNG. *Exynos 5 Octa (5430)*. URL: https://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5430/ (visited on 05/05/2021).

[35] Intel Corporation. *Intel, Tsinghua University and Montage Technology Collaborate to Bring Indigenous Data Center Solutions to China*. URL: https://newsroom.intel.com/news-releases/intel-tsinghua-university-and-montage-technology-collaborate-to-bring-indigenous-data-center-solutions-to-china/ (visited on 05/05/2021).

[36] M. Suzuki et al. "Stream applications on the dynamically reconfigurable processor." In: *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*. 2004, pp. 137–144. DOI: 10.1109/FPT.2004.1393261.

[37] T. Sato, H. Watanabe, and K. Shiba. "Implementation of dynamically reconfigurable processor DAPDNA-2." In: *2005 IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT)*. 2005, pp. 323–324. DOI: 10.1109/VDAT.2005.1500086.

[38] S. Rixner et al. "A bandwidth-efficient architecture for media processing." In: *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. Dallas, TX, USA, USA: IEEE, 1998, pp. 3–13. DOI: 10.1109/MICRO.1998.742118.

[39] Fan Feng et al. "Floating-point operation based reconfigurable architecture for radar processing." In: *IEICE Electronics Express* 13.21 (2016), pp. 20160893–20160893. DOI: 10.1587/elex.13.20160893.

[40] Manhwee Jo et al. "Design of a Coarse-Grained Reconfigurable Architecture with Floating-Point Support and Comparative Study." In: *Integr. VLSI J.* 47.2 (Mar. 2014), pp. 232–241. ISSN: 0167-9260. DOI: 10.1016/j.vlsi.2013.08.003. URL: https://doi.org/10.1016/j.vlsi.2013.08.003.

[41] Claudio Brunelli et al. "A Coarse-Grain Reconfigurable Architecture for Multimedia Applications Supporting Subword and Floating-Point Calculations." In: *J. Syst. Archit.* 56.1 (Jan. 2010), pp. 38–47. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2009.11.003. URL: https://doi.org/10.1016/j.sysarc.2009.11.003.

[42] X. Fan et al. "Stream Processing Dual-Track CGRA for Object Inference." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.6 (2018), pp. 1098–1111. DOI: 10.1109/TVLSI.2018.2797600.

[43] STMicroelectronics. "STM32H7 Dual Core World Most Powerful MCU." In: *STM32H7 Series* . (2019), pp. 1–41. URL: https://www.st.com (visited on 02/09/2021).

[44] S. Mach et al. "A 0.80pJ/flop, 1.24Tflop/sW 8-to-64 bit Transprecision Floating-Point Unit for a 64 bit RISC-V Processor in 22nm FD-SOI." In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. Cuzco, Peru, Peru: IEEE, 2019, pp. 95–98. DOI: 10.1109/VLSI-SoC.2019.8920307.

[45] B. Zimmer et al. “A 0.11 pJ/Op, 0.32-128 TOPS, Scalable Multi-Chip-Module-based Deep Neural Network Accelerator with Ground-Reference Signaling in 16nm.” In: *2019 Symposium on VLSI Circuits*. Kyoto, Japan, Japan: IEEE, 2019, pp. C300–C301. DOI: 10.23919/VLSIC.2019.8778056.

[46] PULP Platform. *Open hardware, the way it should be!* 2013. URL: https://pulp-platform.org/index.html (visited on 02/09/2021).

[47] Norman P. Jouppi et al. “A Domain-Specific Architecture for Deep Neural Networks.” In: *Commun. ACM* 61.9 (Aug. 2018), pp. 50–59. ISSN: 0001-0782. DOI: 10.1145/3154484. URL: https://doi.org/10.1145/3154484.

[48] M. Gautschi et al. “Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices.” In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506.

[49] Computer History Museum. *1964: FIRST COMMERCIAL MOS IC INTRODUCED*. URL: https://www.computerhistory.org/siliconengine/first-commercial-mos-ic-introduced/ (visited on 06/11/2021).

[50] G. Le Lann. “An analysis of the Ariane 5 flight 501 failure-a system engineering perspective.” In: *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*. Monterey, CA, USA: IEEE, 1997, pp. 339–346. DOI: 10.1109/ECBS.1997.581900.

[51] C. Tsen et al. “A Combined Decimal and Binary Floating-Point Multiplier.” In: *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. Boston, MA, USA: IEEE, 2009, pp. 8–15. DOI: 10.1109/ASAP.2009.28.

[52] A. A. Wahba and H. A. H. Fahmy. “Area Efficient and Fast Combined Binary/Decimal Floating Point Fused Multiply Add Unit.” In: *IEEE Transactions on Computers* 66.2 (2017), pp. 226–239. DOI: 10.1109/TC.2016.2584067.

