TÉCNICO LISBOA

Stream-based Coarse-Grained Reconfigurable Accelerator

João Francisco Batista Maia

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Nuno Filipe Valentim Roma
Prof. Pedro Filipe Zeferino Aidos Tomás

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. Nuno Filipe Valentim Roma
Member of the Committee: Prof. João Paulo de Castro Canas Ferreira

June 2025

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I would like to express my profound gratitude to my supervisors, without whom this work would not have been possible. My sincerest thanks to Professor Nuno Roma for his invaluable guidance, feedback, and constant support, always bringing a light-hearted approach to even the most challenging moments. His enthusiasm and positivity were a steady source of encouragement.

To Professor Pedro Tomás, thank you for your insightful suggestions, critical questions, and keen eye for detail, which deepened my understanding and sharpened my work at every stage. Your commitment to excellence was both inspiring and instrumental in advancing this project.

A special thank you to Professor Nuno Neves, whose hands-on assistance and willingness to dive into the development alongside me made a tangible difference. Your practical support and expertise were instrumental throughout this journey.

I owe an immeasurable debt of gratitude to my family, who have been my unwavering foundation. Your endless support and belief in me have been a safe place to land, always encouraging me to reach higher.

To my friends, whom I've had the privilege of meeting during my university years, each of you has added something unique to my journey. Life is truly more colorful with you all in it. Your presence, laughter, and companionship have been a source of strength throughout this journey.

To each of you, I am profoundly thankful.

Work supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under project 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC). We also acknowledge the contributions from project UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020).

Resumo

Esta tese apresenta uma arquitetura de acelerador de *dataflow* reconfigurável, orientada por *streaming*, e baseada em Arquiteturas Reconfiguráveis de Grão Grosso (CGRAs) para computação de alto desempenho e eficiência energética. A arquitetura desagrega o acesso à memória da computação por meio de um motor de *streaming* dedicado, baseado no paradigma de *streaming* proposto para a Unlimited Vector Extension, para lidar eficientemente com padrões de memória afins e suportar ciclos multidimensionais complexos. A integração com um sistema-em-chip baseado na arquitetura RISC-V, utilizando o ambiente Chipyard, permite uma avaliação particularmente abrangente.

Foi desenvolvida uma cadeia completa de ferramentas de compilação para automatizar a extração de gráficos de fluxo de dados, identificação de padrões de *streaming* e mapeamento para CGRA. A arquitetura foi sintetizada numa tecnologia de 7 nm para avaliar o seu desempenho e eficiência. Embora a avaliação direta tenha revelado melhorias modestas em desempenho e eficiência, a análise realizada identificou limitações importantes nas implementações atuais do motor de *streaming* e nas ferramentas de mapeamento. A estimativa dos ganhos previstos com as melhorias propostas para esses componentes demonstrou ser muito significativa, tanto em desempenho como em eficiência energética, evidenciando o potencial promissor da arquitetura.

Os resultados alcançados destacam o valor das arquiteturas reconfiguráveis baseadas em *dataflow* orientadas por *streaming* como uma plataforma muito viável para aplicações intensivas em computação, ao mesmo tempo em que sublinham a importância de soluções robustas de *streaming* e mapeamento. As melhorias propostas criam uma base para futuras investigações que explorem todo o potencial do sistema, visando alcançar um desempenho competitivo e com grande eficiência energética.

Palavras-chave: Data streaming, Dataflow, Motor de streaming, CGRA, Reconfigurável, Acelerador

Abstract

This thesis introduces a stream-based dataflow reconfigurable accelerator architecture based on Coarse-Grained Reconfigurable Architectures (CGRAs) for high-performance computing and energy efficiency. The architecture decouples memory accesses from computation through a dedicated streaming engine based on the Unlimited Vector Extension streaming paradigm to efficiently handle affine memory patterns and support complex multi-dimensional loops. Integration with a RISC-V-based system-on-chip using the Chipyard framework enables a comprehensive evaluation.

An end-to-end compilation toolchain was developed to automate the dataflow graph extraction, streaming pattern identification, and CGRA mapping. The architecture was synthesized in a 7 nm technology node to assess its performance and efficiency. While direct evaluation revealed only modest performance and efficiency improvements, the conducted analysis identified key limitations in the existing streaming engine implementations and mapping tools. Proposed enhancements to these components were estimated to yield substantial performance and energy efficiency gains, demonstrating the architecture's promising potential.

These findings underscore the value of stream-driven reconfigurable dataflow architectures as a highly viable platform for computation-intensive applications while highlighting the importance of robust streaming and mapping solutions. The proposed improvements lay a foundation for future research to fully realize the system's capabilities in delivering competitive energy-efficient performance.

Keywords: Data streaming, Dataflow, Streaming engine, CGRA, Reconfigurable, Accelerator

List of contents

List of tables

List of figures

Acronyms

ADL	Architecture description language
AGI	Address generation instruction
ALU	Arithmetic logic unit
API	Application programming interface
ASIC	Application-specific integrated circuit
AST	Abstract syntax tree
AXI4	Advanced eXtensible Interface 4
BOOM	Berkeley Out-of-Order Machine
BPN	Beneš permutation network
CAD	Computer-aided design
CGRA	Coarse-grained reconfigurable architecture
COTS	Commercial-off-the-shelf
CPU	Central processing unit
DDR	Double data rate
DFG	Data-flow graph
DMA	Direct memory access
DNN	Deep neural network
DRAM	Dynamic random-access memory
DSA	Domain-specific architecture
EC2	Elastic Compute Cloud
EDA	Electronic design automation
FFT	Fast Fourier transform
FIR	Finite impulse response
FPGA	Field-programmable gate array
FU	Functional unit
GEMM	General matrix multiply
GPP	General-purpose processor
GPU	Graphics processing unit
HDL	Hardware description language
HLS	High-level synthesis

HPCAS High-Performance Computing Architectures and Systems
I/O Input/output
II Initiation interval
ILP Integer linear programming
ILP Instruction-level parallelism
IP Semiconductor intellectual property
IR Intermediate representation
ISA Instruction set architecture
ISCAS IEEE International Symposium on Circuits and Systems
JSON JavaScript Object Notation
LF Load FIFO
LLB Load line buffer
LMMU Load memory management unit
LRQ Load request queue
LSU Load/store unit
MMIO Memory-mapped I/O
MMU Memory management unit
MRRG Modulo routing resource graph
N2N Neighbor-to-neighbor
NoC Network-on-chip
OS Operating system
PCU Pattern Compute Unit
PDK Process design kit
PE Processing element
PMU Pattern Memory Unit
QoM Quality of mapping
RF Register file
RoCC Rocket custom co-processor
RTL Register-transfer level
RTU Reconfigurable Tensor Unit
RVV RISC-V Vector Extension
SA Simulated annealing
SC Static configuration
SE Streaming engine
SIMD Single instruction multiple data
SMMU Store memory management unit
SoC System-on-chip
SPM Scratchpad memory
SR Stream register

SRAM Static random-access memory

SSA Static single assignment

TL TileLink

TL-C TileLink Cached

TL-UL TileLink Uncached Lightweight

TPU Tensor Processing Unit

UVE Unlimited Vector Extension

VLSI Very large-scale integration

VMA Vector multiply-accumulate

VTR Verilog to Routing

XML Extensible Markup Language

1

Introduction

Contents

1.1 Motivation

The most recent advances in machine learning and big data research have promoted significant progress in various domains, including computer vision, speech recognition, natural language processing, drug discovery, and genomics. These improvements have been made possible through increasingly complex and computationally demanding algorithms, traditionally supported by advances in integrated circuit technology. However, as Moore's law slows down [1–3] and Dennard scaling reaches its end [4], the *power wall* imposes a strict limitation on the operation of circuits: the power budget of integrated circuits has peaked, and further improvements must now rely on gains in energy efficiency. As such, general-purpose techniques, often supported by central processing units (CPUs) or graphics processing units (GPUs), are no longer sufficient due to the low energy efficiency typically offered by traditional von Neumann architectures.

To address these shortcomings, the research community has been focusing its attention on application-specific and domain-specific accelerators [5–19]. These dedicated architectures achieve greater energy efficiency by specializing their data and control paths to the specific needs of their application domains. However, while these narrow hardware solutions are often effective, they struggle to keep pace with the rapid and constant evolution of software algorithms, resulting in short lifecycles and high non-recurring engineering costs [20].

As a result, there has been a push for research to focus on reconfigurable architectures [21, 22]. In particular, field-programmable gate arrays (FPGAs) offer a high degree of flexibility, which, coupled with recent advances in high-level synthesis (HLS), has made them a popular option as accelerators. However, FPGAs exhibit poor power and area efficiency due to the overhead associated with bit-level reconfigurability.

Conversely, thanks to their flexibility and greater energy efficiency, coarse-grained reconfigurable architectures (CGRAs) have also received considerable attention, particularly in computation-intensive applications such as signal processing, computer vision, artificial intelligence, and cryptography [23–34]. CGRAs consist of word-level reconfigurable processing elements (PEs) and interconnects, which can be configured to execute spatial computations represented as data-flow graphs (DFGs). Additionally, CGRAs can be reconfigured at runtime, as they require much less configuration data than FPGAs.

In many previous CGRA works [24–26, 29, 33, 34], the DFG contains nodes representing the calculation of iteration variables and load/store address generation operations. However, in most kernels, the patterns of these variables are regular and can be extracted at compile time, providing an opportunity to offload their execution to dedicated hardware. This approach can reduce the number of nodes in the DFG, increasing throughput and efficiency by saving resources (PEs and interconnects) and allowing a more efficient mapping of the kernel to the CGRA. Another advantage of removing these nodes from the DFG is that multi-dimensional nested loops can be represented as a single DFG, as the extracted patterns already encode the loop structure. Additionally, because the patterns are explicitly known, this approach also allows for prefetching memory accesses, reducing memory hierarchy latency.

Since conventional CGRAs inherently have little support for complex control flows, they are usually

integrated into a system-on-chip (SoC), coupled with a memory system and a host processor to handle the control flow and data movement. Nevertheless, many prior studies [23–26, 28, 33] have focused on the CGRA itself, lacking or having limited integration within a complete system, which makes it difficult to evaluate system-level effects, such as the impact of the memory hierarchy or the communication between the CGRA and the host processor.

1.2 Objectives

The main objective of this work is to design and implement an accelerator architecture based on a CGRA, in which memory accesses are completely decoupled from computation. This new design will take advantage of significant contributions recently presented by the High-Performance Computing Architectures and Systems (HPCAS) research team, including the development of the Unlimited Vector Extension (UVE) data streaming solution [35] and the corresponding hardware implementation of a streaming engine [36].

To evaluate the proposed contribution, the accelerator will be integrated into a complete system based on the Chipyard SoC framework [37], along with a RISC-V host processor.

Another adressed objective is the development of an end-to-end compilation toolchain that generates the CGRA configuration bitstream from a high-level language, such as C or C++, along with the necessary software for the host processor. The toolchain should be capable of extracting memory access patterns to be offloaded to the streaming infrastructure, as well as generating a feasible mapping of the computation onto the CGRA infrastructure, encoding this information into a configuration bitstream for the accelerator.

The implemented architecture will be evaluated using a diverse set of benchmarks from a diverse set of application domains that include linear algebra, image processing, computational physics, and quantitative finance. These evaluations will be performed by running FPGA-accelerated register-transfer level (RTL) simulations with realistic memory models on the FPGA infrastructure available at Amazon Elastic Compute Cloud (EC2), using the convenient mechanisms provided by the FireSim platform [38].

The obtained results will be compared with state-of-the-art domain-specific architectures (DSAs), as well as single instruction multiple data (SIMD) solutions in off-the-shelf general-purpose processors (GPPs), in terms of performance, energy efficiency, and area.

1.3 Key contributions

Building on the main objectives outlined above, the key contributions of this thesis are summarized as follows:

- A novel accelerator architecture that decouples memory accesses from computation by integrating a data streaming solution with a CGRA.

