Politecnico di Torino

Politecnico di Torino

Master's Degree in Electronic Engineering

A.a. 2024/2025

Graduation Session 09/2025

RISC-V Enabled CGRA SystemC Model for AI and ML Applications

Supervisors:

Prof. Guido Masera

Prof. Maurizio Martina

PhD Candidate Luigi Giuffrida

Candidate:

Saman Alipour

Abstract

The increasing demand for high-performance computation in fields such as signal processing, scientific computing, and embedded systems has highlighted the need for specialized hardware accelerators. Several architectural solutions exist, including Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), and Coarse Grained Reconfigurable Architectures (CGRAs).

The objective of this thesis is the design of a generic CGRA architecture capable of operating on floating-point data. The processing elements within the array support a broad range of functions, including arithmetic operations, Multiply-Accumulate (MAC), direct and inverse trigonometric functions, direct and inverse hyperbolic functions, and exponential functions.

For testing and benchmarking, the proposed accelerator model has been integrated into a SystemC open-source model of a RISC-V-based microcontroller through an Advanced eXtensible Interface (AXI) bus interface. This setup enables the execution of computational kernels and the evaluation of their performance in terms of clock cycles.

In conclusion, the proposed model, combined with the systemC platform, has been shown to effectively benchmark and test the execution of Artificial Intelligence (AI)/Machine Learning (ML) and automotive applications on highly configurable and parallel architectures as the proposed CGRA.

Acknowledgements

I would like to express my deepest gratitude to my supervisors, **Prof. Guido Masera**, **Prof. Maurizio Martina**, and **Dr. Luigi Giuffrida**, for their invaluable guidance, insightful feedback, and constant encouragement throughout the development of this thesis. Their expertise and dedication have been a continuous source of inspiration, shaping both the technical and methodological aspects of my work.

I am sincerely thankful for their patience, availability, and the many constructive discussions that helped me navigate challenges and refine my ideas. It has been an honor to work under their supervision and to learn from their profound knowledge and experience.

Finally, I would also like to thank my family and friends for their unwavering support and understanding during this journey.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Context and Motivation

In recent years, the demand for efficient and flexible computing platforms has grown rapidly due to the increasing adoption of *ML* and *AI* [**liu2019cgra**] in a wide variety of application domains, including automotive, embedded systems, and edge computing. Traditional general-purpose processors such as CPUs often fail to deliver the required performance and energy efficiency for these workloads, while fully customized accelerators such as ASICs may lack flexibility and involve high development costs [1] [2].

As a promising alternative, **CGRAs** have gained significant attention. A CGRA is a programmable architecture composed of an array of interconnected *PEs*, where each PE typically performs arithmetic or logic operations. Unlike fine-grained architectures such as FPGAs, which configure at the bit level, CGRAs operate at a word level granularity [1] [**liu2019cgra**], allowing for a good trade off between performance, energy efficiency, and programmability. This makes them suitable for applications where specialized mathematical operations are required but full hardware customization is not desirable.

By providing flexibility in mapping algorithms and reusing the same hardware for different workloads, CGRAs can accelerate specific computational kernels with reduced design effort compared to ASICs and improved efficiency compared to CPUs or GPUs [1] [3]. Their reconfigurability also makes them attractive for domains such as automotive, where the hardware platform must support evolving standards and algorithms while maintaining predictable performance.

1.2 Goal of the Thesis

The goal of this thesis is to design and model a **generic CGRA architecture** in SystemC. The proposed architecture is conceived to be **configurable** with respect to several key design parameters:

- the **bitwidth** of the datapath,
- the **latency of each PE**,
- and the **number of available PEs** in the array.

The PEs in this CGRA support **floating-point operations**, enabling accurate and efficient computation for workloads with high numerical demands. In addition to the fundamental arithmetic operations (addition, subtraction, multiplication, and multiply–accumulate), the architecture includes support for:

- **trigonometric functions** (direct and inverse),
- **hyperbolic functions** (direct and inverse),
- **exponential function**.

These capabilities make the proposed CGRA suitable for scientific computations and AI/ML workloads that rely on nonlinear activation functions, signal processing routines, and other advanced mathematical operations [4], rather than traditional linear algebra kernels.

Once the CGRA architecture is developed, it will be integrated with a **RISC-V** core on a systemC platform [5]. This integration will enable running benchmarks to evaluate the performance of the proposed CGRA when coupled with a general-purpose core. The benchmarking phase will assess the effectiveness of the design in accelerating representative workloads and demonstrate the benefits of reconfigurable architectures in embedded computing scenarios.

The remainder of this thesis is organized as follows:

- **Chapter 1** introduces the context, motivation, and objectives of the work.
- **Chapter 2** provides the background, including an overview of CGRAs, a comparison with other architectures such as CPUs, GPUs, FPGAs, and ASICs, their applications in scientific and AI domains, and a discussion of *SystemC* for modeling and the role of the RISC-V core.
- **Chapter 3** presents a design overview of the proposed CGRA architecture, highlighting the main blocks and interconnection.
- **Chapter 4** describes the design details of the architecture and its blocks.

- **Chapter 5** discusses the integration of the CGRA with the RISC-V core and describing the communication mechanisms.
- **Chapter 6** reports and analyzes the benchmarking results.
- **Chapter 7** concludes the thesis and outlines possible directions for future work.

Chapter 2

Background

2.1 CGRA

2.1.1 Definition

CGRAs are programmable hardware accelerators designed to provide a balance between the performance of ASIC and the flexibility of general purpose processors. Unlike fine grained architectures such as FPGAs, which configure at the level of individual logic gates or bits, CGRAs operate at the word level (typically 8, 16, or 32 bits). This coarser granularity enables a more efficient mapping of arithmetic operations, while still maintaining some degree of reconfigurability [1].

A typical CGRA is organized as an array of *PE*, interconnected through a reconfigurable network. Each PE can perform arithmetic or logical operations, and the network determines how data flows across the array. By configuring the interconnect and the operation of each PE, a CGRA can be tailored to accelerate a wide range of computational kernels.

2.1.2 Architecture and Features

The main characteristics of a CGRA can be summarized as follows:

- **PE:** The building blocks of a CGRA, typically supporting arithmetic and logical operations. In more advanced designs, PEs may also implement complex mathematical functions [4] such as trigonometric, hyperbolic, or exponential computations.
- **Interconnection Network:** A reconfigurable routing fabric that defines how data is transferred among PEs.
- **Local Memory:** Provides fast storage close to the PEs for operands and intermediate results, reducing the need for repeated access to external memory.

- **DMA:** Responsible for efficiently reading input data from local memory into the PEs and writing back computation results.
- **Configuration Registers:** Store the control information needed to configure the PEs, DMAs and the interconnect, also allows mapping different applications onto the same hardware by changing the configuration without altering the underlying physical design.

2.2 Comparison with Other Architectures

2.2.1 CPU

CPUs are designed to handle a wide range of tasks with maximum flexibility. They offer well established toolchains and are straightforward to program, which makes them highly versatile. However, CPUs rely on a sequential execution model with limited parallelism, and as a result, they struggle to deliver the performance and energy efficiency required by compute intensive workloads [1] such as those found in ML and AI.

2.2.2 GPU

GPUs provide massive parallelism and have become the dominant platform for training and inference in modern ML [1] [3]. Their programming model, however, is relatively complex, and their high power consumption can limit their applicability in embedded or energy constrained environments. While GPUs excel at large scale parallel data processing, they are not always the most suitable choice for low-power domains such as automotive or Internet Of Things (IOT) systems.

2.2.3 FPGA

FPGAs offer fine grained configurability and can achieve high performance for specific applications. However, their flexibility comes at the cost of design complexity: developing efficient FPGA based accelerators requires Hardware Description Languages (HDLs) expertise and specialized toolchains. This makes programming and optimization significantly more challenging compared to CGRAs, which operate at a higher level of abstraction and provide a more software-friendly reconfiguration model [1] [2].

2.2.4 ASIC

ASICs deliver the highest performance and energy efficiency for a given application, as the hardware is fully tailored to the target workload. The drawback is their

lack of flexibility: once fabricated, an ASIC cannot be adapted to new algorithms or evolving standards. This rigidity makes ASICs unsuitable in domains where adaptability and long term reusability [1] [2] are important.

2.2.5 Position of CGRA

CGRAs occupy an intermediate position in the design space. They provide better efficiency than CPUs and GPUs for targeted kernels, greater flexibility than ASICs, and a more accessible programming model than FPGAs [1]. This balance makes CGRAs particularly attractive for workloads that require both performance and adaptability, such as AI inference or scientific computations involving nonlinear mathematical functions.

2.2.6 Comparison Summary

Table 2.1 provides a qualitative comparison of the main characteristics of different architectures.

Table 2.1: Comparison of architectures in terms of flexibility, efficiency, and programmability.

Architecture	**Flexibility**	**Efficiency**	**Programmability**
CPU	High	Low	Easy
GPU	Medium	High	Medium
FPGA	High	High	Difficult
ASIC	None	Very High	Not Applicable
CGRA	Medium	Medium–High	Medium–Easy

2.3 Applications in AI and Scientific Computing

AI and scientific workloads often rely on computationally intensive mathematical functions, such as nonlinear activation functions(e.g., sigmoid, hyperbolic tangent), trigonometric evaluations, or exponential calculations. While matrix-heavy linear algebra is a primary focus for many accelerators, there is also a strong need for efficient hardware support of nonlinear functions.

CGRAs are well suited to this domain because:

- They can implement specialized PEs that directly support floating-point arithmetic.