[53] M. Gautschi et al. “An Extended Shared Logarithmic Unit for Nonlinear Function Kernel Acceleration in a 65-nm CMOS Multicore Cluster.” In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 98–112. DOI: 10.1109/JSSC.2016.2626272.

[54] John Gustafson. *The End of Error: Unum Computing*. USA: Chapman and Hall/CRC Computational Science, Feb. 2015. ISBN: 1482239868. DOI: 10.1201/9781315161532.

[55] F. Glaser et al. “An 826 MOPS, 210uW/MHz Unum ALU in 65 nm.” In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. Florence, Italy: IEEE, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351546.

[56] Stefan Mach et al. *FPnew: An Open-Source Multi-Format Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing*. 2020. arXiv: 2007.01530 [cs.AR].

[57] Jiaxiang Wu et al. “Quantized Convolutional Neural Networks for Mobile Devices.” In: *CoRR* abs/1512.06473 (2015). arXiv: 1512.06473. URL: http://arxiv.org/abs/1512.06473.

[58] M. Zanghieri et al. "Robust Real-Time Embedded EMG Recognition Framework Using Temporal Convolutional Networks on a Multicore IoT Processor." In: *IEEE Transactions on Biomedical Circuits and Systems* 14.2 (2020), pp. 244–256. DOI: 10.1109/TBCAS.2019.2959160.

[59] N. Burgess et al. "Bfloat16 Processing for Neural Networks." In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. Kyoto, Japan: IEEE, 2019, pp. 88–91. DOI: 10.1109/ARITH.2019.00022.

[60] G. Tagliavini, A. Marongiu, and L. Benini. "FlexFloat: A Software Library for Transprecision Computing." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.1 (2020), pp. 145–156. DOI: 10.1109/TCAD.2018.2883902.

[61] S. Mach et al. "A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing." In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. Florence, Italy: IEEE, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351816.

[62] G. Burns et al. "Reconfigurable accelerator enabling efficient SDR for low-cost consumer devices." In: *SDR Technical Forum*. Orlando, USA, 2003.

[63] Ming-Hau Lee et al. "Design and Implementation of the MorphoSys Reconfigurable Computing Processor." In: *Field-Programmable Custom Computing Technology: Architectures, Tools, and Applications*. Ed. by Jeffrey Arnold, Wayne Luk, and Ken Pocek. Boston, MA: Springer US, 2000, pp. 21–38. ISBN: 978-1-4615-4417-3. DOI: 10.1007/978-1-4615-4417-3_3. URL: https://doi.org/10.1007/978-1-4615-4417-3_3.

[64] Carl Ebeling. *The General Rapid Architecture Description*. Mar. 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.11.314&rep=rep1&type=pdf.

[65] Eberhard Schüler and Markus Weinhardt. "XPP-III." In: *Dynamic System Reconfiguration in Heterogeneous Platforms: The MORPHEUS Approach*. Ed. by Nikolaos S. Voros, Alberto Rosti, and Michael Hübner. Dordrecht: Springer Netherlands, 2009, pp. 63–76. ISBN: 978-90-481-2427-5. DOI: 10.1007/978-90-481-2427-5_6. URL: https://doi.org/10.1007/978-90-481-2427-5_6.

[66] R. Hartenstein et al. "Mapping Applications onto Reconfigurable KressArrays." In: *Field Programmable Logic and Applications*. Ed. by Patrick Lysaght, James Irvine, and Reiner Hartenstein. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 385–390. ISBN: 978-3-540-48302-1.

[67] Joseph Fisher, Paolo Faraboschi, and Cliff Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Jan. 2005. ISBN: 978-1-55860-766-8. URL: http://www.vliw.org/book/.

[68] John Paul Shen and M. Lipasti. "Modern Processor Design: Fundamentals of Superscalar Processors." In: 2002.

[69] K. Sankaralingam et al. “Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture.” In: *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.* 2003, pp. 422–433. DOI: 10.1109/ISCA.2003.1207019.

[70] Mark Gebhart et al. “An Evaluation of the TRIPS Computer System.” In: *SIGARCH Comput. Archit. News* 37.1 (Mar. 2009), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/2528521.1508246. URL: https://doi.org/10.1145/2528521.1508246.

[71] Daniele Paolo Scarpazza et al. “Software Simultaneous Multi-Threading, a Technique to Exploit Task-Level Parallelism to Improve Instruction- and Data-Level Parallelism.” In: *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*. Ed. by Johan Vounckx, Nadine Azemard, and Philippe Maurine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 12–23. ISBN: 978-3-540-39097-8. DOI: 10.1007/11847083_2.