- A novel parameterized implementation of a stream-supported CGRA, capable of executing multi-dimensional nested loops with complex memory access patterns, while maintaining a similar area footprint and power consumption to a comparable DSA.
- A complete prototyping architecture, based on the Chipyard SoC framework, that integrates the CGRA with a RISC-V host processor capable of reconfiguring the accelerator at runtime.
- An end-to-end compilation toolchain that automatically generates the accelerator configuration bitstreams from kernels written in C/C++ source code. This toolchain encompasses loop transformation tools, the extraction of computation DFGs and memory patterns, the mapping to the accelerator, and configuration bitstream generation.
- An evaluation of the implemented architecture using a diverse set of benchmarks from various domains, executed on the FPGA infrastructure available on Amazon EC2. Results demonstrate speedups of up to 17.87× and energy efficiency improvements of up to 2.95×.
- The identification of overheads and bottlenecks in the current implementation, proposals for improvement, and an estimate of the performance of a hypothetical improved system. The projected results indicate speedups of up to 668× and energy efficiency gains of up to 110×.

Part of the work presented in this thesis was published in a peer-reviewed paper entitled “Stream-Driven Acceleration for Embedded RISC-V SoCs”, presented at the 2025 IEEE International Symposium on Circuits and Systems (ISCAS) [39].

1.4 Outline

This thesis is organized as follows. Chapter 2 provides background on the state of the art in accelerator architectures and systems, focusing on CGRAs and data streaming applied to accelerators. Chapter 3 introduces the computational and abstraction models that underpin the proposed system and presents an overview of the system, including the architecture of the accelerator and its compilation framework. Chapters 4 and 5 detail the implementation of the accelerator’s microarchitecture and the compilation toolchain, respectively. Chapter 6 presents the experimental evaluation, including the testing methodology, the benchmarks, the obtained results, and the corresponding discussion. Finally, Chapter 7 concludes the thesis and suggests directions for future work.

2

Background and related work

Contents

This chapter provides a theoretical framework for understanding the topics investigated in this dissertation. The landscape of computing accelerators is diverse, with DSAs having received considerable attention from the computer architecture community in recent years. These architectures consist of fixed dataflow structures tailored to specific application domains such as machine learning, cryptography, or data compression, delivering substantial gains by optimizing their datapaths to the unique computational requirements of a particular domain's workloads. In addition, some DSAs exhibit some adaptability through reconfigurable dataflows, most notably CGRAs.

This chapter explores fixed dataflow architectures and CGRAs, discussing their benefits and limitations, while addressing the compilation challenges inherent to CGRAs. This chapter also addresses data streaming, especially in the context of its use together with accelerators, with a special interest in UVE. Finally, the works related to the tools and frameworks used to implement the proposed architecture are presented.

2.1 Fixed dataflow architectures

We consider fixed dataflow architectures those that implement fixed functions and present datapaths with very little to no reconfigurability capabilities. These architectures commonly take the form of specialized application-specific integrated circuits (ASICs), designed to accelerate a specific workload. This level of optimization results in accelerators with significant performance gains for the targeted workloads. They occupy the rightmost space in the flexibility-efficiency space presented in Fig. 2.1. These kinds of architectures have been successfully used in the industry [6, 10, 19, 40], with notable examples including Google's TPU [6] and NVIDIA's Tensor Cores [40], and in academia with DianNao [8] or Gemmini [9].

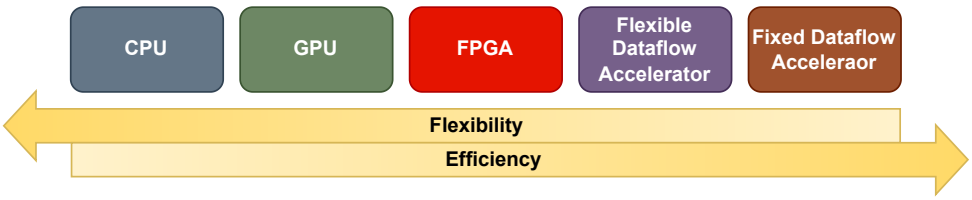


Figure 2.1: Flexibility-efficiency space.

Tensor Processing Unit

The demanding processing needs of deep neural network (DNN) applications propelled the creation of novel and sophisticated tensor-based architectures. One such example of a successful accelerator is Google's Tensor Processing Unit (TPU), which has been specifically designed to accelerate DNNs and has been deployed in data center environments [6]. Google's TPU success also led to it being released to the general public through Google's cloud platform, allowing users to access and utilize the advanced hardware acceleration capabilities of TPUs for their own machine learning workloads [41].

TPUs are implemented around the use of a systolic array, as shown in Fig. 2.2. A systolic array is a

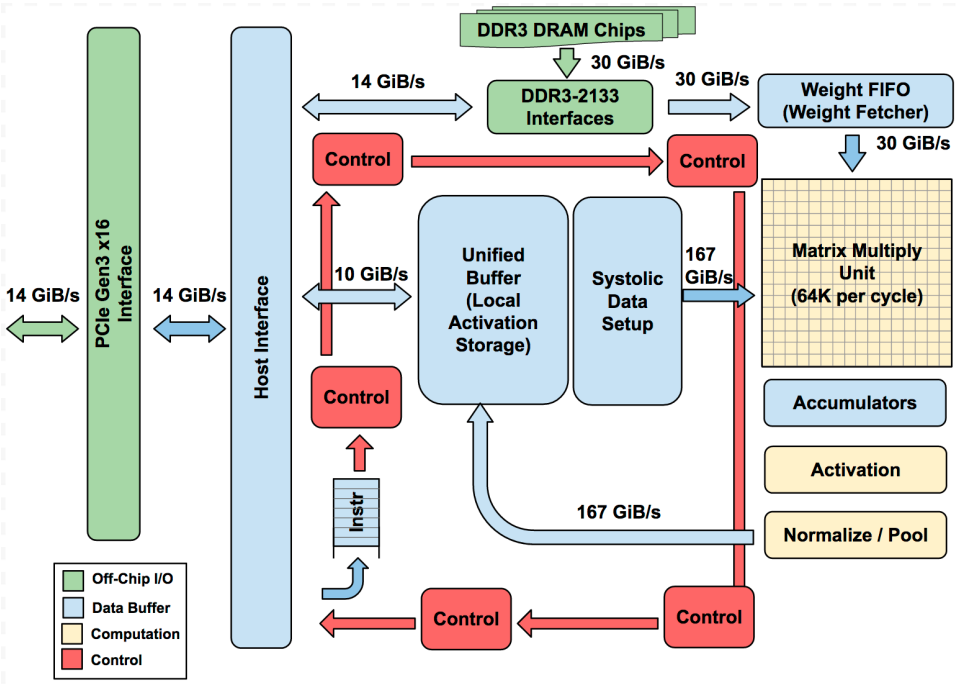


Figure 2.2: TPU block diagram [6].

type of parallel architecture that consists of a grid of PEs, each of which performs a specific computation on a set of inputs, and passes the outputs to adjacent PEs [42, 43]. This paradigm contrasts with the traditional von Neumann architectures, in which a program tells the arithmetic logic units (ALUs) which registers to read, the operation to perform, and the register in which to store the results. Systolic array architectures thus exploit a greater spatial and temporal locality of data, which can significantly improve the performance of vector and matrix operations compared to von Neumann architectures.

Nvidia's Tensor Cores

Nvidia's Tensor Cores are another example of DSAs designed for tensor multiplication. Although they also perform a fixed function, they offer an increased level of flexibility through their integration into general-purpose GPUs [40]. Tensor Cores were first introduced in the NVIDIA Tesla V100 GPU and have been improved and enhanced in each subsequent generation of NVIDIA GPU architectures. Current-generation NVIDIA H100 Tensor Cores support mixed-precision arithmetic, with a wide range of integer and floating point numerical formats, and also support sparse tensor arithmetic, doubling the performance of standard Tensor Core operations on sparse deep learning networks [44].

Gemmini

Gemmini is an open-source full-stack accelerator generator that allows architects and researchers to customize and generate hardware accelerators for DNN applications. It provides a wide range of options for hardware architectures, programming interfaces, and system integration, so that users can generate accelerators that are optimized for a variety of different scenarios, from low-power edge devices to high-performance cloud servers [9, 45].

One of its advantages is that Gemmini is provided within the Chipyard framework [46], meaning that it is distributed within a complete SoC, with a robust software toolchain supporting it. This way, it provides researchers with the ability to study how different components of the system and software stack interact and affect the resulting DNN performance, including the hardware accelerator itself, the SoC, the operating system (OS), and the software overhead.

As shown in Fig. 2.3, Gemmini's architectural template is based on a spatial array of PEs that perform dot products and accumulations to implement matrix multiplication. Moreover, Gemmini is implemented as a co-accelerator for a RISC-V core, interacting with the system via non-standard RISC-V custom instructions.

By default, it supports both output-stationary and weight-stationary dataflows, as illustrated in Fig. 2.4. For a $C = A * B + D$ matrix multiplication implemented with an output-stationary dataflow, the biases (D) are preloaded, and the matrices that enter the array in the execution stage are the operands (A) and the weights (B). In weight-stationary dataflow, the weights matrix (B) is preloaded, and the operands (A) and bias (D) matrices are what enter the systolic array. The dataflow can be chosen at runtime or hardened at elaboration time. Fig. 2.4 also shows that Gemmini's systolic array is organized into a two-level hierarchy, with each tile being fully combinatorial and a mesh of tiles having pipeline registers between each tile.

The systolic array's inputs and outputs are stored in an explicitly managed scratchpad of banked static random-access memories (SRAMs). A direct memory access (DMA) engine facilitates data transfers between the main memory (visible to the host CPU) and the scratchpad. In weight-stationary dataflow, an additional SRAM bank, equipped with adder units, is used as an accumulator, storing partial results from the systolic array and providing new inputs to the array. The DMA engine can also transfer data directly between the accumulator and the main memory.

In addition to matrix multiplication, Gemmini includes optional peripheral circuitry that can apply activation functions, scale the output results, and transpose matrices.

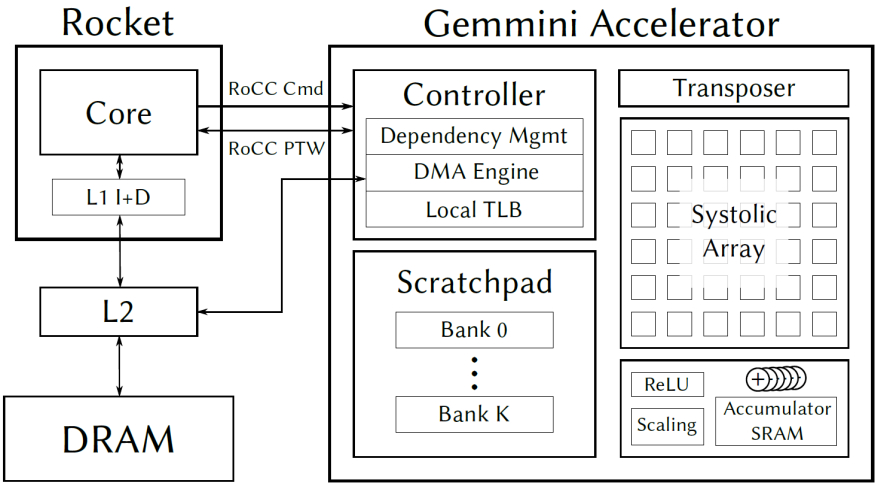


Figure 2.3: Gemmini architectural template overview [9].