- They provide configurability to adapt to different function pipelines.
- They offer a middle ground between fixed function accelerators and fully general purpose solutions, enabling the execution of both AI inspired and scientific kernels.

As a result, CGRAs are gaining attention in research and industry for applications where flexibility, reusability, and efficiency must coexist.

2.4 SystemC for Architecture Modeling

SystemC is a C++ based modeling framework widely used for system level design and simulation of hardware architectures. It provides constructs to describe both the structural and behavioral aspects of hardware, while enabling simulation at multiple levels of abstraction [6](transaction-level, cycle-accurate, or register-transfer level).

Key advantages of using SystemC for CGRA modeling include:

- **High-level abstraction:** Enables faster development and exploration of design alternatives without the need for low-level HDL coding.
- **Reusability:** Modules can be parameterized and reused across multiple experiments.
- **Fast simulation:** Compared to RTL simulation, SystemC provides faster execution, making it practical for benchmarking.

SystemC therefore provides an effective environment for developing and validating the configurable CGRA architecture presented in this thesis.

2.5 RISC-Vp++

RISC-Vp++ is a SystemC-based enhanced virtual platform built around the RISC-V instruction set architecture. It is designed to support architectural exploration, system-level design, and research in processor-accelerator integration. As an open-source and modular platform, RISC-Vp++ provides a flexible simulation environment for experimenting with new hardware components and communication mechanisms [5].

In this thesis, the RISC-Vp++ platform is used as the host microcontroller and control unit for the proposed CGRA architecture. Specifically, the RISC-V core is responsible for:

- Managing the configuration and control of the CGRA,

- Providing a software interface to initiate and coordinate workload execution.

The integration of the SystemC-modeled RISC-Vp++ platform with the proposed CGRA enables system-level benchmarking and evaluation. This setup highlights the benefits and trade-offs of reconfigurable acceleration in embedded applications, particularly for scientific computing and AI/ML workloads.

Chapter 3

Design Overview

3.1 Accelerator architecture

The proposed accelerator is composed of an array of PEs, connected through a mesh interconnection network. This topology allows each PE to access not only its own input operands, but also the results produced by other PEs in the array. Such flexibility enables both memory-driven execution and data reuse across the computed results [1].

Each PE supports a rich set of floating-point operations, including:

- **Arithmetic:** addition, subtraction, and multiplication.
- **MAC:** iterative accumulation over a programmable number of iterations.
- **Trigonometric functions:** direct functions (sin, cos) and inverse functions (arctan, arcsin, arccos).
- **Hyperbolic functions:** direct (sinh, cosh) and inverse (arctanh).
- **Exponential function:** e^x.

This capability allows the array to efficiently accelerate a wide range of scientific, signal processing, and ML/AI workloads that rely heavily on floating-point computation.

In addition to the compute array, the architecture integrates an internal memory subsystem together with multiple DMA units. Each row of the array is equipped with two read DMAs, which provide input operands from memory to all PEs in the row, and one write DMA to write the result of the row in memory. Consequently, for an array of I rows and J columns, the system instantiates $2I$ read DMAs, I write DMAs and $I * J$ PEs.

Each PE can flexibly select its inputs either from the row's read DMAs or from the outputs of other PEs in the mesh. This choice is performed by multiplexers located at the input of every PE. Similarly, the write DMA in each row is responsible for writing results back to memory. A multiplexer at the row level selects which PE result is forwarded to the write DMA.

Since the architecture features multiple rows, and each row integrates two read channels, the memory must sustain a high number of simultaneous accesses. For this reason, the internal memory is organized as a **multi-bank memory** with multiple read and write ports. Each DMA is connected to a dedicated port, from which it can access any of the banks. This design ensures sufficient bandwidth for concurrent operand fetching and result storing [7].

Figure 3.1 illustrates this organization for a design with I rows and J columns. Memory feeds the read DMAs, which distribute operands to the PEs; results are collected by the write DMAs and written back into memory. As mentioned earlier, a design with I rows includes $2I$ read DMAs and I write DMAs, while the memory provides $2I$ ports to serve concurrent read and write traffic [2].

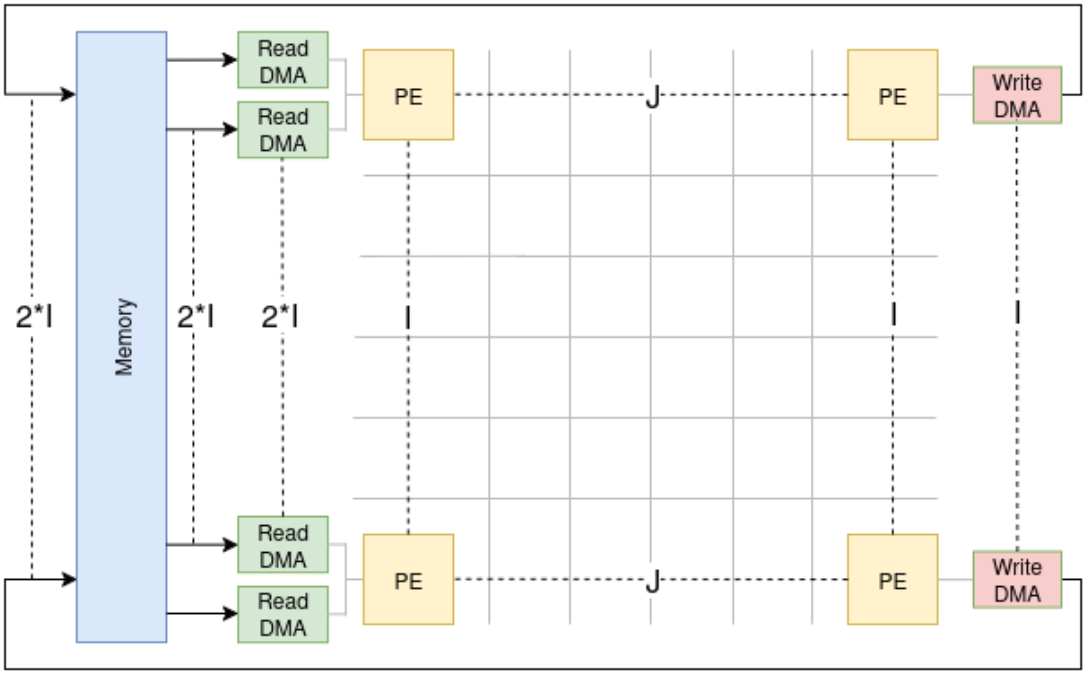

Figure 3.1: Architecture overview of the proposed CGRA for an $I \times J$ array of PEs

This accelerator includes a set of configuration registers that can be written by the host controller in order to program its operation, Also at the end of each

execution, the accelerator updates a dedicated status register that contains the number of clock cycles required to perform the programmed operation. This information can be read by the host processor and used for benchmarking and performance analysis. In addition, the internal memory is accessible to the host processor for both read and write operations. However, memory accesses from the host are only permitted when the accelerator is idle, since simultaneous access during execution could lead to conflicts.

3.2 Host processor

The accelerator is managed by a host processor implemented using the RISC-Vp++ virtual platform, developed at Johannes Kepler University (JKU). RISC-Vp++ is an enhanced version of the original Virtual Platform (VP), offering a modular and extensible SystemC-based simulation environment for architectural exploration and hardware/software co-design [5].

At its core, the platform models a RISC-V processor that acts as the central control unit. This core executes software workloads and orchestrates the operation of external components, such as custom accelerators. The simulated RISC-V core is capable of interacting with memory, peripherals, and other hardware blocks through accurately modeled system buses and interconnects.

RISC-Vp++ supports cycle-accurate simulation of processor internals (e.g., pipeline stages, memory hierarchies), as well as surrounding peripheral devices. This level of detail allows for precise evaluation of timing behavior, communication patterns, and performance bottlenecks. Furthermore, it provides a clean interface for integrating user-defined hardware modules such as the proposed CGRA into the virtual system for full system co-simulation [5].

By using this platform, the thesis leverages a realistic execution environment that closely resembles actual embedded systems, enabling early-stage validation of hardware-software interaction and performance trade-offs.

In this work, the accelerator is connected to the host processor via an AXI bus interface. This standardized protocol enables the accelerator to appear as a memory mapped peripheral [8]. Through this interface, the host processor can configure the accelerator by writing to its control registers and can also access the internal memory to load inputs and read back results.

On the software side, a dedicated C/C++ library has been developed to abstract low-level register and memory operations. The library provides definitions and structured access to registers and memory locations, in a style similar to Cortex Microcontroller Software Interface Standard (CMSIS) [9]. This approach removes the need for programmers to manually calculate addresses for configuration and data transfers. As a result, the accelerator can be controlled in a clear and

error-free way, while still being seamlessly integrated into applications running on RISC-Vp++. This setup supports benchmarking and rapid prototyping of heterogeneous architectures.

Figure 3.2 illustrates the interaction between the user code, the software libraries, the RISC-V core, and the accelerator.