[72] Hung-Wei Tseng and Dean M. Tullsen. “Data-triggered threads: Eliminating redundant computation.” In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 2011, pp. 181–192. DOI: 10.1109/HPCA.2011.5749727.

[73] Hyunchul Park, Yongjun Park, and Scott Mahlke. “Polymorphic Pipeline Array: A flexible multicore accelerator with virtualized execution for mobile multimedia applications.” In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009, pp. 370–380. DOI: 10.1145/1669112.1669160.

[74] B. R. Rau and C. D. Glaeser. “Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing.” In: *SIGMICRO Newsl.* 12.4 (Dec. 1981), pp. 183–198. ISSN: 1050-916X. DOI: 10.1145/1014192.802449. URL: https://doi.org/10.1145/1014192.802449.

[75] Bingfeng Mei et al. “Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling.” In: *2003 Design, Automation and Test in Europe Conference and Exhibition*. 2003, pp. 296–301. DOI: 10.1109/DATE.2003.1253623.

[76] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. “EPIMap: Using Epimorphism to map applications on CGRAs.” In: *DAC Design Automation Conference 2012*. 2012, pp. 1280–1287. DOI: 10.1145/2228360.2228600.

[77] M. Hamzeh, A. Shrivastava, and S. Vrudhula. “REGIMap: Register-aware application mapping on Coarse-Grained Reconfigurable Architectures (CGRAs).” In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–10. DOI: 10.1145/2463209.2488756.

[78] Steven Swanson et al. “The WaveScalar Architecture.” In: *ACM Trans. Comput. Syst.* 25.2 (May 2007). ISSN: 0734-2071. DOI: 10.1145/1233307.1233308. URL: https://doi.org/10.1145/1233307.1233308.

[79] Dani Voitsechov and Yoav Etsion. "Single-graph multiple flows: Energy efficient design alternative for GPGPUs." In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 205–216. DOI: 10.1109/ISCA.2014.6853234.

[80] Satyajit Das et al. "An Energy-Efficient Integrated Programmable Array Accelerator and Compilation Flow for Near-Sensor Ultralow Power Processing." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.6 (2018), pp. 1095–1108. DOI: 10.1109/TCAD.2018.2834397.

[81] S.C. Goldstein et al. "PipeRench: a reconfigurable architecture and compiler." In: *Computer* 33.4 (2000), pp. 70–77. DOI: 10.1109/2.839324.

[82] Venkatraman Govindaraju et al. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing." In: *IEEE Micro* 32.5 (2012), pp. 38–51. DOI: 10.1109/MM.2012.51.

[83] Jared Pager, Reiley Jeyapaul, and Aviral Shrivastava. "A Software Scheme for Multithreading on CGRAs." In: *ACM Trans. Embed. Comput. Syst.* 14.1 (Jan. 2015). ISSN: 1539-9087. DOI: 10.1145/2638558. URL: https://doi.org/10.1145/2638558.

[84] Changkyu Kim et al. "Composable Lightweight Processors." In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 2007, pp. 381–394. DOI: 10.1109/MICRO.2007.41.

[85] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. URL: https://www.eclipse.org/modeling/emf/ (visited on 06/03/2021).

[86] IBM Corporation. *The Eclipse Modeling Framework (EMF) Overview*. 2005. URL: https://www.ibm.com/docs/en/z-open-development/2.0.x?topic=guides-emf-framework-programmers-guide (visited on 06/03/2021).

[87] Philippe Coussy et al. "GAUT: A High-Level Synthesis Tool for DSP Applications." In: *High-Level Synthesis: From Algorithm to Digital Circuit*. Ed. by Philippe Coussy and Adam Morawiec. Dordrecht: Springer Netherlands, 2008, pp. 147–169. ISBN: 978-1-4020-8588-8. DOI: 10.1007/978-1-4020-8588-8_9. URL: https://doi.org/10.1007/978-1-4020-8588-8_9.

[88] Satyajit Das et al. "Efficient mapping of CDFG onto coarse-grained reconfigurable array architectures." In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 127–132. DOI: 10.1109/ASPDAC.2017.7858308.

[89] T. Peyret et al. "Efficient application mapping on CGRAs based on backward simultaneous scheduling/binding and dynamic graph transformations." In: *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*. 2014, pp. 169–172. DOI: 10.1109/ASAP.2014.6868652.

[90] Satyajit Das et al. "A Scalable Design Approach to Efficiently Map Applications on CGRAs." In: *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2016, pp. 655–660. DOI: 10.1109/ISVLSI.2016.54.