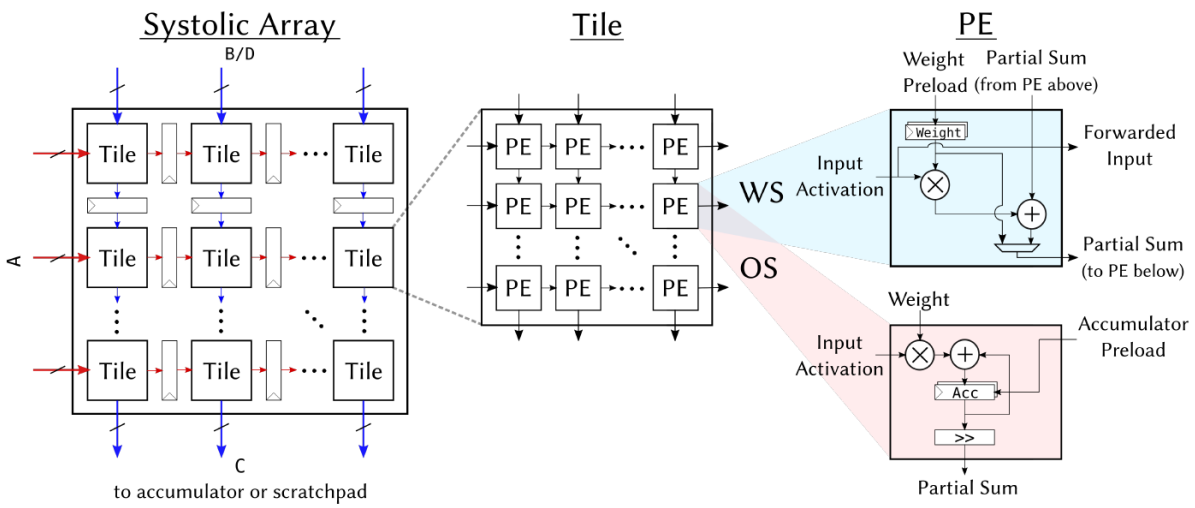


Figure 2.4: Microarchitecture of Gemmini's spatial array [9].

2.2 Coarse-grained reconfigurable architectures

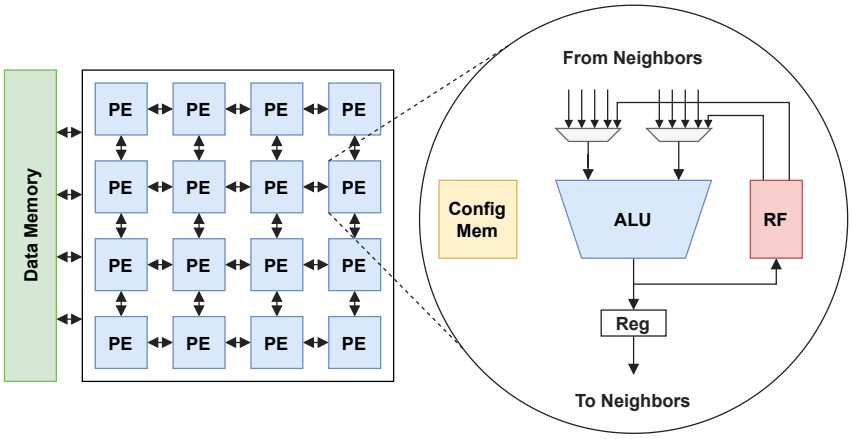


Figure 2.5: Common CGRAs architecture.

Several recent research efforts have renewed and highlighted the potential of CGRAs as a type of accelerator known for their combination of high computational throughput, energy efficiency, and flexibility [47].

2.2.1 Architecture

CGRAs usually consist of an array of PEs connected through an interconnect network, usually a 2D mesh network, as depicted in Fig. 2.5. Each PE is composed of an ALU, a register file (RF), and a configuration memory. At each clock cycle, the contents of the configuration memory determine the operation performed by the ALU, as well as the source of the operands and the destination of the result. These sources and destinations can be selected from either the register file or a neighboring PE.

Unlike traditional fixed dataflow accelerators, the flexibility of CGRAs allows the dataflow to be adapted on a kernel-by-kernel basis. This flexibility comes at the cost of a loss of specialization, re-

sulting in a less efficient accelerator than a dedicated ASIC, while using more area. Hence, CGRAs balance the efficiency achieved by the rigidness of fixed dataflows in specialized ASICs and the flexibility of FPGAs' fine-grained structures. Due to the coarse-grained nature of CGRAs' configuration options, the reconfiguration time is much short (ns-µs range) compared to the reconfiguration time of FPGAs (in the order of ms-s) since there is much less configuration data to be written [47].

2.2.2 Related works

HyCUBE

Most CGRAs are characterized by static interconnects between neighbors in the form of either neighbor-to-neighbor (N2N) connections or standard network-on-chip (NoC). In N2N, the functional units (FUs) only connect to neighbors, meaning that transfers between distant FUs have to be routed through intermediate FUs, which makes them unavailable for computing. Standard NoC-based solutions allow for dedicated node-to-node links, independent of the FUs, allowing for a PE to simultaneously perform routing and computation, but still introducing latency due to the registers between PEs.

However, both of these approaches make routing a challenging problem for the compiler. By having this into account, Karunaratne et al. have proposed HyCUBE [26], which uses a reconfigurable interconnect supporting single-cycle communications between distant PEs. Their implementation is based on a NoC with bypassable registers between PEs, allowing for multi-hop connectivity within a single cycle, as depicted in Fig. 2.6.

By limiting the maximum multi-hop distance to 4 hops (enough for most kernels), the authors obtain a critical path delay comparable with standard NoC implementations and the complexity of routing is greatly reduced, as most routes become a single hop. This leads to a better mapping quality, which, coupled with the reduction of the communication latency between FUs, results in a performance-per-watt that is 1.5× and 3× better than conventional CGRAs with a standard NoC or N2N interconnect, respectively [26].

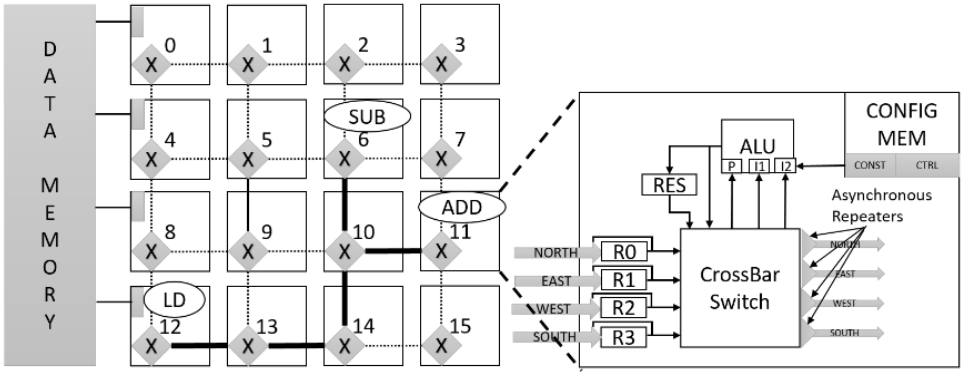


Figure 2.6: HyCUBE block diagram with multi-hop paths between PEs [26].

Reconfigurable Tensor Unit

The Reconfigurable Tensor Unit (RTU) proposed by Neves et al. [28] is a new type of tensor computing unit designed to be more flexible and general-purpose than current tensor units. Their RTU architecture, shown in Fig. 2.7, includes a dense 2D array of reconfigurable PEs, each implementing a vector multiply-accumulate (VMA) unit with variable-precision SIMD capabilities, supported by the use of the posit number system. This allows for a higher degree of vectorization and flexibility, when compared to traditional fixed-precision SIMD architectures. Posit numbers [48] are an alternative numbering format to the IEEE-754 standard, commonly used in floating-point arithmetic. The posit format has some hardware-friendly properties, as it does not require intermediate normalization and rounding in fused operations and allows for similar accuracy with lower precision than the IEEE-754 standard [49]. The RTU's execution model is based on data streaming, in which autonomous stream generators produce common data patterns and feed them into the processing elements connected to a banked scratchpad/buffering memory structure. The RTU is programmed by providing configurations for each individual PE and defining a memory access pattern descriptor for each stream generator. The PE configurations are managed by the controllers within each PE, while the stream generators use the memory access patterns to fetch data.

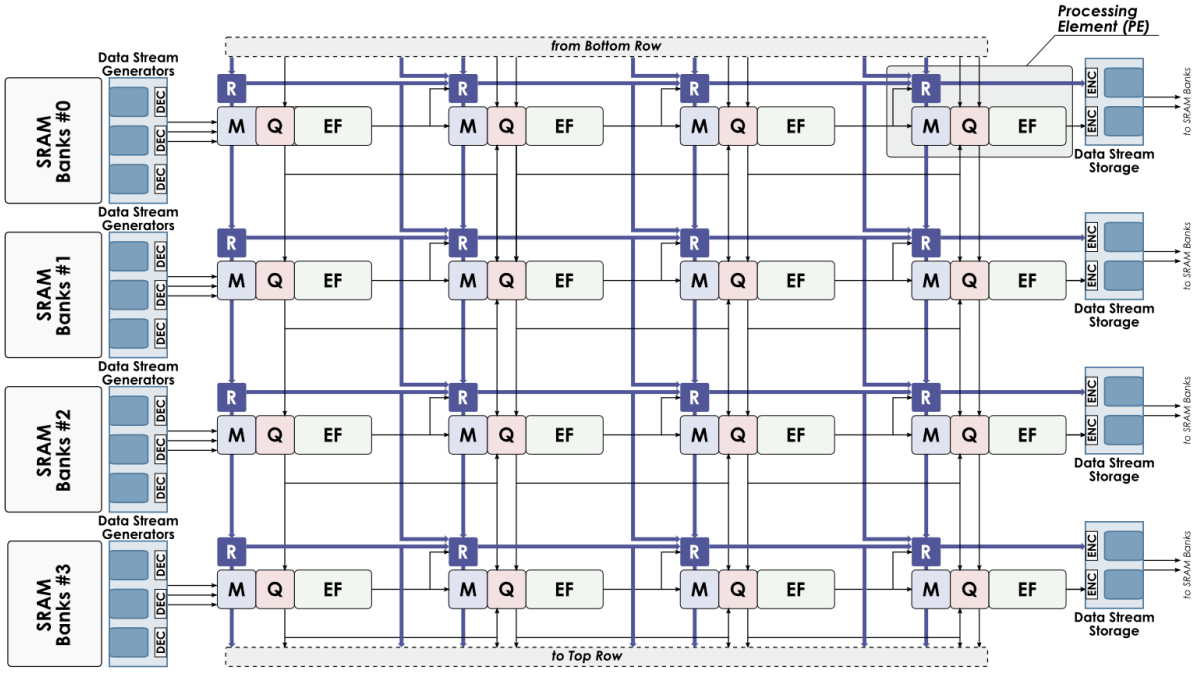


Figure 2.7: RTU block diagram [28].

Cascade

In most kernels, address generation instructions (AGIs) represent a significant part of computations. Conventional CGRAs perform address generation computations within the processing array, which limits the achievable throughput due to the resource contention between address generation and data

processing. To address this issue, Wijerathne et al. have proposed a decoupled access-execute CGRA denoted by CASCADE [27].

Similarly to the previously described RTU [28], CASCADE offloads the address generation to a dedicated streaming engine (SE), implemented as a dedicated programmable hardware unit to handle conflict-free access to a multi-bank memory. Its architecture, depicted in Fig. 2.8, comprises two main components: the execute part and the access part. While the execute part is a conventional 2D-mesh CGRA, the access part features a SE, a multi-bank scratchpad memory (SPM), and a set of stream registers (SRs), connected via two crossbars, one for selecting non-conflicting banks and the other for selecting the stream registers.