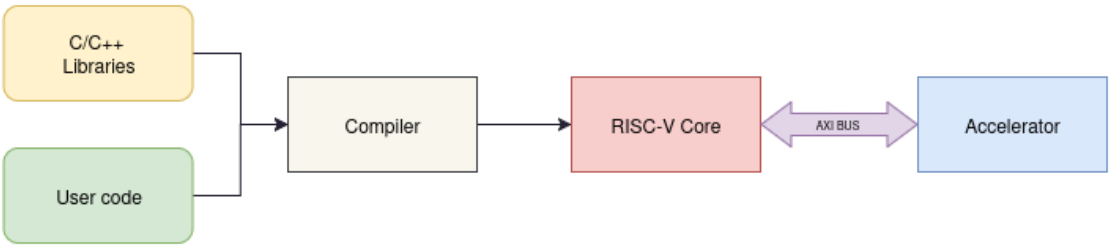

Figure 3.2: Software–hardware interaction model

Chapter 4

Design Details

This chapter provides an in-depth description of the proposed accelerator. The architecture is designed in a *generic* manner, allowing parameters such as array dimensions, datapath width, and latency to be configured at design time. The accelerator is connected to a RISC-Vp++ platform [5] through an AXI bus interface [8] [8], which enables the host to program configuration registers, control execution, and access the internal memory when the accelerator is idle.

The organization of this chapter is as follows. Section 4.1 introduces the generic configuration parameters that define the flexibility of the design. Section 4.2 describes the registers visible to the host processor, which form the programming and control interface. Section 4.3 presents the internal components of the accelerator, including the memory subsystem, DMAs, PEs, and interconnection logic. Section 4.4 analyzes the potential hazards and conflicts that may occur during operation and how they are handled.

4.1 Configuration Parameters

The accelerator is designed as a *generic architecture*, where a number of structural and functional parameters can be defined at design time. This configurability makes the system flexible, enabling exploration of different architectural trade-offs in terms of performance, resource usage, and latency. The main parameters are described below:

- **Array dimensions:** define the number of rows and columns in the array of PEs. The size of the array can be scaled according to the target application, from compact arrays for embedded systems to larger configurations for compute-intensive workloads.
- **Processing element latency:** sets the number of clock cycles between

accepting valid operands and presenting the corresponding result at the PE output. A value of zero makes the PE combinational (result in the same cycle), whereas a value of $N > 0$ inserts N pipeline stages so the result for a given input appears exactly N clock cycles later.

- **Datapath bitwidth:** determines the width of operands and results. The initial design was intended for integer arithmetic, where this parameter could be freely tuned. In the final design, however, floating-point arithmetic is employed, requiring a fixed 32-bit datapath to comply with the IEEE 754 single-precision format.
- **Memory datapath width:** defines the width of the internal memory interface. To ensure consistency with the PEs, this parameter is fixed to 32 bits in the floating-point version of the design.
- **Memory size:** specifies the total depth of the internal memory. This parameter allows scaling the memory capacity to match the storage requirements of different workloads.
- **Number of memory banks:** determines how many independent banks compose the internal memory. Multi-banked organization enables concurrent access from multiple DMAs and PEs, reducing contention and improving throughput.
- **Memory read latency:** specifies the number of cycles required to retrieve data from the memory. Although this parameter is generic, in the implemented design it is set to a single cycle, ensuring simple synchronization with the rest of the accelerator pipeline.

4.2 Register Map and Control Interface

The accelerator provides a memory-mapped control space that is divided into categories based on their function within the system. Registers are written by the host only when the accelerator is idle; once programmed, the configuration values remain stored and are applied during the next execution. Conversely, status registers and counters are read-only and provide feedback on the accelerator state.

In summary, the register space is organized into:

- **Global control and status registers**
- **Host-accessible registers of internal memory**
- **Read DMA configuration registers**

- **Array configuration registers**
- **Write DMA configuration registers**

4.2.1 Top-Level Control and Status

- **Run control:** a control bit to start a run and a busy flag that reports whether the accelerator is currently executing.
- **Performance counter:** a read-only register reporting the number of clock cycles elapsed during the last (or current) run, used for benchmarking.

4.2.2 Host Access to Internal Memory

The accelerator exposes a set of registers that allow the host to read from and write to the internal memory when the accelerator is idle.

- **Write operation:** the host specifies the target address, the data word, and the byte-enable mask through dedicated registers. A write-enable register triggers the transaction, after which the data is stored in the internal memory.
- **Read operation:** the host specifies the target address in an address register. The addressed memory word is then returned through a read-only data register.

4.2.3 Read DMA Configuration

Each row of the array integrates two independent read DMAs. For each channel, a set of registers defines how data is fetched from memory and delivered to the array during execution. The main registers are:

- **Enable:** activates the data stream so it is used in the next run.
- **Base address:** defines the starting point in memory from which the burst begins.
- **Transfer length:** specifies the number of elements that will be fetched.
- **Address direction:** indicates whether the access pattern progresses forward (increment) or backward (decrement) in memory.
- **Address increment type:** Sets the stride between successive elements, expressed as a power-of-two byte step (e.g., 1,B, 2,B, 4,B). However, in the final design, the stride is set to 4,B to match the 32-bit floating-point data format.

4.2.4 Array Configuration

The array configuration defines how PEs are activated and how they interact with incoming data streams and results during execution. It specifies the operations performed by each PE, the sources of their operands, and optional accumulation settings for iterative modes. The main configuration aspects are:

- **Output selection:** determines which results produced inside a row are forwarded to the write back DMA.
- **PE enable mask:** selects which PEs are active for the next run, allowing unused elements to be disabled.
- **PE operation selection:** specifies the computation performed by each PE, ranging from basic arithmetic operations to trigonometric, hyperbolic, exponential, or MAC functions.
- **PE operand routing:** assigns the input sources for operand A and operand B of each PE. Possible sources include data streams supplied by the DMAs and intermediate results generated by other PEs.
- **PE MAC iteration count:** defines the number of accumulation cycles executed when a PE operates in MAC mode.

4.2.5 Write DMA Configuration

Each row includes a dedicated write DMA channel that transfers results produced by the array back to memory. The configuration of this channel defines where the results are stored and how the memory access pattern is performed. The main registers are:

- **Enable:** activates the write-back stream so that results generated during the next run are committed to memory.
- **Base address:** specifies the starting address in memory where the first result will be written.
- **Transfer length:** determines how many elements will be stored in memory.
- **Address direction:** selects whether consecutive writes progress forward (increment) or backward (decrement) in memory.
- **Address increment type:** Defines the stride between successive writes, expressed as a power-of-two step in bytes (e.g., 1 B, 2 B, 4 B). In the final design, this stride is fixed to 4 B to match the 32-bit floating-point data format.

4.3 Components of the Accelerator

The proposed accelerator is composed of a set of hardware components that cooperate to execute compute intensive workloads. Each component fulfills a specific role in the dataflow, from fetching operands to computation and writing results back to memory. Together, these blocks form a flexible and configurable architecture that supports parallel execution and efficient data reuse.

The following subsections provide detailed descriptions of each component.

4.3.1 Internal Memory

The architecture employs a dedicated internal memory subsystem to provide operands and store results for the PEs. This memory is essential to sustaining high throughput by reducing dependence on external memory and enabling fast data access close to the computation array. It is designed to be generic, supporting configurability in size, width, latency, and banking structure, making the accelerator adaptable to different design points.

Configurable Parameters

The internal memory is parameterizable to accommodate various application requirements. The main configurable aspects include:

- **Number of banks:** determines how many independent banks compose the memory, directly affecting parallelism.
- **Total capacity:** defines the overall storage size of the memory array, expressed in words.
- **Word width:** specifies the datapath granularity, i.e., the number of bits per stored element.
- **Access latency:** defines how many clock cycles are required for a read operation; the design supports both zero-latency (combinational) and pipelined access modes.
- **Initialization:** each bank is backed by a text file used to preload its contents and update them during execution, enabling reproducible simulation runs and traceability.

Interface Registers

Each bank is accessed through a standard interface. The interface provides:

- Global synchronization via clock and reset.
- Request and enable signals to initiate read or write operations.
- Address, data input, and data output channels.
- Byte-level enable lines to support partial-word updates.
- Status signals to indicate when a request cannot be served because the target bank is already in use.

Banked Organization

The internal memory is physically divided into multiple banks, each representing a subset of the overall address space. By splitting the memory into independent banks, multiple data accesses can proceed in parallel as long as they target different banks, which increases overall throughput compared to a single monolithic memory.

For simulation purposes, each bank is mirrored into a dedicated text file, with one line per memory word (4 bytes). This format makes it straightforward to inspect or modify the memory contents with standard tools, thereby improving transparency during debug and verification. At system initialization, every bank can either be loaded with predefined data (for example, test vectors or input datasets) or instantiated as empty if no file is provided. This flexibility enables controlled experiments, reproducibility of test cases, and easier integration with software-driven workloads.

Arbitration and Conflicts

To fully exploit the banked structure, the memory subsystem exposes one logical access port per bank. From the perspective of the requesters (e.g., DMAs), any bank can be targeted from any port, but within a single cycle only one access per bank can be granted. If two or more requests simultaneously address the same bank, a conflict arises. In this case, an arbitration mechanism resolves the contention by assigning priority to one request while stalling the others. The stalled requesters receive a signal indicating that their access must be retried in a subsequent cycle.

This arbitration also ensures correctness by preventing simultaneous writes or overlapping reads/writes to the same bank, which could otherwise lead to data corruption. At the same time, it preserves parallelism: as long as requests are distributed across different banks, they can be served concurrently without interference. In practice, this scheme balances simplicity of control with efficient utilization of memory bandwidth, making it well-suited for streaming and parallel workloads mapped onto the array.

4.3.2 Read DMA

The DMA unit streams operands from internal memory into the processing elements (PEs) at the row level. It autonomously issues read requests based on a programmable address sequence and transfers the fetched data to the compute array, while handling back-pressure from both the memory system and the consuming PEs.