[91] D. Rossi et al. “Energy-Efficient Near-Threshold Parallel Computing: The PULPv2 Cluster.” In: *IEEE Micro* 37.5 (2017), pp. 20–31. DOI: 10.1109/MM.2017.3711645.

[92] OpenRISC. *OpenRISC 1000 Architecture*. 2014. URL: http://openrisc.io/or1k.html (visited on 02/09/2021).

[93] Michael Gautschi et al. “Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of OpenRISC cores.” In: *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2015, pp. 25–30. DOI: 10.1109/VLSI-SoC.2015.7314386.

[94] A. Rahimi et al. “A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters.” In: *2011 Design, Automation Test in Europe*. Grenoble, France: IEEE, 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763085.

[95] Michael J. Flynn. “Some Computer Organizations and Their Effectiveness.” In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071.

[96] Inc. Object Management Group®. *What is UML — Unified Modeling Language*. 2005. URL: https://www.uml.org/what-is-uml.htm (visited on 06/03/2021).

[97] Giorgio Levi. “A note on the derivation of maximal common subgraphs of two directed or undirected graphs.” In: *Calcolo* 9.4 (1973), pp. 341–352. URL: https://doi.org/10.1007/BF02575586.

[98] Pasquale Davide Schiavone et al. “Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications.” In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976.

[99] Krste Asanović Andrew Waterman. “The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA.” In: *RISCV Foundation* (2019). URL: https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf.

[100] P. Meinerzhagen, C. Roth, and A. Burg. “Towards generic low-power area-efficient standard cell based memory architectures.” In: *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*. 2010, pp. 129–132. DOI: 10.1109/MWSCAS.2010.5548579.

[101] Igor Loi et al. “The Quest for Energy-Efficient I$ Design in Ultra-Low-Power Clustered Many-Cores.” In: *IEEE Transactions on Multi-Scale Computing Systems* 4.2 (2018), pp. 99–112. DOI: 10.1109/TMSCS.2017.2769046.

[102] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[103] NVDLA. *NVIDIA Deep Learning Accelerator*. URL: http://nvdla.org/ (visited on 05/17/2021).

[104] Dhiraj Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. arXiv: 1905.12322 [cs.LG].

[105] Gianmarco Ottavi et al. “A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference.” In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 512–517. DOI: 10.1109/ISVLSI49217.2020.000-5.

[106] Manuele Rusci, Alessandro Capotondi, and Luca Benini. *Memory-Driven Mixed Low Precision Quantization For Enabling Deep Network Inference On Microcontrollers*. 2019. arXiv: 1905.13082 [cs.LG].

[107] Bert Moons et al. “Minimum energy quantized neural networks.” In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 2017, pp. 1921–1925. DOI: 10.1109/ACSSC.2017.8335699.

[108] Itay Hubara et al. *Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations*. 2016. arXiv: 1609.07061 [cs.NE].

[109] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. USA: Kluwer Academic Publishers, 1999. ISBN: 0792384601. DOI: 10.1007/978-1-4615-5145-4.

[110] Scott A. Mahlke et al. “Effective Compiler Support for Predicated Execution Using the Hyperblock.” In: *SIGMICRO Newsl.* 23.1–2 (Dec. 1992), pp. 45–54. ISSN: 1050-916X. DOI: 10.1145/144965.144998. URL: https://doi.org/10.1145/144965.144998.

[111] B. Ramakrishna Rau. “Iterative modulo Scheduling: An Algorithm for Software Pipelining Loops.” In: *Proceedings of the 27th Annual International Symposium on Microarchitecture*. MICRO 27. San Jose, California, USA: Association for Computing Machinery, 1994, pp. 63–74. ISBN: 0897917073. DOI: 10.1145/192724.192731. URL: https://doi.org/10.1145/192724.192731.

[112] M. Lam. “Software Pipelining: An Effective Scheduling Technique for VLIW Machines.” In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI ’88. Atlanta, Georgia, USA: Association for Computing Machinery, 1988, pp. 318–328. ISBN: 0897912691. DOI: 10.1145/53990.54022. URL: https://doi.org/10.1145/53990.54022.

[113] Mickael Lanoe et al. “A modeling and code generation framework for critical embedded systems design: From Simulink down to VHDL and Ada/C code.” In: *2014 21st IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2014, pp. 742–745. DOI: 10.1109/ICECS.2014.7050092.

[114] GCC team. *12 GIMPLE*. URL: https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html (visited on 06/03/2021).

[115] STMicroelectronics. *Learn More About FD-SOI*. URL: https://www.st.com/content/st_com/en/about/innovation---technology/FD-SOI/learn-more-about-fd-soi.html (visited on 04/29/2021).