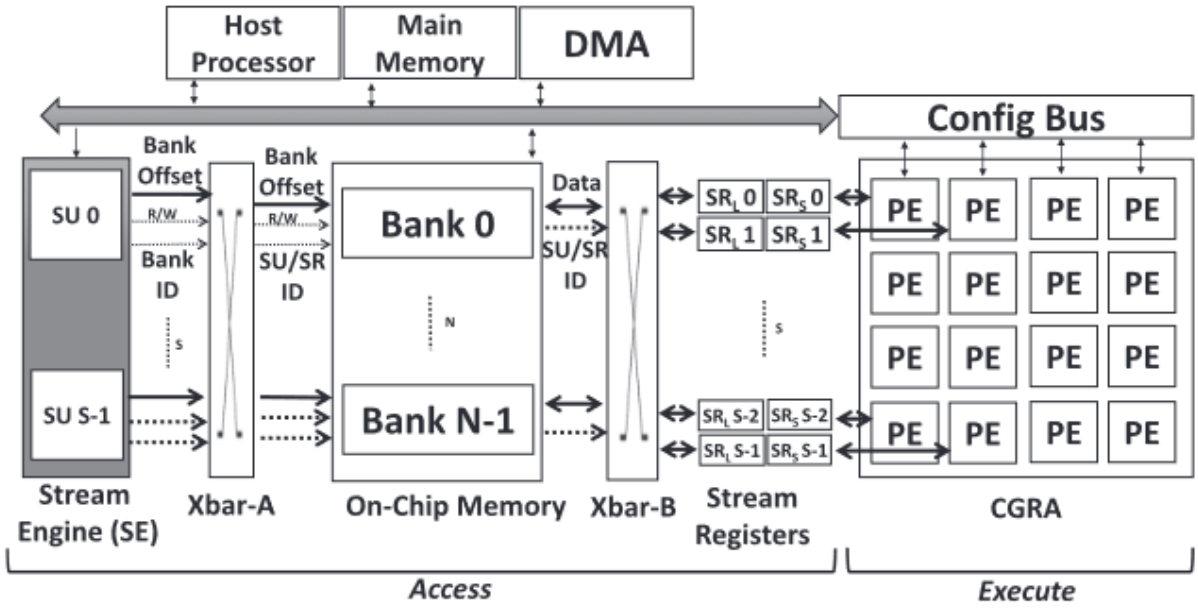


Figure 2.8: CASCADE architecture [27].

This structure is supported by a fully automated compiler that, besides performing the CGRA mapping, also extracts memory access patterns and manages the required bank allocation in order to generate conflict-free data streaming configurations. Fig. 2.9 depicts the end-to-end compiler framework.

Experimental evaluations demonstrate that CASCADE achieves an average performance improvement of 3× and a performance-per-watt improvement of 2.2×, when compared to conventional CGRAs with similar area, which typically compensate for the lack of dedicated address generation hardware by employing larger processing arrays.

Plasticine

Plasticine is a CGRA accelerator designed for the efficient execution of parallel patterns [24]. As shown in Fig. 2.10, it consists of a two-dimensional array composed of two types of coarse-grained reconfigurable units: Pattern Compute Units (PCUs) and Pattern Memory Units (PMUs). Each PCU consists of a reconfigurable pipeline with multiple stages of SIMD functional units, with support for cross-SIMD lane shifting and reduction, as shown in Fig. 2.11. PMUs are composed of a banked scratchpad

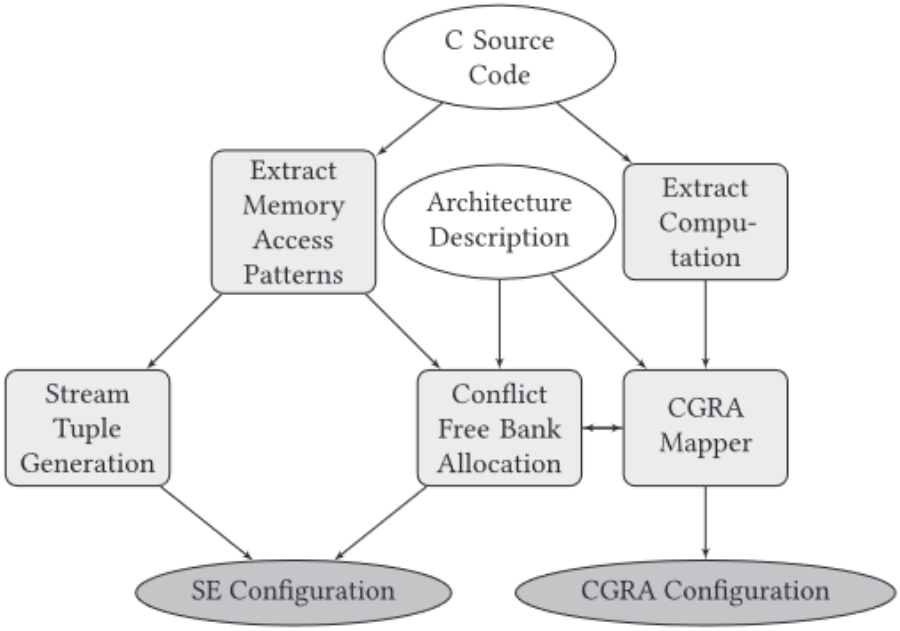


Figure 2.9: CASCADE compiler framework [27].

memory and dedicated addressing logic and address decoders. These units communicate with each other through a pipelined interconnect that provides the flexibility that is enough to support a wide range of parallel patterns, including Map, Reduce, Fold, and FlatMap, as depicted in Fig. 2.12. Moreover, plasticine uses a distributed and hierarchical control scheme to minimize the synchronization between units and adapt the connectivity in the interconnect. There are three types of controller protocols: sequential execution, coarse-grained pipelining, and streaming. These control schemes correspond to outer loops in the input program and determine how the execution of individual units is scheduled relative to other units. Plasticine also includes support for double buffering, which can be generalized as N-buffering, enabling coarse-grain pipelined execution of imperfectly nested patterns. The direct architectural support for parallel patterns in Plasticine simplifies program development by allowing the use of high-level parallel patterns rather than low-level programming models.

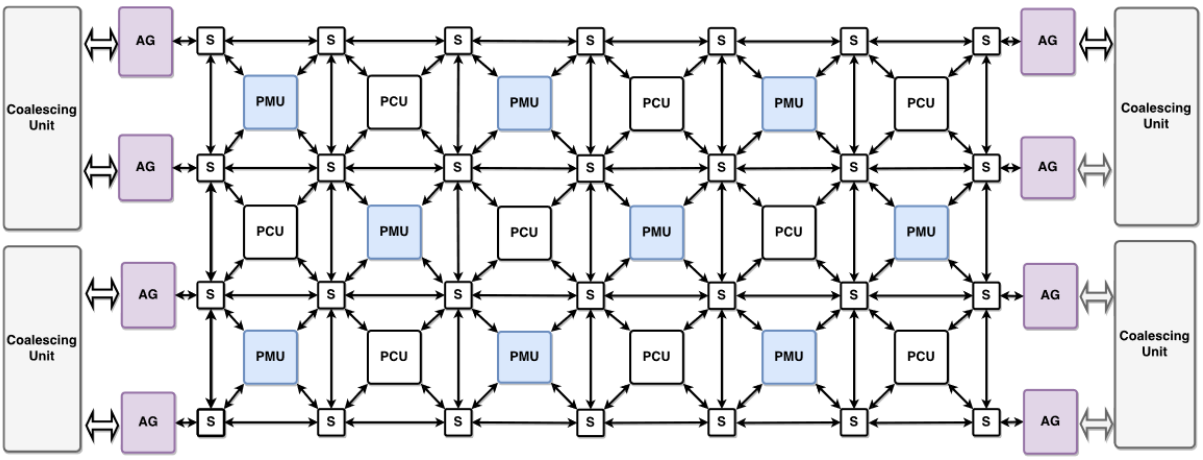


Figure 2.10: Plasticine top-level architecture [24].

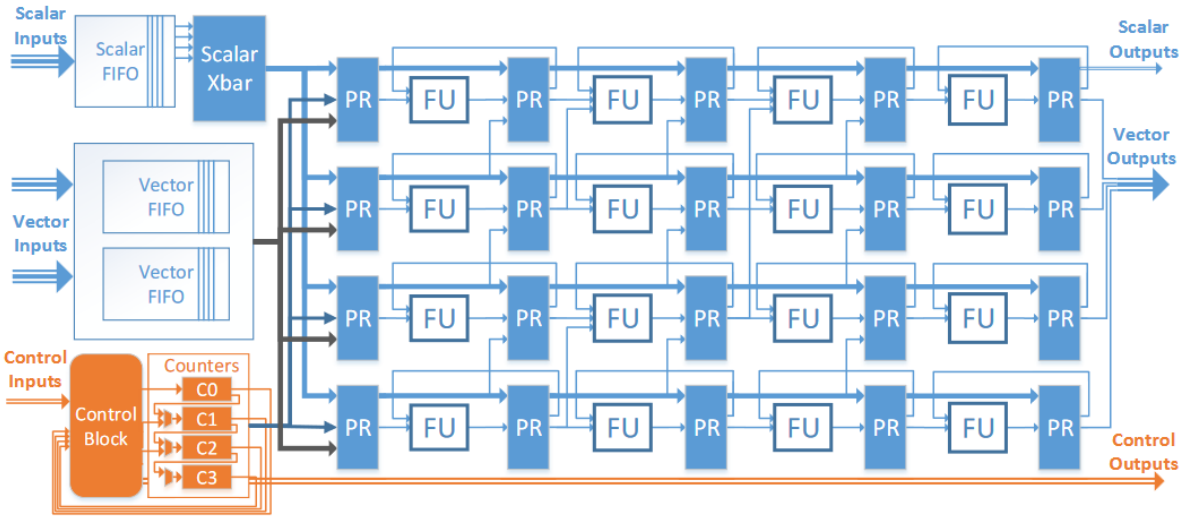


Figure 2.11: Plasticine's PCU architecture [24].

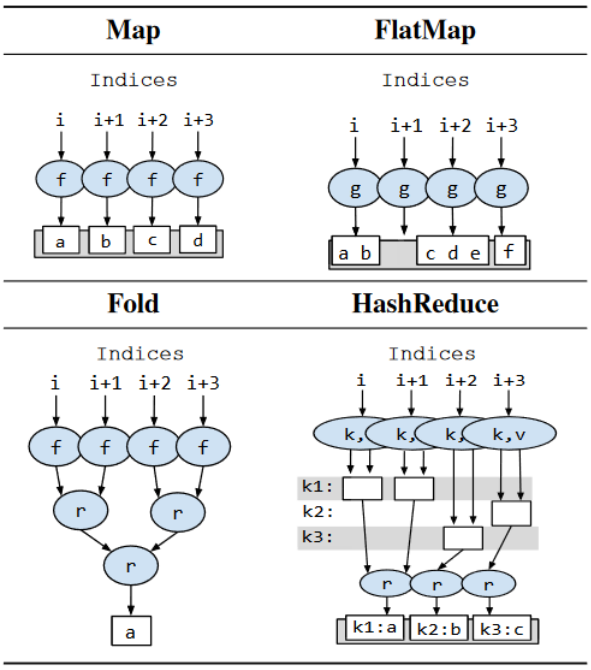


Figure 2.12: Parallel patterns supported by Plasticine's programming model [24].

Softbrain

Another recent approach to domain-specific acceleration is the use of the stream-dataflow architecture proposed by Nowatzki et al. [23], whose Softbrain accelerator is illustrated in Fig. 2.13. The Softbrain microarchitecture consists of a low-power control core that generates stream commands, a set of stream engines that efficiently interface with memories and move data, and a deeply-pipelined reconfigurable dataflow substrate (CGRA) for efficient parallel computation. The Control Core is a low-power, single-issue in-order core that is responsible for generating stream-dataflow commands for the stream dispatcher to execute. It acts as a programmer for the stream dispatcher and is designed to be low power and low area due to its limited function as a specialized rather than a general-purpose

processor. The Stream Dispatcher Unit coordinates the execution of the stream engines by tracking resource dependencies and issuing commands when needed. It also manages the concurrent execution of the stream engines by sending them commands and maintaining the program order of streams with the same source or destination port. There are three main types of stream engines in the Softbrain microarchitecture: the Memory Stream Engine, the Scratchpad Stream Engine, and the Recurrence Stream Engine. The Memory Stream Engine processes streams that access the main memory, issuing read and write requests and exchanging data with the Vector Port Interfaces. The Scratchpad Stream Engine processes streams that access the scratchpad memory instead of the main memory, allowing for the efficient reuse of data that has been loaded. The Recurrence Stream Engine is used for the immediate reuse of data without the need for memory storage, forwarding data directly from the Output Vector Port to the Input Vector Port after processing. The Vector Port Interfaces act as the interface between the computations performed by the CGRA and the incoming and outgoing data streams. The Input Vector Port Interface stores incoming data from streams until the CGRA is ready to consume it, while the Output Vector Port Interface stores the result of the computation performed by the CGRA until it is ready to be consumed by an outgoing stream. The Indirect Vector Port Interface stores the streaming addresses of indirect loads or stores. Finally, the CGRA supports pipelined computation of DFGs and avoids the overheads of accessing register files or memories for live values. It has no flow control inside its mesh, which reduces area, but requires the DFG compiler to ensure that all computation paths are appropriately matched in terms of delay.