This mechanism effectively decouples data movement from computation: the DMA can prefetch data from memory while the PEs are still processing previous data. As a result, memory access and computation can proceed in parallel, reducing idle time and improving overall throughput and resource utilization.

Configurable Parameters

- **Addressable depth:** defines the portion of memory that can be accessed for reads.
- **Datapath width:** sets the width of the data bus, controlling how many bits are transferred per access.

Interface Signals

- Global synchronization signals (clock and reset).
- Enable input to start a new burst when idle.
- Base address, specifying the starting point of the read sequence.
- Transfer length, defining the number of elements to be fetched.
- Address direction flag, selecting incrementing or decrementing sequences.
- Address step encoding, where the effective step size is 2^{step} bytes.
- Request signal toward memory, asserted when a new read must be issued.
- Address output specifying the memory location of each read.
- Read-data input from memory, carrying the fetched word.
- Data output toward the PEs, accompanied by a validity flag.
- Back-pressure input from memory, signaling that the target bank is busy.
- Back-pressure input from the consumer side, signaling that no new data can be accepted.

- Stall signal toward memory or producer, asserted when the DMA cannot issue new transfers.
- Busy flag, asserted during burst startup and while transfers are in progress.

Functionality

- **Burst activation.** A burst begins when the enable input is asserted while idle and no stalls are present. The first request is issued at the base address, and the DMA transitions to active mode.
- **Address generation.** During an active burst, successive addresses are generated incrementally or decrementally based on the address direction. The stride between accesses is determined by the step encoding.
- **Data path.** Data words returned from memory are forwarded to the PEs, with validity signals ensuring proper synchronization.
- **Burst completion.** A counter tracks the number of elements fetched. When the transfer length is reached, the DMA ends the burst and returns to idle.
- **Back-pressure handling.**
  - If the memory bank is busy, requests are withheld until the stall clears, preserving the current address.
  - If the consumer side stalls, the DMA holds the fetched data valid until it can be accepted.
- **Status reporting.** The busy signal indicates overall activity (from burst start to completion), while the stall signal reports temporary inability to fetch or deliver data.

4.3.3 Processing Element (PE)

The PE executes floating-point operations on one or two operands and produces a result with a configurable latency. It supports arithmetic (ADD, SUB, MUL, MAC), trigonometric (direct and inverse), hyperbolic (direct and inverse), and exponential functions. The PE interfaces use simple handshaking and back-pressure signals so that data movement via the row DMAs can be decoupled from computation. Internally, results are represented in single-precision format using a 32-bit raw encoding.

Configurable Parameters

- **Datapath width:** defines the width of operands and results. In this design it is fixed to 32 bits, corresponding to single-precision floating-point raw encoding.
- **Pipeline depth:** sets the number of pipeline stages. A value of zero makes the PE combinational, while higher values insert pipeline registers for result and validity.

Interface Registers

- Global synchronization signals (clock and reset).
- Enable input to activate or deactivate the PE.
- Operand inputs (A and B) as raw 32-bit values.
- Operand availability flags (Ra and Rb) indicating when inputs are valid.
- Output back-pressure signal (Ro), which stalls the PE when the consumer is not ready.
- Operation selector (Sel), defining which arithmetic or transcendental operation is executed.
- Iteration counter input for MAC operations.
- Result output containing the computed value.
- Validity output (V) marking when the result is valid.
- Stall feedback signals (Sa, Sb) toward the operand producers, requesting them to pause.

Functionality

- **Operand model.** Arithmetic operations require both operands, while single-operand functions only require one. Missing operands are requested through stall feedback signals.
- **Back-pressure.** If the consumer of the result is not ready, the PE halts new outputs and stalls its input sources until the consumer resumes.
- **Operation domains.** Trigonometric inputs for sin and cos are interpreted in degrees, and inverse trigonometric results are returned in degrees. Domain checks are applied for arcsin, arccos, and arctanh to prevent invalid inputs.

- **MAC mode.** When configured for MAC, the PE performs iterative accumulation of $A \times B$ products for the programmed number of iterations, outputting the result at the end.
- **Latency behavior.**
  - *Combinational mode:* results are produced in the same cycle as operands are accepted.
  - *Pipelined mode:* results propagate through shift registers and emerge after the configured number of pipeline stages.
- **Data format.** All operands and results are handled as 32-bit raw single-precision values, with computations performed in floating point.

Instantiation in PEUnit

Each PE is instantiated within a Processing Element Unit (PEUnit), which wraps the PE with operand multiplexers. This allows operands to be dynamically selected either from the row's DMA inputs or from the outputs of other PEs in the array, enabling a fully configurable dataflow.

Additional Parameters

- **Array dimensions:** define the number of potential input sources available to each PE.
- **Multiplexer size:** equal to (rows $\times$ columns) $+ 1$, covering all PEs plus the DMA input.

Additional Interface Registers

- Vectors of candidate operand inputs from all other PEs and DMA streams.
- Validity flags corresponding to each candidate operand.
- Selection inputs for multiplexers to route the chosen operand to the PE.

4.3.4 Write DMA

The Write DMA streams results from the PEs back into internal memory. It autonomously issues write requests, follows a programmable address sequence, and respects back-pressure from both the memory system and the producer side. By decoupling data movement from computation, it allows the array to overlap the compute and store phases, thereby improving throughput and resource utilization.

Configurable Parameters

- **Addressable depth:** defines the total memory space that can be targeted for writes.
- **Datapath width:** sets the width of the write data bus.

Interface Signals

- Global synchronization signals (clock and reset).
- Enable input to start a new burst when idle.
- Address direction flag, selecting incrementing or decrementing sequences.
- Transfer length, defining the number of elements to be written.
- Base address, used as the starting point of the burst.
- Address step encoding, where the effective step size is 2^{step} bytes.
- Input data from the producer side (results of PEs).
- Back-pressure input from memory, signaling that the target bank is busy.
- Back-pressure input from the producer, signaling that no new data are ready.
- Write data output forwarded to memory.
- Address output specifying the destination of each write.
- Request signal toward memory, asserted when a new write must be issued.
- Stall signal toward the producer, asserted when memory or the DMA is not ready.
- Busy flag, asserted during burst startup and while transfers are in progress.

Functionality

- **Burst activation.** A burst begins when the enable input is asserted while idle and no stalls are present. The first issued address is the base address, and an internal flag marks the DMA as active.
- **Address generation.** During active bursts, the next address is computed incrementally or decrementally depending on the address direction. The step size is determined by the encoded stride parameter.

- **Data path.** The write data output directly forwards the producer data, ensuring alignment with issued write requests.
- **Handshake protocol.** After each successful write beat, the DMA enforces a one-cycle idle gap to avoid consecutive hazards on the memory interface.
- **Burst completion.** A counter tracks the number of elements written. When the programmed length is reached, the burst ends and the DMA returns to idle.
- **Back-pressure handling.**
  - If memory is busy, requests are withheld, and the current address is preserved until the stall clears.
  - If the producer stalls, requests are paused.
- **Status reporting.** The busy signal reflects overall activity (from burst start to completion), while the stall signal indicates temporary inability to accept or issue transfers.

4.3.5 Interconnection Logic

The interconnection logic implements the dataflow backbone [**liu2019cgra**] of the accelerator, linking PEs, read DMAs, and write DMAs into a coherent compute fabric. It is organized as a mesh topology, where each PE can access operands not only from its row-level read DMAs but also from the outputs of other PEs. This flexibility allows results produced in one part of the array to be reused in another without committing them to memory, thereby improving performance and reducing memory bandwidth pressure.

Operand selection is achieved through multiplexers placed at the inputs of each PE. These multiplexers choose between DMA-fed operands and forwarded results from other PEs, providing fine-grained configurability at the operand level. Similarly, each row integrates a result multiplexer that selects which PE output is routed to its corresponding write DMA, enabling dynamic mapping of computation results back to memory.

Back-Pressure Propagation (*Ro* wiring). The output back-pressure input of each PE (*Ro*) is not programmed via registers; instead, it is derived dynamically from the current operand-routing configuration. The array controller inspects the operand selection signals of every PE and routes the corresponding stall feedback to the producer side as follows:

- **PE→PE flow:** If a consumer PE selects another PE as the source for operand A (resp. B), the consumer's output stall signal for that operand (*Sa* or *Sb*) is forwarded to the selected producer's *Ro*. This causes the producer to pause result generation until the consumer is ready again, ensuring lossless handshaking across the mesh.

- **DMA→PE flow:** If a consumer PE selects a row Read DMA as the source for operand A (resp. B), the consumer's *Sa* (resp. *Sb*) is forwarded to the corresponding row DMA back-pressure input, stalling the DMA stream when the consumer cannot accept new data.

- **Row write-back path:** For each row, a result multiplexer selects which PE drives the Write DMA. When the Write DMA signals stall (e.g., due to a busy memory bank), that stall is propagated to the currently selected producer PE by asserting its *Ro*. If the row's Write DMA is disabled, no *Ro* is asserted from the write-back path.

- **Self-loop guard:** When a PE selects its *own* output as an operand source (feedback), the routing logic explicitly avoids feeding its *Sa*/*Sb* back into its own *Ro*. This prevents combinational back-pressure loops and ensures forward progress (no deadlock due to self-referential stalls).

- **Enable-aware routing:** Stall propagation only considers active producers and consumers. If a consumer or the row Write DMA is disabled, the corresponding *Ro* lines toward potential producers are deasserted.