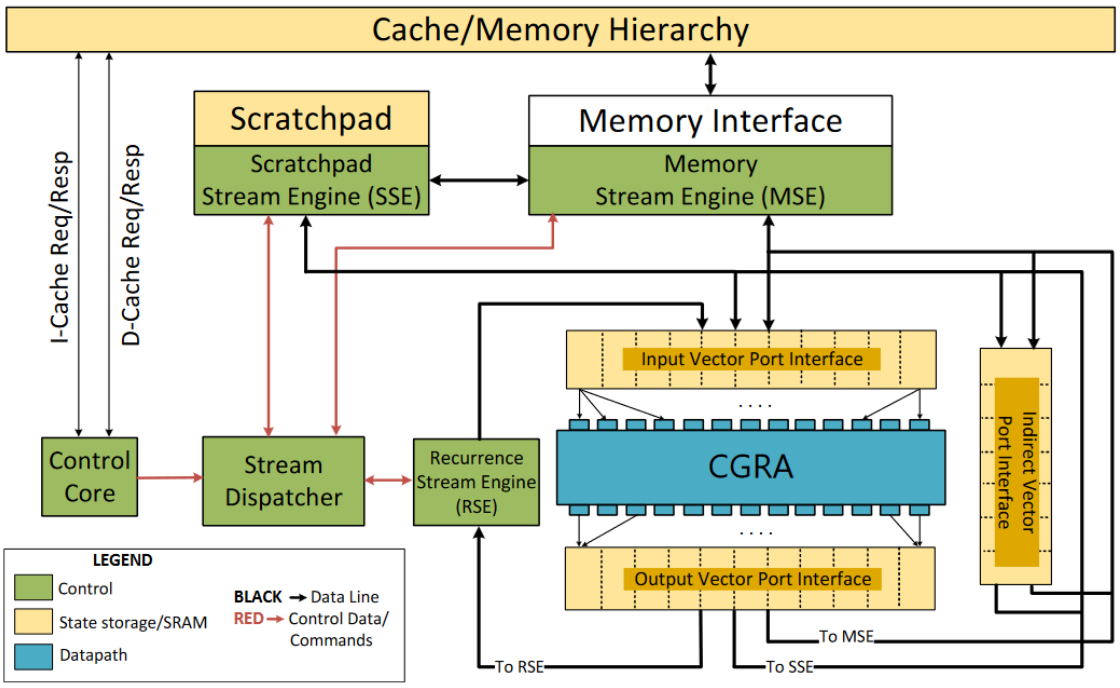


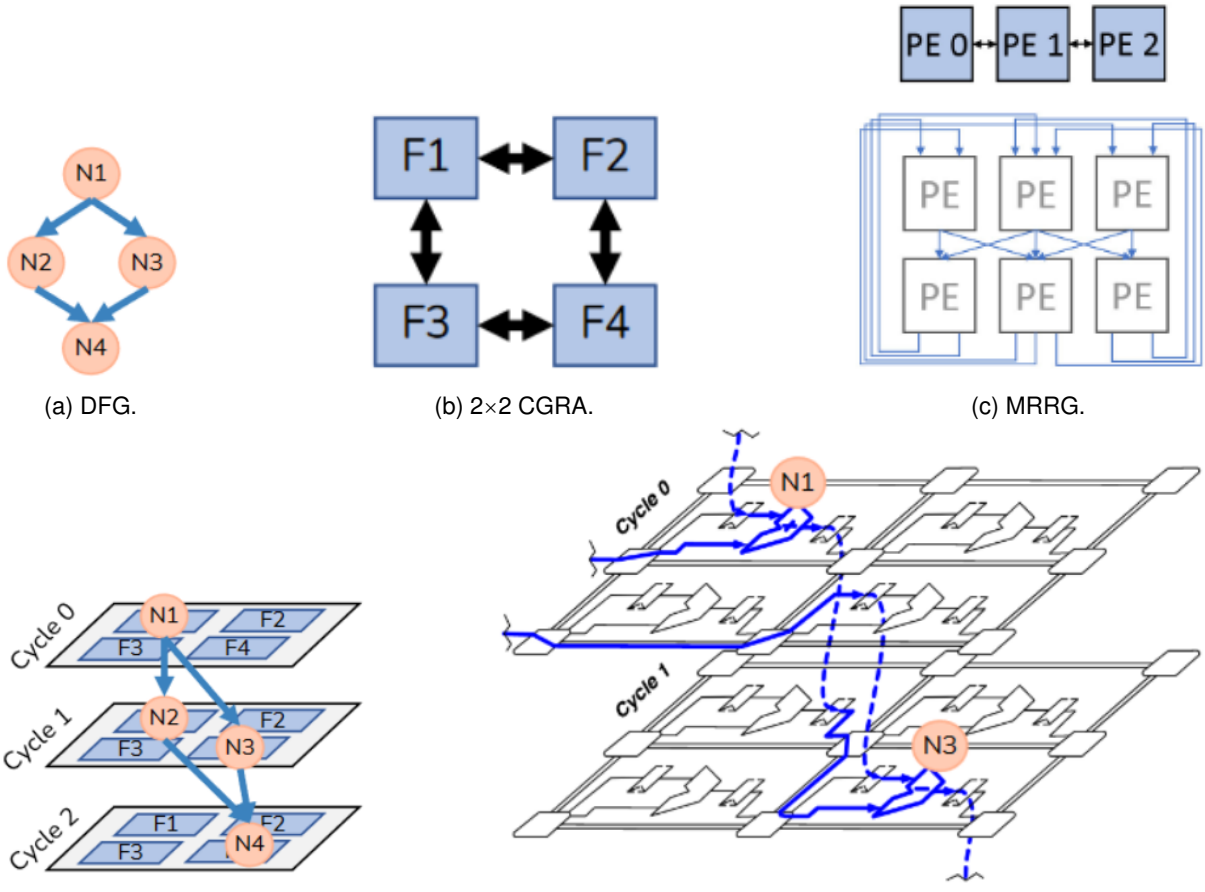
Figure 2.13: Softbrain microarchitecture overview [23].

2.2.3 Compiling for CGRAs

While the CGRAs microarchitecture has been relatively straightforward along the last years, the compilation process for these architectures presents a computationally complex challenge since it usually requires solving an NP-complete problem [50]. The main objective of a CGRA compiler is to take a compute-intensive loop kernel and generate a schedule that is feasible on the target architecture, while maximizing the throughput. The loop kernel is usually represented as a DFG where nodes symbolize operations and edges denote data dependencies.

In practice, the CGRAs mapping is a place and route problem, where placement involves assigning DFG operations to FUs in space and time, while adhering to dependency constraints; the routing entails mapping data signals between operations using wires (for spatial connections) and registers (for temporal connections).

This process leverages instruction-level parallelism (ILP) within loop kernels by employing a software pipelining technique denoted as modulo scheduling [51]. This technique employs ILP at the iteration level, by overlapping independent operations in consecutive iterations, executing them simultaneously. The goal of modulo scheduling is to generate a repeating schedule that allows multiple iterations to be initiated in a pipelined fashion while minimizing the time it takes between the initiation of consecutive iterations. This time interval is called the initiation interval (II).



(a) DFG. (b) 2×2 CGRA. (c) MRRG.

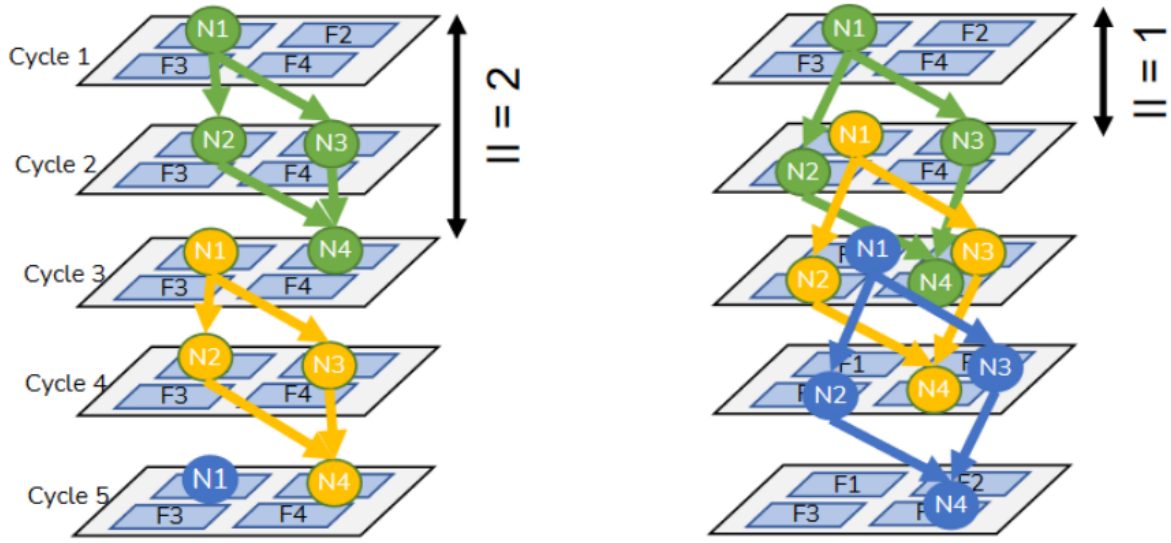
(d) Spacio-temporal mapping. (e) Resources used by mapping.

Figure 2.14: Mapping process [52].

The mapper starts with the minimum II, based on the characteristics of the DFG, represented in Fig. 2.14a, which corresponds to a loop kernel previously extracted and transformed into a dataflow representation, as well as the characteristics of the CGRA architecture [53]. The minimum II is typically computed as the maximum between a resource-constrained lower bound and a recurrence-constrained lower bound, as described in [51].

The CGRA architecture, in Fig. 2.14b, is represented as a modulo routing resource graph (MRRG). The MRRG models the essential CGRA components (FUs, RFs, switches and connections) in a time-extended resource graph [53]. This time-expansion allows the modeling of resource usage across multiple scheduling cycles, enabling overlapping execution of successive loop iterations. This model contains both spatial dimensions of the CGRA array, repeated in a time dimension with a size of II, since the schedule repeats at every II. The resources of the PEs in the last cycle of the MRRG wrap around and connect to the PEs in the first cycle, as depicted in Fig. 2.14c. This wrap-around models the cyclic nature of modulo scheduling, where operations in one iteration may produce results consumed in the next. Because of this, the resulting schedule is called a modulo schedule.

The compiler aims to find a valid modulo schedule of the DFG on the MRRG, increasing the II until a valid mapping (without resource conflicts) is found, represented in Fig. 2.14d. This mapping represents the configurations of the CGRA's resources at each cycle, as shown in Fig. 2.14e.



(a) Low performance mapping with II = 2 cycles.

(b) High performance mapping with II = 1 cycle.

Figure 2.15: Software pipelined schedule [52].