This dynamic *Ro* wiring implements back-pressure at run time based solely on the operand selection state, requiring no dedicated configuration registers. It guarantees that stalls always propagate from consumers to the actual, currently selected producers, while avoiding self-feedback loops hazars.

4.3.6 Host Accessibility to Internal Memory

The internal memory exposes a dedicated host port that is time-multiplexed between the host processor and the accelerator fabric. Ownership of this port is switched by the accelerator's run control so that software can safely load inputs and read back results without interfering with on–going DMA traffic.

Access Model

- **Idle phase (host ownership).** When the accelerator is idle, the host port is driven by the host side. The host supplies the address, write data, write request, write enable, and byte enables; the corresponding read data is returned

on a read-back register. This enables memory initialization, parameter loading, and result dumping.

- **Run phase (accelerator ownership).** When a run starts, ownership of the host port is transferred to the accelerator fabric. During this phase, the port carries the array's internal traffic (row DMAs and write-backs). Host accesses are blocked to avoid conflicts with compute traffic.

4.3.7 Accelerator Controller

The accelerator controller manages the main tasks needed to run the array. It loads and writes the configuration, starts and stops execution, arbitrates read/write traffic toward the internal memory, and measures how long the execution takes in clock cycles. The controller also enables the sub-blocks that are active for the current run.

Run–State Machine and Enables

The controller implements a two-state finite-state machine:

- **IDLE:** the accelerator is quiescent. Configuration registers are visible and can be written by the host. When the run command is asserted, the controller:

  1. raises the global busy flag,
  2. It sends enable signals only to the read DMAs, write DMAs, and PEs that are activated in the configuration.
  3. clears and starts the cycle counter.

- **RUN:** the accelerator executes the programmed kernel. The controller increments the cycle counter every clock. Completion is detected when no enabled write DMA reports activity (i.e., all write-backs have drained). At that point the controller

  1. latches the final cycle count into a status register,
  2. deasserts the busy flag,
  3. returns to **IDLE**.

This sequencing ensures that only the intended subset of engines is activated per run and that the host observes a stable configuration interface while the accelerator is idle.

Performance Counter

A free-running cycle counter is cleared at run start and increments every cycle while in **RUN**. Upon completion it is copied into a readable status register, reporting the exact number of cycles required for the programmed operation. This supports deterministic benchmarking and simplifies performance analysis.

Memory Request Arbitration

The accelerator controller arbitrates access to the internal memory between read streams (operand fetch) and write-back streams (result commit). Since both operations may target the same memory port in the same cycle, a clear policy is required to guarantee correctness and sustained throughput.

In the adopted scheme, each row has one memory port that is shared between its read DMAs and its write DMA. Write operations are given priority whenever data is ready, while read operations continue steadily whenever no write is pending. This behavior is defined by the following rules:

- **Default path (reads):** When there is no pending write request, the memory port is assigned to the read DMA channel. This ensures continuous operand fetching, maintaining throughput and preventing the compute pipeline from stalling due to lack of input data.
- **Write takeover:** whenever the write DMA presents a valid write beat, the controller grants that request immediately, overriding the read path for one cycle. After the write completes, control reverts to the read path. This mechanism effectively assigns *beat-level priority* to writes.
- **Busy propagation:** if the addressed memory bank is busy, the stall condition is propagated back to the requester. During read cycles, the read DMA receives the stall indication; during write cycles, the write DMA sees its own busy handshake asserted. In both cases, engines hold their current address and avoid over counting until the stall clears.
- **Fairness:** after each successful write, one read cycle is granted if there are pending read requests. If further write data is available, control switches back to the write DMA. This ensures that producer results are never lost, while the read path continues to make progress between consecutive writes.

4.4 Hazards and Conflict Handling

As discussed in the previous sections, the proposed accelerator has several classes of hazards due to its highly parallel nature. These hazards stem from concurrent

accesses to shared resources, arbitration of operand routing, pipeline timing effects, and the interaction between multiple producers and consumers. This section consolidates all identified hazard types and outlines the mechanisms employed to handle them, providing a comprehensive view of how correctness and performance are maintained across the architecture.

4.4.1 Memory Conflicts

The internal memory is organized as a multi-bank structure to sustain parallelism. However, since each bank can only serve one access per cycle, conflicts occur when multiple DMAs request access to the same bank simultaneously.

- **Bank conflicts.** When two or more ports target the same bank in the same cycle, only the request with the lowest priority index is granted. The other ports receive a `Busy` signal and must retry in later cycles.
- **Read/Write arbitration.** Each memory bank can be accessed by both a Read DMA and a Write DMA, but not at the same time. To prevent simultaneous access, a controller arbitrates between the two using a fixed-priority scheme [7]. This ensures deterministic behavior, one DMA is always granted access, while the other is temporarily stalled and automatically resumes once the bank becomes available.

4.4.2 DMA Hazards

DMAs introduce hazards if requests are not properly synchronized with producers and consumers.

- **Read DMA stalls.** A Read DMA must pause when either memory is busy or the consuming PE row signals back-pressure. Without this, operands could be overwritten or skipped. The stall handling logic freezes the current address and retries in the next available cycle.
- **Write DMA stalls.** A Write DMA must pause when either memory is busy or the producing PE row does not provide valid data. To maintain correct alignment between addresses and data, the Write DMA inserts a controlled one-cycle gap after each successful write.

4.4.3 Processing Element Hazards

Within each PE, hazards may occur if operands or results are not available at the right time.

- **Operand hazards.** If only one of the required operands is present, the PE asserts back-pressure (`Sa` or `Sb`) to request the missing one. This prevents partial or invalid computations.
- **Output hazards.** If the consumer of the PE result is not ready, the PE uses the `Ro` signal to stall its pipeline and avoid producing unconsumed results.

4.4.4 Interconnection Hazards

The mesh interconnection allows processing elements (PEs) to forward results either to other PEs or to DMA streams. Hazards may occur if routing is not properly constrained, such as when multiple consumers attempt to use the same producer output, or when a PE's output is fed back into its own input.

In the proposed design, the following rules are imposed to avoid such hazards:

- **Fixed routing during execution.** The configuration of the result multiplexers is established before a run begins and remains constant until completion. Dynamic changes during execution are not supported.
- **Single-consumer rule.** The output of each PE or DMA must be connected to exactly one destination. Connecting a output to multiple consumers is forbidden, as this would create deadlock situations.

4.4.5 Host Access Hazards

The host processor shares access to the internal memory with the accelerator fabric.

- **Simultaneous access.** To prevent conflicts between host and accelerator, host access is only allowed while the accelerator is idle. During execution, the memory interface is exclusively reserved for DMA traffic.
- **Consistency hazards.** By enforcing this separation, the design guarantees that the host always observes a consistent memory state and that DMA transfers are never corrupted by external writes.

Through a combination of arbitration, back-pressure signals, pipeline control, and mutual exclusion with the host, the proposed accelerator ensures hazard-free operation while sustaining high parallelism. These mechanisms guarantee correctness even under heavy contention, at the cost of occasional stalls when conflicts cannot be avoided.

Chapter 5

Integration With RISC-Vp++

The accelerator is integrated into the RISC-Vp++ platform as a memory mapped AXI slave [5]. A thin SystemC/TLM transactor terminates the Advanced eXtensible Interface 4 (AXI4) protocol [8] and bridges register and memory requests to the accelerator's native control and host memory ports. This chapter describes the bus attachment, address decoding, timing adaptations of memory read latency, and the mapping between software visible registers and internal accelerator registers.

5.1 Address Map and Decoding

Each incoming access is decoded against two disjoint regions:

- **Register window:** accesses outside the internal memory window are directed to the software register map, organized as a word addressed dictionary. This region contains all control and status information, including run control, DMA descriptors, array routing, PE configuration, and performance counters.

- **Internal memory window:** a contiguous region [`mem_base`, `mem_base` + `mem_size`) is reserved for the accelerator's internal multi bank memory. When the target address falls in this range, the access bypasses the register map and is instead directed to the host memory interface of the accelerator, allowing read and write operations on the internal memory banks.

5.2 Write Path (AXI → Accelerator)

When a write request is received from the host, it is handled based on the target address range:

- **Configuration writes:** If the request targets a control register, the write is applied immediately. An AXI write response is returned on the next clock cycle's negative edge.
- **Internal memory writes:** If the request targets the accelerator's internal memory, the address is translated to a word-aligned index. The write operation is then issued to the memory port, along with control signals indicating a valid request and write operation. All bytes in the word are written simultaneously. The write completes in a single cycle at the accelerator boundary, and the AXI write response is sent on the following negative edge.

5.3 Read Path and Latency Alignment

Read requests from the host are handled differently depending on the target region:

- **Configuration reads:** If the request targets a control register, the corresponding value is returned immediately, with no additional latency.
- **Internal memory reads:** If the request targets the internal memory, the memory read is issued through the accelerator interface using a valid read request. A small pipeline is used to simulate memory latency. When a read is initiated, a pending read record is created and tracked over multiple cycles. After a fixed number of cycles corresponding to the modeled memory latency the read data is captured and returned to the host. The AXI read response is then issued on a negative clock edge, aligning with the expected timing behavior.

This approach ensures that the latency observed by the AXI interface accurately reflects the accelerator's internal memory timing, preserving cycle level accuracy at the boundary.

5.4 Programming Model Summary

From the software point of view, interacting with the accelerator involves a simple, deterministic sequence of steps:

1. **Load inputs:** While the accelerator is idle, the host processor has access to the internal memory. During this phase, input data is written into the memory window that the accelerator will later read from. This memory mapped region acts as the data staging area for the computation.