Due to the computational complexity of the mapping process, optimal solutions cannot often be found in practical time, with routing being the most intensive step of the algorithm [54]. Non-optimal solutions can be found much faster by increasing the II of the MRRG, which causes the DFG nodes to have a sparser placement, in turn making it easier for non-conflicting routes to be found. However, these non-optimal solutions result in lower throughputs since the II is larger, as exemplified in Fig. 2.15.

Several categories of mapping algorithms have been proposed to tackle this problem, differing mainly in how they search for a valid schedule on the MRRG. These include heuristic-based approaches [55–

60], graph-based formulations [61, 62], and constraint-based methods using integer linear programming (ILP) [63]. More recent methods propose scalable techniques such as learning-induced [64] and hierarchical approaches [54, 65]. While their internal mechanisms differ, all these methods follow the same general structure described above, aiming to generate valid modulo schedules efficiently.

2.3 Data streaming

In data streaming architectures, the memory access patterns are explicitly described, based on the principle that regular applications have memory access patterns that can be represented by an n-dimensional affine function [66].

By relying either on the programmer's knowledge of the algorithm or a compiler analysis of the algorithm, these patterns can be explicitly encoded into stream descriptors. These explicit descriptors thus allow for the data transfers to be matched with the hardware capabilities, achieving significant performance gains [67].

2.3.1 Unlimited Vector Extension

Stream specification based on the UVE [35] instruction set architecture (ISA) extension has been recently proposed by Domingos et al., combining a descriptor-based representation with a stream configuration ISA interface.

UVE is a custom, SIMD extension developed for the RISC-V ISA. It was designed to reduce the amount of overhead instructions in loop code and improve performance by exploiting data-level parallelism. To achieve this, UVE makes use of stream descriptors to define the memory access patterns. It is also based on a design principlee where the memory instructions are separated from the computation instructions, allowing for a co-located streaming engine to process the descriptors configured in the preamble and autonomously stream the data to the execution units.

The UVE model proposed by Domingos et al. follows an affine function:

$$\begin{gathered} y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times S_k \\ \text{with } X = \left\{x_0, \ldots, x_{dim_y}\right\} \text{ and } x_k \in [O_k, E_k + O_k], \end{gathered} \tag{2.1}$$

where each stream access y is obtained by adding y_{dim} pairs of indexing variables x_k and stride multiplication factors S_k to a base address y_{base}. Each indexing variable x_k represents an integer in the range $[O_k, E_k + O_k]$, with O_k and E_k representing the indexing offset and number of data elements for dimension k. Further complexity can be achieved by setting the base address and the offset value to the result of another affine function.

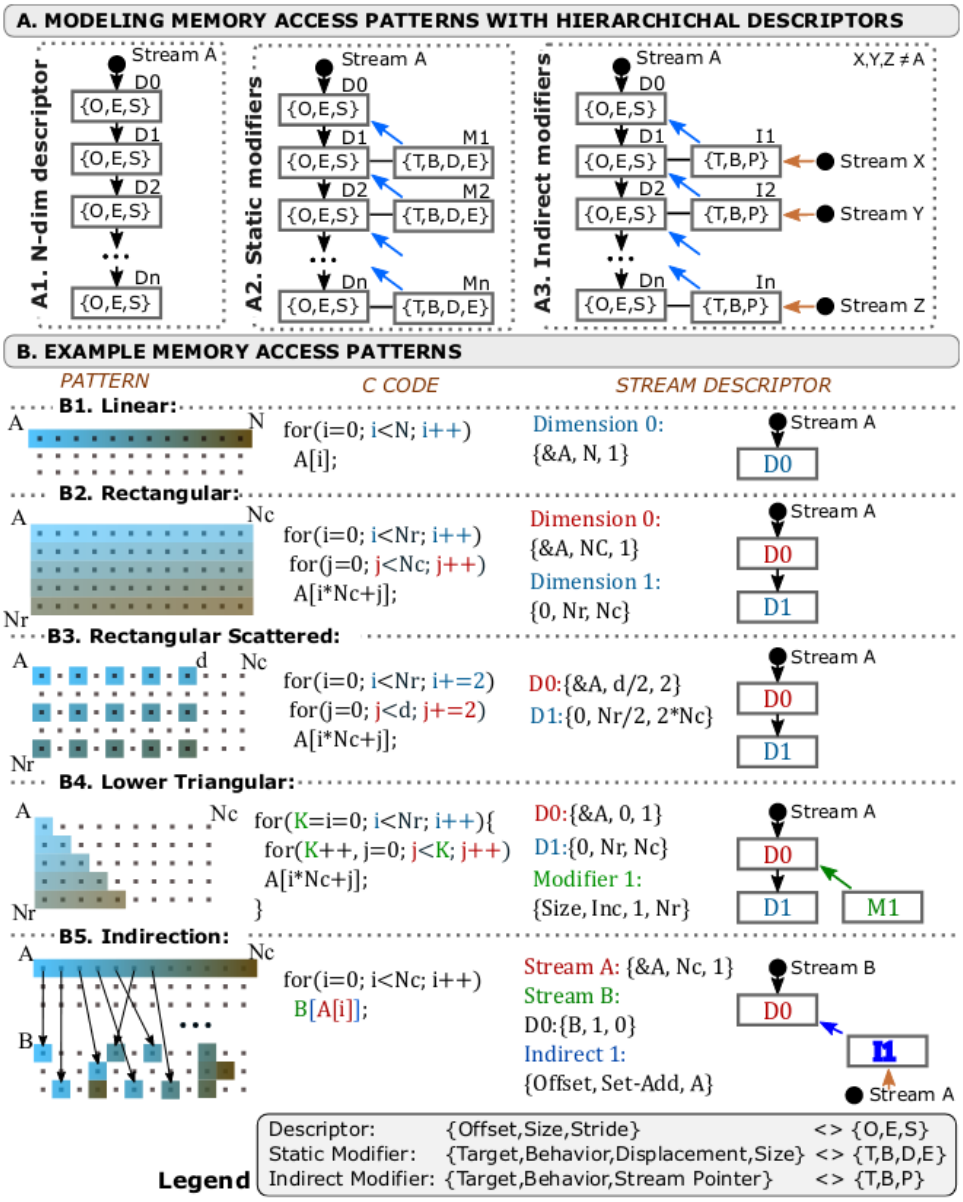


Figure 2.16: UVE memory access pattern representation with hierarchical descriptors and examples cases.

Dimensional descriptors

In UVE, memory access patterns are described by a hierarchy of dimensional descriptors and modifiers. Each dimensional descriptor is associated with a dimension (k), and consists of a tuple of three parameters: the offset (O_k), describing the base address for the dimension; the stride (S_k), representing the interval between consecutive elements; and the size (E_k), which determines the number of elements in the dimension.

A single-dimensional descriptor, exemplified in Fig. 2.16B1, represents a linear memory access pattern, which can either be strided or not. By hierarchically combining multiple descriptors, n-dimensional patterns can be described, such as the rectangular and scattered rectangular patterns exemplified in Figs. 2.16B2 and 2.16B3. The descriptors are combined in a cascaded scheme, analog to a nested

programming loop.

To describe more complex patterns, the use of modifiers is required, with UVE supporting two types of modifier descriptors: static modifiers and indirect modifiers.

Static modifiers

Static modifiers allow for non-linear modifications to dimensional descriptors, by incrementing or decrementing a parameter of a given dimensional descriptor.

They are composed of four fields: the target, which specifies the parameter to be modified; the behavior, which determines if the target parameter gets incremented or decremented; the displacement, defining the amount of the increments or decrements; and the size, which represents the number of iterations in which the target gets modified.

A single static modifier is associated with a given dimensional descriptor, and the modification is applied every time that descriptor is iterated. To simultaneously modify more than one parameter of a dimensional descriptor, multiple static modifiers can be configured for the same dimension, up to a maximum of three (one for each parameter).

An example of a lower triangular memory access pattern is depicted in Fig. 2.16B4, which is achieved by attaching a modifier targeting the size parameter of the inner dimension. This modifier is configured to increment the size (by one), every time the descriptor for the inner dimension gets iterated. It is also configured with its size equal to the size of the outer dimension, meaning that the modifier will be applied in all iterations.

Indirect modifiers

Further complexity can be achieved by the use of indirect modifier descriptors. These modifiers allow for the contents on one stream to modify parameters of another stream. Indirect modifiers are similar to static modifiers, but the displacement is the value of the origin stream, called the dynamic displacement.

Indirect modifiers are represented by a tuple of three parameters: the target, like in the static modifiers, selects the target parameter to which the modifier applies; the origin data stream pointer, which defines which stream's values are used as the dynamic displacement; and the behavior, which can be to add or subtract the dynamic displacement to the target, just like in the static modifiers, or additionally to set the target parameter. No size parameter is explicitly required, since the relation between the stream and the origin stream implicitly sets the size to the size of the origin stream.

The use of indirect modifiers allows the representation of rather complex patterns, such as indexed scatter-gather and indirect patterns, as depicted in Fig. 2.16B5.

Compiler support

In order to support the automatic generation of UVE code, Neves et al. have recently proposed a dedicated compilation flow based on an extension of LLVM [68]. Their tool analyzes the LLVM intermediate representation (IR) of the target application in order to extract the DFGs and memory access patterns.

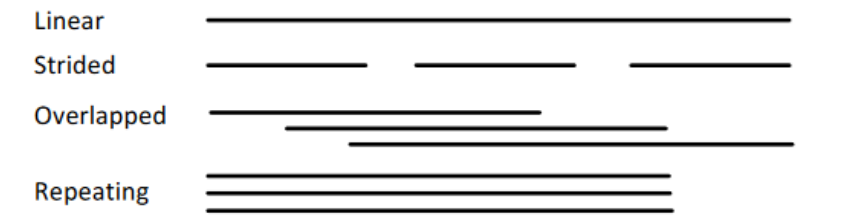
With this data, it generates a high-level data streaming representation, which is directly compiled in UVE instructions and linked to conventional machine code.

2.3.2 Related works

Stream-Dataflow Acceleration

In the same scope of work where Nowatzki et al. present the Softbrain accelerator [23], whose microarchitecture was briefly explored in subsection 2.2.2, these authors also propose a data streaming scheme to support efficient data transfers between the accelerator and to memory.

To support their stream-dataflow execution model, the authors define a stream specification based on an extension of the control core ISA. This stream specification is able to express memory access patterns for two-dimensional affine accesses, which can result in a contiguous, strided, overlapped or repeating pattern, as shown in Fig. 2.17. It can also express indirect patterns, where one stream's data is either treated as a starting address or as a pointer value of another stream. Additionally, the specification allows for sending streams of constants.



(a) Supported types of access patterns.

(b) Affine stream abstraction.

Figure 2.17: 2D affine memory access patterns supported in Stream-Dataflow Acceleration [23].

Cascade

In Cascade [27], another work previously described in subsection 2.2.2, the data streams are represented as tuples of configuration values, configured via a configuration bus (just like the rest of the accelerator), instead of using custom instructions. The stream tuple definition supported in Cascade allows for 2D strided patterns, functionally equivalent to the code in Fig. 2.18. Streams with more than two dimensions are handled by an interative reconfiguration of the SE by the host CPU.

2.4 Tools and frameworks

In this section, we introduce some of the works that have developed tools and frameworks used in the development of this thesis.

```
for (i = 0; i < max_i; i = i + 1) {
    for (j = 0; j < max_j; j = j + 1) {
        addr = AS_i * i + AS_j * j + SA;
    }
}
```

Figure 2.18: Cascade address computation by stream tuple.

2.4.1 Morpher Framework

In the realm of reconfigurable computing, the success of FPGAs has been partly driven by the quality and availability of several design tools, such as the AMD Vivado Design Suite or the open-source Verilog to Routing (VTR) computer-aided design (CAD) flow [69]. However, tools for CGRAs are still in their early stages. The Morpher Framework [70] addresses this gap. It is an open-source, end-to-end framework for compiling and simulating CGRA-based systems, easing both the design space exploration and application development.

Morpher takes compute-intensive kernels as input and compiles them onto a CGRA architecture model specified by the user, achieving on higher mapping quality at shorter compilation times when compared to other existing open-source tools [70]. It operates through the following steps:

DFG generator

This first step takes an application source code with an annotated kernel and extracts the corresponding DFG. This step is implemented as an LLVM pass and supports kernels with control divergence and recurrence edges, allowing for it to be used on more complex real-world kernels.

CGRA mapper

By using a flexible architecture description language (ADL), Morpher allows users to define their own architecture models comprised of model hardware blocks like PEs, RFs, ALUs, and memories, as well as their respective connectivity.

Morpher currently supports three state-of-the-art mapping methods: a heuristic-based approach [55], a simulated annealing (SA)-based approach [71], and a novel learning-induced approach [64]. Alternatively, the modular code base of Morpher allows researchers to bring their own mapping methods to the framework.

Test data generation and simulation

To eficially generate the simulations' test data, Morpher also supports automatic instrumentation for capturing live-in and live-out variables. Additionally, it includes a simulator capable of verifying the generated mapping on a model CGRA, validating the results against the expected test data.

However, all these functions are currently only implemented for variations of the HyCUBE architecture [26].

2.4.2 Chipyard Framework

Chipyard is an open-source framework for developing Chisel-based SoCs. It is actively developed by the Berkley Architecture Research Group at the University of California, Berkeley [37], and it provides designers with an agile design methodology that alleviates the increased design costs of custom silicon architectures. Chipyard does this by offering composable, generator-based semiconductor intellectual property (IP) blocks that maintain design intent and integration consistency across multiple stages of the hardware development flow. This allows designers to easily develop, simulate, and validate architectures within a full SoC integration, thus making it possible to evaluate system-level implications [37].

Chisel is an open-source hardware description language (HDL) that expands the Scala programming language to support hardware construction primitives, allowing designers to write complex and highly parametrizable circuit generators with the power of a modern programming language. The Chisel circuit generators are compiled into synthesizable Verilog [72].

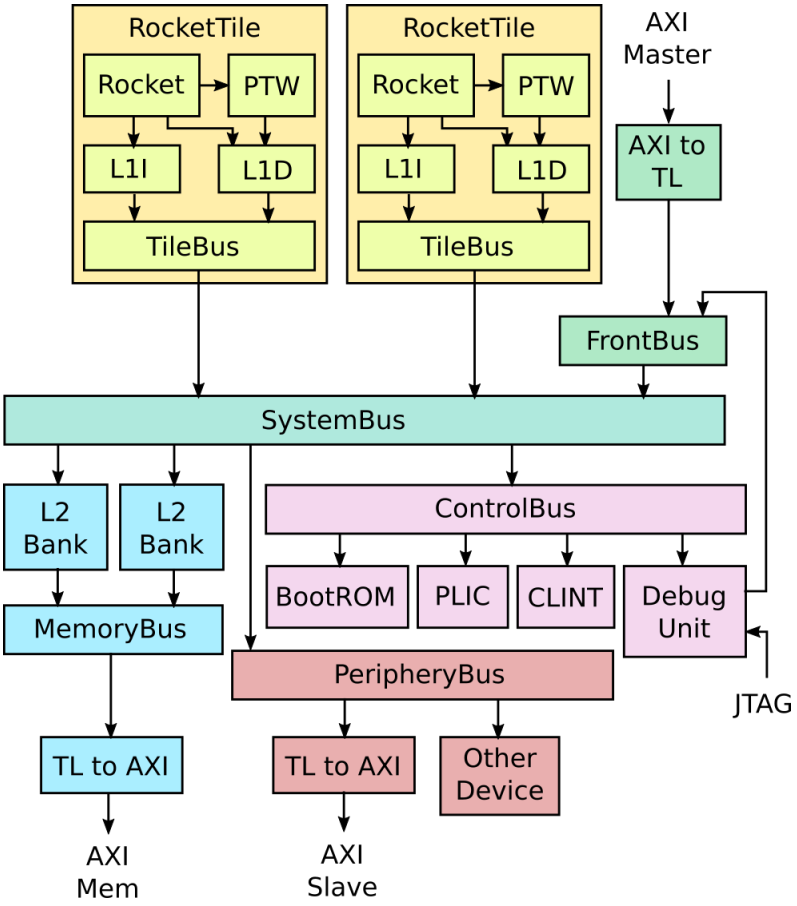


Figure 2.19: Diagram of a typical Rocket Chip system [46].

The Chipyard framework is centered arround the Rocket Chip SoC generator [73], which includes various components such as a CPU, memory system, peripherals, and an accelerator interface called the Rocket custom co-processor (RoCC) [46], as illustrated in Fig. 2.19. By default, the used CPU core is the Rocket Core, which is a 5-stage in-order scalar processor. Chipyard also supports another Berkley CPU project, the Berkeley Out-of-Order Machine (BOOM) Core [74], a more complex out-of-order superscalar processor that can be used as a drop-in replacement for the Rocket core. Recently,

Chipyard added support for other cores such as CVA6 [75], Ibex [76], Sodor [77] and Shuttle [78].

Additionally, Chipyard also includes several accelerator generators, such as Gemmini [9], a generator for DNN accelerators; Hwacha [79], a vector unit architecture; NVDLA [15], a deep-learning accelerator developed by NVIDIA; a SHA3 accelerator, exemplifying the use of the RoCC interface; and a fast Fourier transform (FFT) accelerator generator, to exemplify a memory-mapped I/O (MMIO)-based interface accelerator.

Besides the RTL generators, Chipyard also includesother tools for software RTL simulation, FPGA-accelerated simulation, software workload generation for bare-metal and Linux-based systems, as well as wrappers for common very large-scale integration (VLSI) flows.

The FPGA-accelerated simulation (used in the scope of this thesis) is provided by FireSim [38], which is an open-source cycle-accurate full-system hardware simulator. FireSim allows for much faster simulation speeds than traditional software RTL simulators, allowing for full-system simulations including realistic memory models and network models. It runs on Amazon EC2 F1 FPGA-enabled cloud instances or optionally on local machines with attached FPGAs.

2.4.3 Clava source-to-source compiler

Clava [80] is a C/C++ source-to-source compiler built atop the LARA engine [81]. It allows for the relatively easy specification of complex code analysis and transformations using JavaScript scripts.

It leverages the Clang's parser to generate a C/C++ abstract syntax tree (AST), which can then be directly transformed. After all transformations are applied, Clava emits valid C/C++ code.

In the particular scope of this thesis, this Clava's approach allows for a more nimble implementation of code analysis and transformations, compared to modifying the compiler to implement custom passes. This allows for an initial development and testing without the effort and in-depth knowledge required for modifying the compiler.

2.5 Summary

This chapter began by describing the landscape of computing accelerators as a response to the power wall in traditional von Neumann architectures. In particular, fixed dataflow accelerators in the form of ASICs offer significant gains in energy efficiency by specializing their hardware to specific applications or domains. Examples include Google's TPU, NVIDIA's Tensor Cores, and the Gemmini accelerator. However, this specialization comes at the cost of flexibility, limiting their usefulness in evolving workloads.

As an alternative, reconfigurable accelerators have emerged to support a wider range of applications by allowing the function of the accelerator to be adapted. Among these, CGRAs have been gaining attention for combining the flexibility of FPGAs with the efficiency of fixed-function accelerators. This chapter reviewed several state-of-the-art CGRA architectures, highlighting their strengths and discussing limitations related to memory access and compiler support.

To mitigate some of these limitations, the concept of data streaming was introduced. By describ-

ing memory access patterns explicitly, data streaming decouples memory access from computation, enabling better specialization and utilization of resources. A key example is the UVE data streaming solution developed by the HPCAS research team, which provides a compact and expressive descriptor model for regular memory access patterns.

Despite these advances, several key challenges remain. Most CGRA-based systems still perform memory accesses and address generation within the compute fabric, causing contention for functional units and interconnects, and ultimately reducing efficiency. Moreover, few systems fully decouple memory from computation or support flexible, streaming-based access patterns. There is also a lack of open, end-to-end compilation flows that can extract both the computation and memory behavior from high-level code and target a complete, reconfigurable system. These gaps motivate the work presented in the next chapter, which proposes a new accelerator architecture based on CGRA and tightly integrated with the UVE streaming model, supported by a compilation toolchain and system-level infrastructure.

To support the development and evaluation of the proposed system, this chapter also introduced several key tools and frameworks. The Morpher framework is used for extracting DFGs from code and for CGRA mapping; Chipyard provides the SoC generator for system-level integration; and Clava offers a flexible source-to-source compiler for analyzing and transforming C/C++ code.

3

Hardware-software architecture of the proposed computational infrastructure

Contents

To address the challenges presented in the previous chapter, we propose an architecture consisting of a CGRA with decoupled memory access supported by data streaming. This architecture, abstractedly depicted in Fig. 3.1, offloads memory access to a dedicated unit, called the streaming engine (SE), and uses the CGRA's datapath exclusively for computation.

The SE is responsible for providing the data required for the computation to the CGRA's datapath, at the right time, and for consuming the outputs of the computation to store them back in memory. The inbound streams of operands are called *load streams*, while the outbound streams of results are called *store streams*.

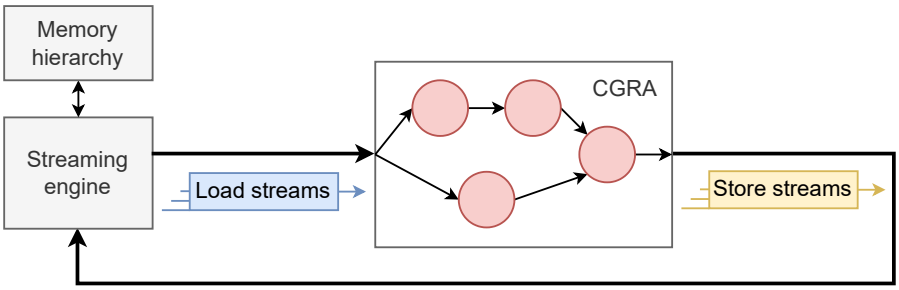


Figure 3.1: Abstract architecture.

Besides a comprehensive description of the architecture abstraction, this chapter introduces the computational model that supports the proposed architecture as well as the key components of hardware and software that compose the system. The following chapters will detail the design and implementation of each of the system's elements.

3.1 Computational and abstractions model

To support the envisaged architecture, the proposed computational model is based on two main abstractions: DFGs to represent the computation, and data streams to represent data movement.

The following subsections elaborate on each abstraction in turn. First, data streams and dataflow graphs are introduced in more detail, showing how they describe memory access and computation. Then, vectorization is presented as a strategy to enhance parallelism and data reuse. This is followed by an explanation of how these abstractions are mapped to the hardware architecture and executed, and the section concludes by describing how the resulting hardware and data stream configuration is encoded and loaded into the system for execution.

3.1.1 Data streaming and dataflow

To demonstrate how the data streaming and dataflow abstractions relate, a simple kernel that computes the dot products between each row of a matrix A and the corresponding row of matrix B will be used, as depicted in Fig. 3.2 and assuming matrices with $R \times N$ elements.

In the kernel code (Fig. 3.2a), the loop headers define the memory indexing and loop control, while the loop body specifies the computation. By abstracting the memory indexing as a data stream, several

advantages emerge:

- **Decoupled memory access**: Abstracting memory indexing allows memory access to be performed independently and in parallel with computation, while also enabling prefetching.
- **Accurate prefetching**: The memory access patterns are explicitly described in the data stream descriptors, enabling prefetching without miss-predictions.
- **Indexing-free computation representation**: The abstraction leads to a computation representation that is free of any iteration variable calculation and address generation overheads.
- **Linearization of computation**: From the perspective of the execution datapath, the data stream is perceived as a linear sequence of data elements, independent of the eventual complexity of the multidimensional memory access patterns, as shown in Fig. 3.2b. This also simplifies the control flow, as the loop access patterns are encoded within the data stream descriptors.