2. **Configure:** Before execution, the software configures all accelerator parameters through the control register interface. This includes setting up:

   - DMA descriptors to define memory transfer patterns,
   - Interconnection and routing within the array,
   - Operations performed by each PE,
   - Output selection and control for storing results.

   This step ensures the accelerator behaves deterministically for the given workload.

3. **Run:** Once configuration is complete, execution is triggered by setting the enable bit in the control interface. At this point, the accelerator gains ownership of the host port and begins processing. Memory transfers and computation proceed according to the previously programmed settings, operating autonomously without further software intervention.

4. **Wait for completion:** The host can monitor progress by polling a busy status bit or waiting for an interrupt or event. Upon completion, a performance counter register provides the number of cycles taken, which can be used for profiling or optimization purposes.

5. **Read back results:** Once the accelerator has completed execution and becomes idle, host has access to internal memory. The host can detect this state by checking the accelerator's *busy* flag. When the flag is cleared, it indicates that the computation has finished and the results are ready to be retrieved.

Chapter 6

Results

This chapter presents the evaluation of the proposed accelerator through two main tests. The first is a functional test aimed at verifying the correct operation of the PEs, DMAs, and memory system. The second test focuses on mapping a Fast FFT algorithm onto the array in order to evaluate its computational capabilities on a representative workload.

Both tests are conducted on a prototype instance of the architecture featuring a 4×4 array of PEs. The physical layout of this array is shown in Figure 6.1. These experiments validate the accelerator's ability to execute diverse operations, handle data routing correctly, and interact efficiently with memory and DMA units.

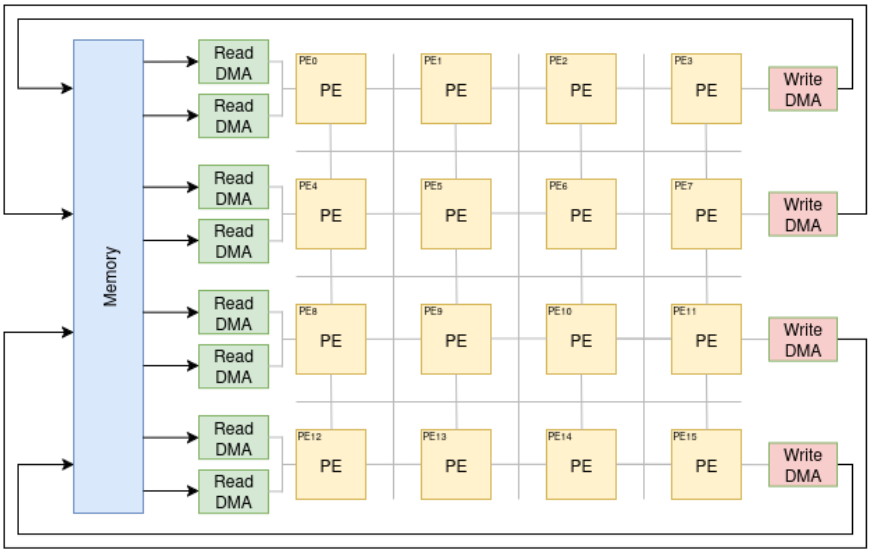

Figure 6.1: Prototype instance with a 4×4 PE array

6.1 Test 1: Baseline Functional Test

The first experiment serves as a baseline configuration to validate the functionality of the interconnection fabric, operand routing, and operation selection within the PEs.

In this setup, the read DMAs of rows one, two, and three are enabled to fetch data from memory. Each read DMA retrieves 8 data values per transfer, which are delivered to PE0, PE4, and PE8. These PEs are configured to perform addition on their respective input operands.

The results from PE0 and PE4 are forwarded to PE1, which performs a multiplication. The output of PE1 is then combined with the result of PE8 in PE13 using a MAC operation and the iteration value for this PE is 4. The resulting value is further processed in PE14 with a sine function, followed by PE15, which applies the arcsine. The final result is written back to memory via the write DMA of row three.

This test exercises a diverse set of operations to verify the correctness of handshaking between interconnected PEs, the selection of functional units, and the reliability of result write-back to memory.

The full execution including data fetch, computation, and write-back completes in 16 clock cycles. This includes the latency associated with data movement, PE operations, and synchronization across the array.

Figure 6.2 illustrates the configuration and dataflow across the 4×4 array. The interconnections between components are highlighted in red.

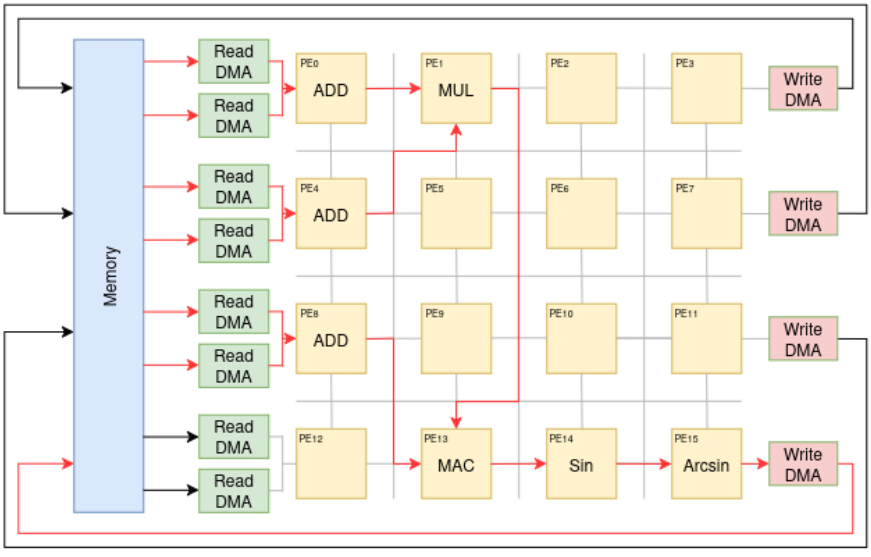

Figure 6.2: Baseline functional test dataflow over the 4×4 array

6.2 Test 2: FFT Mapping Test

The second experiment evaluates the computational capabilities of the proposed accelerator by mapping a single butterfly computation stage from a Decimation In Time (DIT) FFT algorithm onto the processing array. This test is designed to demonstrate the architecture's ability to handle a realistic, computation-intensive workload that features both parallelism and data reuse—characteristics that align well with the strengths of a CGRA-style accelerator.

6.2.1 FFT Butterfly Computation Overview

The core operation in the DIT FFT algorithm is the *butterfly*, a dataflow pattern that combines two complex inputs into two outputs using a mix of complex additions, subtractions, and multiplications with precomputed coefficients called *twiddle factors*.

In this implementation, the complex arithmetic required by the FFT is broken down into simpler real-valued operations. This transformation is necessary because the PEs in the array are designed to perform only real arithmetic operations, such as addition, subtraction, and multiplication.

To make the complex butterfly computation compatible with the hardware, each complex operation is rearranged into a sequence of real-valued multiplications and additions that produce the same mathematical result. For example, a complex multiplication involves four real multiplications and two additions, which can be individually mapped to the available PEs. By restructuring the computation in this way, the algorithm takes full advantage of the hardware's capabilities without requiring dedicated complex arithmetic support.

Given complex input pairs A, A' and B, B', along with twiddle factors T and T', the butterfly performs the following operations:

$$P = B \cdot T - B' \cdot T' \tag{6.1}$$
$$P' = B' \cdot T + B \cdot T' \tag{6.2}$$
$$R = A + P \tag{6.3}$$
$$R' = A' + P' \tag{6.4}$$
$$Q = A - P \tag{6.5}$$
$$Q' = A' - P' \tag{6.6}$$

This rearrangement preserves the mathematical correctness of the FFT but allows the mapping to leverage the CGRA's word-level compute model, avoiding unnecessary complexity related to handling full complex numbers.

6.2.2 Mapping on the PE Array

The butterfly computation is mapped onto a 4×4 instance of the accelerator array, which consists of 16 PEs arranged in 4 rows. For this test, only 6 PEs are used. This minimal usage is intentional: it demonstrates how an efficient schedule can be devised even under constrained hardware resources.

The 6 selected PEs are responsible for executing all the arithmetic operations required for one butterfly instance.

Stage 1: Computing P and P' In the first stage, inputs B, B', and their corresponding twiddle factors T, T' are read from memory via the available read DMA channels. Each read DMA fetches 4 data elements per request. These inputs are routed to the PEs configured for multiplication:

- Four PEs (indices 0, 4, 8, and 12) perform the real multiplications.
- Two additional PEs (indices 1 and 9) compute the addition and subtraction between the partial products to produce P and P'.

Due to the limited number of read DMA channels (8 total, 2 per row), there is insufficient bandwidth to simultaneously fetch all operands, including A and A', in the same cycle. Therefore, the intermediate values P and P' are written back to internal memory for use in the next stage.

For a set of 8 data inputs (i.e., 4 butterflies), this first stage requires **10 clock cycles** to complete, including data fetch, computation, and memory write-back. Figure 6.3 illustrates the dataflow.

Stage 2: Computing R, Q, R', Q' After storing P and P', the second stage computes the final FFT outputs using A, A', P, and P'. The stage consists of four arithmetic operations for each butterfly instance:

$$R = A + P, \quad Q = A - P$$
$$R' = A' + P', \quad Q' = A' - P'$$

Two execution strategies were explored:

1. **Single-Stage Execution with Memory Bank Conflicts:** All outputs are computed in a single execution window using the same 6 PEs. However, this causes multiple operands to be read from the same memory banks at the same time, leading to conflicts. These conflicts serialize memory accesses and increase latency. This version takes **20 clock cycles** to complete for 8 data inputs.

   Figure 6.4 illustrates the access conflict in this approach.

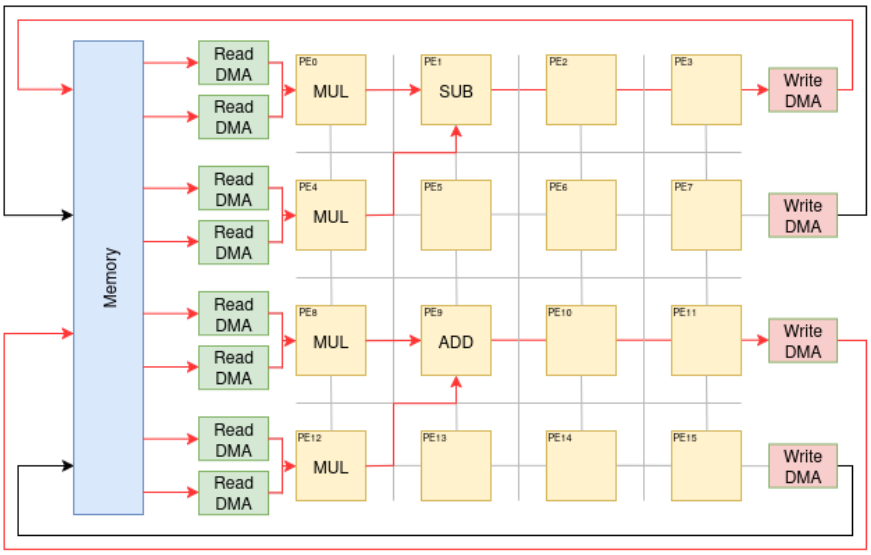

Figure 6.3: Stage 1: FFT butterfly mapping for computing P and P'

2. **Two-Stage Conflict-Free Execution:** To avoid simultaneous access to the same memory banks, the computation is split into two sub-stages:

   - First, $R = A + P$ and $R' = A' + P'$ are computed.
   - Then, $Q = A - P$ and $Q' = A' - P'$ are computed.

   By splitting the stage, bank conflicts are completely eliminated, resulting in more efficient execution. Each sub-stage takes **10 clock cycles**, totaling **20 cycles** for the full operation—but without conflict penalties. Figures 6.5 and 6.6 show the two steps.

6.2.3 Execution Time Summary

Table 6.1 summarizes the total number of clock cycles required to process 8 data points (i.e., 4 butterfly operations) using different execution strategies.

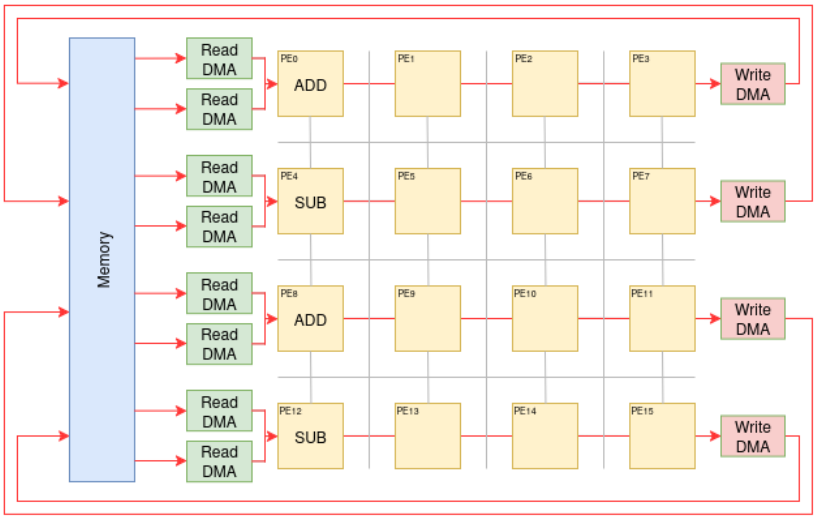

Figure 6.4: Single-stage execution with memory bank conflict during concurrent access

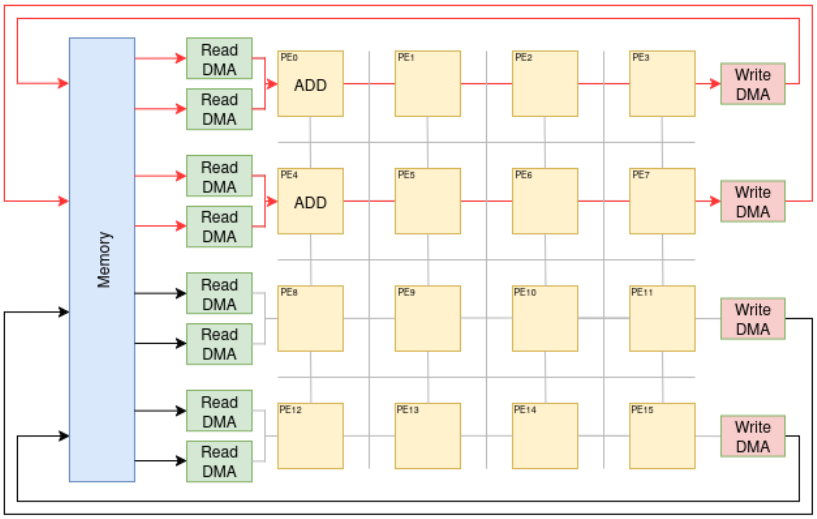

Figure 6.5: Stage 2a: Computing R and R' (conflict-free)

Table 6.1: Execution time (in clock cycles) for different FFT mapping strategies

Execution Strategy	**Total Clock Cycles**
Stage 1 (compute P, P')	20
Stage 2 (single-stage with conflicts)	20
Stage 2 (conflict-free two-stage)	$9 + 9 = 18$
Total (with conflicts)	**40**
Total (conflict-free)	**38**

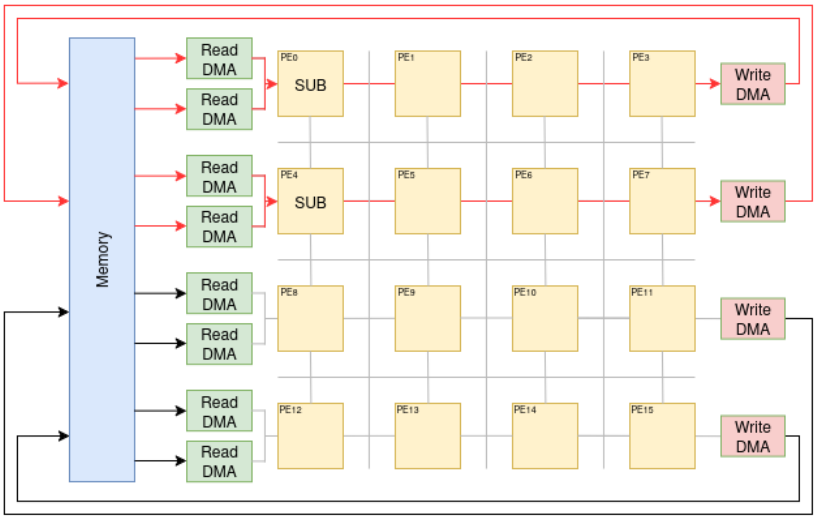

Figure 6.6: Stage 2b: Computing Q and Q' (conflict-free)

The conflict-free version completes the operation in 2 fewer cycles than the single-stage strategy with memory bank conflicts. While the difference is small in this test case, it demonstrates the advantage of avoiding concurrent memory access contention. As the number of butterfly units increases or as the array becomes more densely utilized, this advantage is expected to grow, making conflict-free scheduling a more scalable and performance-efficient strategy.

Chapter 7

Conclusion

This thesis presented the design, implementation, and validation of a reconfigurable accelerator architecture based on a word-level, coarse-grained processing array. The architecture features a scalable grid of floating-point PEs connected via a programmable mesh interconnect and supported by a multi-banked memory system accessed through autonomous DMA units. The design aims to balance flexibility and efficiency for accelerating structured, dataflow-oriented workloads, such as those found in scientific computing and ML/AI applications.

The accelerator was developed as a generic, parameterizable architecture with configurable datapath width, pipeline depth, memory latency, and array dimensions. It supports a rich instruction set, including arithmetic, transcendental, and MAC operations, while maintaining a regular and lightweight interconnection model. Integration with the RISC-Vp++ virtual platform was achieved through a SystemC/AXI interface, enabling full system simulation and host controlled benchmarking.

To validate the system, two experiments were conducted. The first was a functional test verifying core data routing and chaining across PEs, as well as DMA driven operand transfers and memory interactions. The second experiment mapped a single stage of a FFT onto the array. This test demonstrated the ability of the architecture to handle real, computation-intensive workloads, while also highlighting the impact of memory bank conflicts and how execution strategies can be adapted to mitigate them.

The evaluation shows that the proposed accelerator successfully delivers correct, deterministic, and configurable execution, with a modular structure that supports future enhancement. It forms a solid baseline for continued research into efficient reconfigurable computing.

7.1 Future Work

Several directions are envisioned to extend the capabilities of the current design:

- **FIFO-based buffering:** Introducing FIFO queues between components can reduce stalling due to timing mismatches, allowing better overlap of memory and compute phases, and improving pipeline utilization.
- **Compiler integration:** Developing a compiler to automatically map high-level kernels onto the array by configuring the PEs, interconnect routing, and DMA scheduling would enable broader applicability and reduce programmer effort.
- **Multi-destination operand routing:** Enhancing the interconnect to allow a PE's output to be routed to multiple consumers would increase data reuse opportunities and enable more complex graph based computations.

These improvements aim to boost performance, programmability, and flexibility, making the architecture suitable for a wider range of domain specific applications in AI, scientific computing, and embedded systems.

Appendix A

Automation Scripts and Usage

This appendix describes the Python script `SyncExec.py`, which automates building, running, and testing the accelerator design either locally or on a remote server. The script simplifies simulation workflow by managing file transfers, compiling with SystemC, running the testbench, and collecting logs.

A.1 Overview

The script supports two primary modes:

- **Local mode**: Runs simulations directly on the local machine using a SystemC-based Makefile.
- **Remote mode**: Uploads files to a remote server via SSH, runs the build remotely, executes the simulation, and logs the output.

A.2 Configuration via JSON File

The behavior of the script is driven by a configuration file located at `sim/config.json`. This file allows users to easily specify:

- Remote server credentials and target directory
- Testbench type and test case number
- Accelerator architecture parameters such as array dimensions, bitwidth, and interconnection type

- Memory subsystem configuration including size, number of banks, latency, and memory file paths

A.3 Usage

A.3.1 Run Locally

```
python SyncExec.py -local
```

A.3.2 Run on Remote Server

```
python SyncExec.py -server
```

A.3.3 Run and update a Specific File

```
python SyncExec.py -server -file sim/main.cpp
```

A.3.4 Exclude Files and Folders to upload

```
python SyncExec.py -server -ignore_files temp.txt -ignore_folders
   __pycache__
```

A.3.5 View Help

```
python SyncExec.py -server -help
```

A.4 Script Features

- Reads the configuration file and extracts simulation parameters
- Establishes a secure SSH connection to the server (if in remote mode)
- Uploads required files using SFTP
- Executes simulation via SystemC `make` commands
- Collects and logs output into a local log file for review

A.5 Dependencies

The script requires the Python package `paramiko` for SSH and file transfer:

```
pip install paramiko
```

Appendix B

Memory Initialization Script

This appendix describes a Python script used to initialize the accelerator's internal memory banks with random floating-point data for simulation purposes. This is particularly useful for testing, benchmarking, and validating the correctness of memory access and dataflow functionality across the processing array.

B.1 Purpose

The script generates random single-precision floating-point numbers in the range `[0.5, 8.5]` and converts them into raw 32-bit IEEE 754 hexadecimal format. These values are then distributed across multiple text files—one per memory bank. Each file corresponds to a memory bank used by the accelerator's simulation model and is later loaded during simulation to provide initial data to the compute array.

B.2 Usage

Run the script from the project root directory using:

```
1 python3 init_memory.py
```

This will:

- Create the directory `sim/` if it does not already exist.
- Generate one file per memory bank (`Bank0.txt`, `Bank1.txt`, ..., `Bank7.txt`).
- Each file contains a list of hex-formatted 32-bit float values, one per line.

B.3 Example Output Format

Each line of the output file contains one 32-bit float value encoded as an 8-digit hexadecimal number. For example:

```
3F19999A
41200000
40A66666
...
```

These values correspond to raw IEEE 754 encodings and are directly used by the simulator to preload memory contents.

B.4 Customization

To change the value range, bank count, or memory size, modify the following constants at the beginning of the script:

- `NUM_BANKS` – number of memory banks.
- `VALUES_PER_BANK` – number of words per bank.
- `FLOAT_MIN`, `FLOAT_MAX` – range of floating-point values.

Appendix C

Software Reference FFT Implementation

This appendix describes a Python script used to validate the output of the accelerator by performing the same butterfly-stage FFT computation in software. It loads raw input data from the same memory bank files used during simulation and performs the full arithmetic operation flow. This enables a direct comparison between the hardware accelerator results and the expected reference results computed in software.

C.1 Purpose

The goal of this script is to replicate the computation performed by the accelerator in software, using standard Python floating-point arithmetic. This allows checking that the hardware performs as expected on real input data, and serves as a debug and validation tool during simulation development.

C.2 Functionality

The script performs the following steps:

1. **Loads 6 input vectors** from memory bank files:
    - B, B': complex FFT inputs
    - T, T': twiddle factors
    - A, A': other FFT inputs used in butterfly combination

2. **Converts each 32-bit hex value** (raw IEEE 754) into floating-point values using Python's `struct` module.

3. **Computes the following intermediate values**:

$$P = B \cdot T - B' \cdot T'$$
$$P' = B' \cdot T + B \cdot T'$$

4. **Computes the final butterfly outputs**:

$$R = A + P$$
$$Q = A - P$$
$$R' = A' + P'$$
$$Q' = A' - P'$$

5. **Prints all intermediate and final results** in a readable, formatted list.

C.3 Usage

The script can be run as a standalone tool:

```
1 python3 fft_check.py
```

It assumes that the following bank files have been generated in the `sim/` directory:

- `Bank0.txt` – B
- `Bank1.txt` – T
- `Bank2.txt` – B'
- `Bank3.txt` – T'
- `Bank4.txt` – A
- `Bank5.txt` – A'

Each file should contain a list of 32-bit raw floating-point values in hex format (as generated by the memory initialization script). The script reads the first 4 values from each file and performs the butterfly computation.

C.4 Customization

The script is designed to be easily adapted for different testing scenarios. The following parameters can be modified to suit the needs of the simulation:

- `NUM_INPUTS`: Defines how many input values to read from each bank file. Increase this to validate larger FFT stages or multiple butterfly instances.
- `BANK_B`, `BANK_T`, etc.: Control which files are used as sources for the operands. These constants can be changed to point to different bank files if the input data layout changes.
- `SIM_FOLDER`: Points to the directory containing the memory files (default is `sim/`). This can be updated if the project uses a different folder structure.

Acronyms

ML Machine Learning

AI Artificial Intelligence

CPU Central processing unit

ASIC Application Specific Integrated Circuit

CGRA Coarse Grained Reconfigurable Architecture

PE Processing Element

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

DMA Direct Memory Access

MAC Multiply-Accumulate

AXI Advanced eXtensible Interface

RISC-Vp++ RISC-V Virtual Prototype Plus Plus

FFT Fast Fourier Transform

DIT Decimation In Time

Bibliography

[1] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. «A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications». In: *ACM Computing Surveys* 52.6 (2019), 118:1–118:39. DOI: 10.1145/3357375 (cit. on pp. 1, 4–6, 9).

[2] Allan Carroll, Stephen Friedman, Brian Van Essen, Aaron Wood, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. «Designing a Coarse-Grained Reconfigurable Architecture for Power Efficiency». In: *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Based on DOE Grant No. DE-FG52-06NA27507. IEEE. 2009 (cit. on pp. 1, 5, 6, 10).

[3] Luca Zulberti, Matteo Monopoli, Pietro Nannipieri, Luca Fanucci, and Silvia Moranti. «Highly Parameterised CGRA Architecture for Design Space Exploration of Machine Learning Applications Onboard Satellites». In: *2023 International Conference on Energy-Efficient High-Performance Computing (EDHPC)*. IEEE, 2023. DOI: 10.23919/EDHPC59100.2023.10396632 (cit. on pp. 1, 5).

[4] Egbart de Bruin. «Design of Energy-Efficient CGRA-based Systems». PhD Thesis. PhD thesis. Eindhoven University of Technology, 2024. URL: https://research.tue.nl/en/publications/1d432d9b-8e40-4878-ad00-e08d165c344d (cit. on pp. 2, 4).

[5] Daniel Mühlbacher, Fabian Prantl, Wolfgang Puffitsch, Andreas Wallner, Andreas Krall, and Thomas Krennwallner. «RISC-V VP++: A Flexible and Extensible RISC-V Based Virtual Platform». In: *Proceedings of the Workshop on Open-Source Design Automation (OSDA)*. https://ics.jku.at/files/2024OSDA_RISCV-VP-plusplus.pdf. ACM/IEEE. 2024 (cit. on pp. 2, 7, 11, 13, 30).

[6] Doulos. *Introduction to TLM 2.0*. https://www.doulos.com/knowhow/systemc/tlm-20/. Accessed: 2025-09-30. n.d. (Cit. on p. 7).

[7] Yasuto Aihara, Boma Anantasatya Adhi, Kota Aiyoshi, Chenlin Shi, Jason Anderson, Tomohiro Ueno, Kentaro Sano, and Takaaki Miyajima. «Impact of Reconvergent DFG Paths and Buffer Depth on Elastic CGRA Throughput». In: *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. Kumamoto, Japan: ACM, 2025. DOI: 10.1145/3728179.3728193 (cit. on pp. 10, 28).

[8] ARM Ltd. *Introduction to AMBA-PV: Extensions to TLM 2.0*. https://developer.arm.com/documentation/100962/0200/Introduction-to-AMBA-PV-Extensions-to-TLM-2-0. Accessed: 2025-09-30. n.d. (Cit. on pp. 11, 13, 30).

[9] ARM Ltd. *CMSIS: Cortex Microcontroller Software Interface Standard*. Available at https://developer.arm.com/tools-and-software/embedded/cmsis. 2024. URL: https://developer.arm.com/tools-and-software/embedded/cmsis (cit. on p. 11).