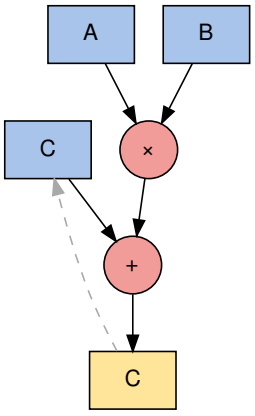
The computation itself is abstracted as a DFG, which is an acyclic graph where the nodes represent operations and the edges denote data dependencies. In this model, source nodes represent the input data streams, and sink nodes represent the output data streams. Cyclic data dependencies are implicitly represented by store data streams that are subsequently read by load data streams.

```
void kernel_dot_product() {
  /* Memory indexing / Loop control */
  for (int r = 0; r < R; r++)
    for (int i = 0; i < N; i++)
  /* Computation */
      C[r] += A[r][i] * B[r][i];
}
```

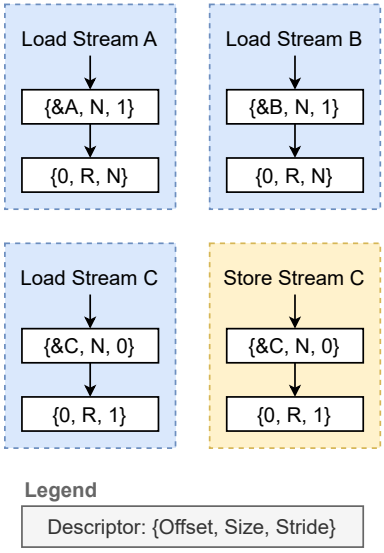
(a) Original kernel code.

```
void kernel_dot_product() {
  for (int idx = 0; idx < R * N; idx++)
    stream_C[idx] += stream_A[idx] * stream_B[idx];
}
```

(b) Linearized kernel.



(c) Computation DFG



(d) UVE data stream descriptions.

Figure 3.2: DFG and streaming abstractions. The kernel shown computes the dot product for R vectors with size N.

Figure 3.2c shows the DFG for the dot product computation, with the implicit recurence between

iterations is depicted by the dashed arrow. The data streams are described in Fig. 3.2d, using UVE descriptors.

By employing these abstractions, the computational model exposes the decoupled memory interface provided by data streaming, and the co-existing computation of dataflow architectures.

3.1.2 Vectorization

While the previously discussed abstractions lay the basis of the envisaged model, they are insufficient to fully exploit the parallelism of the considered architecture. Unless the computation DFG has many nodes, the utilization of the execution datapath will be low. This is because the CGRA's datapath contains multiple functional units that can operate in parallel, but a small DFG may not expose enough independent operations to keep all units active. Consequently, a portion of the hardware remains idle during execution.

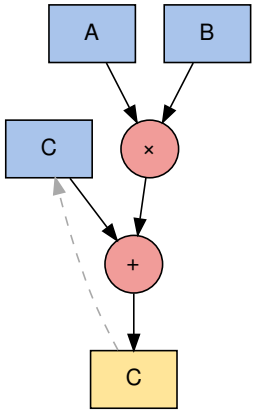
Furthermore, the arithmetic intensity of the computation is low, as the data is repeatedly loaded and stored in each iteration, instead of being reused within the datapath when possible. This means that the number of memory accesses per computation is high, making the system more bound by memory bandwidth than by compute capacity.

```
void kernel_dot_product() {
  for (int r = 0; r < R; r++)
    for (int i = 0; i < N; i++)
      C[r] += A[r][i] * B[r][i];
}
```

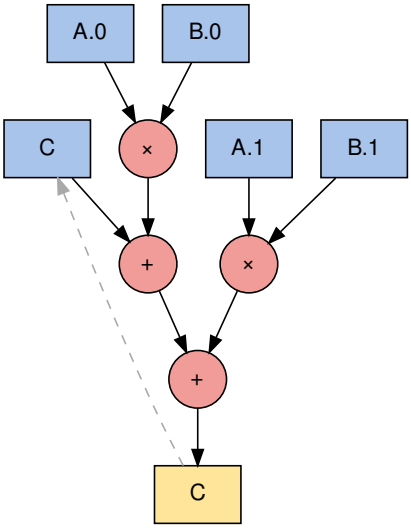
(a) Original kernel code.

```
void kernel_dot_product() {
  for (int r = 0; r < R; r++)
    for (int i = 0; i < N; i += 2) {
      C[r] += A[r][i + 0] * B[r][i + 0];
      C[r] += A[r][i + 1] * B[r][i + 1];
    }
}
```

(b) Unrolled kernel code (unrolled factor of 2).



(c) Original DFG.



(d) Unrolled DFG (unroll factor of 2).

Figure 3.3: Kernel unrolling.

To address these limitations, we propose the extensive application of vectorization in the computation, by unrolling the computation DFG as much as possible, as illustrated in Fig. 3.3. This increases the number of nodes in the DFG, allowing for a parallel execution of the computation. More importantly, it also increases the arithmetic intensity of the computation, as data dependencies from subsequent iterations are now exposed in the DFG, allowing for data reuse within the datapath.

However, each source and sink node in the DFG still represents a data stream of scalar elements. We denote these streams as microstreams. Generating all these microstreams independently would result in prohibitive hardware overheads and neglect the inherent data locality of the computation, as data elements in microstreams would be accessed independently despite their locality in the original computation.

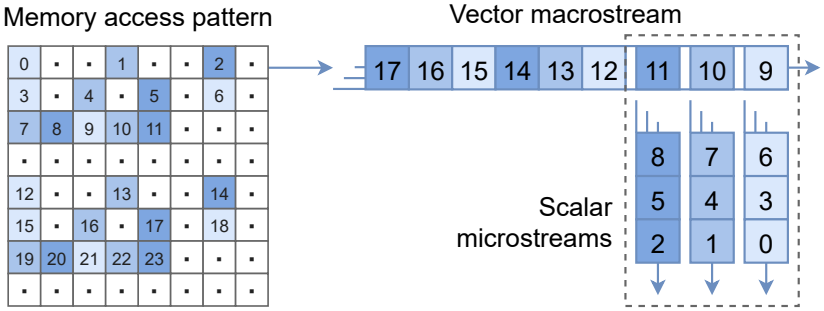


Figure 3.4: Microstreams and macrostreams.

Instead, we propose to coalesce these microstreams into vector streams, which we call macrostreams. This resolves the overhead problem, by only requiring the generation of the macrostreams and preserving the data locality of the computation. Fig. 3.4 shows an example of how the coallescing of microstreams into macrostreams is handled.

3.1.3 Mapping and execution

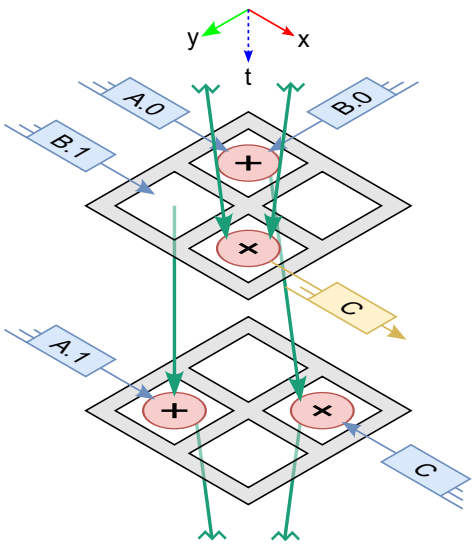


Figure 3.5: Spatio-temporal mapping of the DFG.

The envisaged computational model also relies on an efficient mapping of the computation DFG into the underlying hardware architecture, with this process following a standard approach like the one detailed in Section 2.2.3. Additionally, our DFG abstraction represents microstreams as load and store nodes (source and sink) within the DFG, and so these nodes get mapped to the microstream ports of the CGRA using the same process as the rest of the DFG.

Figure 3.5 illustrates an example of a valid mapping of the DFG shown in Figure 3.3d onto a 2×2 CGRA, with this mapping achieving an II of 2. The nodes in the second cycle wrap around to the first cycle, connecting to the input nodes of the first cycle.

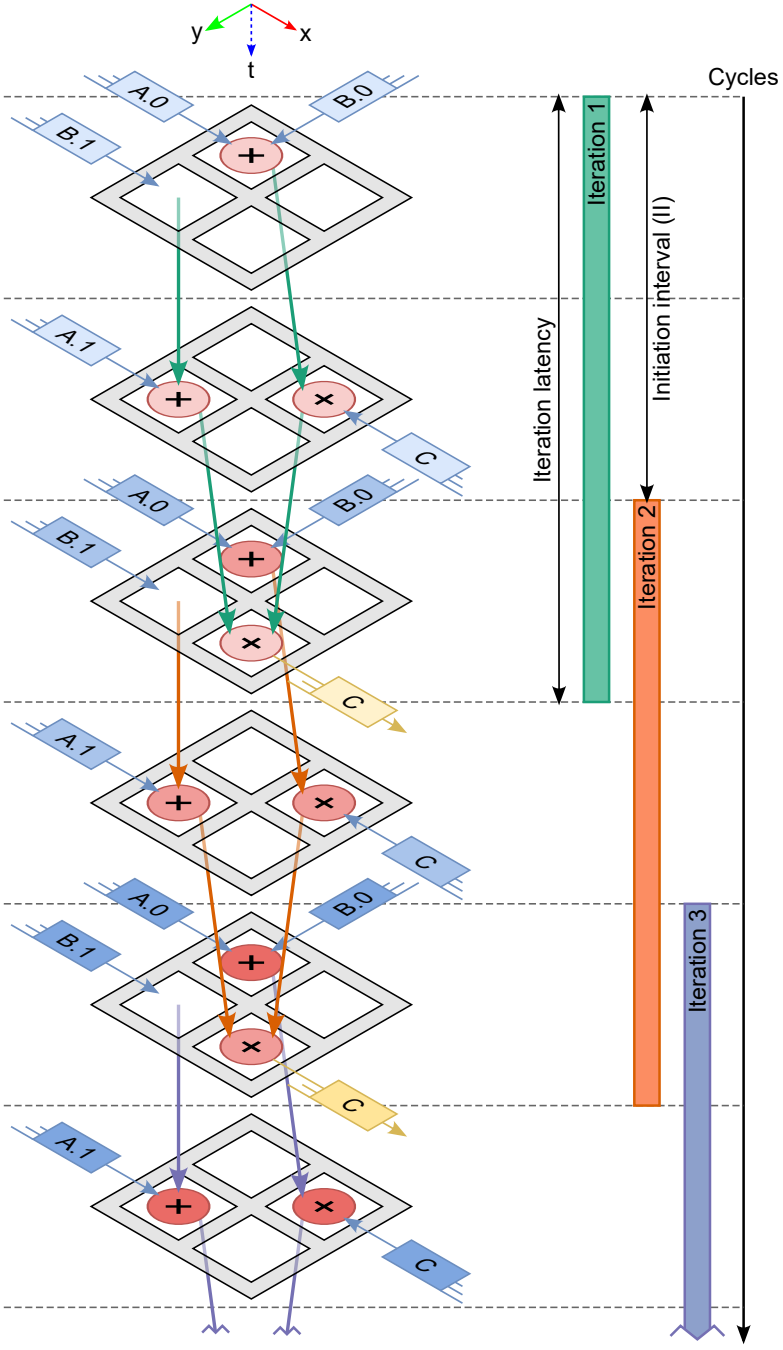


Figure 3.6: Execution example of the repeating schedule.

Hence, each DFG node is mapped to a spatial coordinate and a modulo cycle. While the spatial coordinate represents the PE to which the node is mapped, the modulo cycle is the nth cycle within the repeating schedule. An important and related metric is the initial latency of each node in the schedule, which is the number of cycles required for the first execution of the node. For instance, in the example shown in Figure 3.5, the last addition and the store of microstream C only occur in the first modulo cycle of the second execution of the schedule. Thus, in the first execution of the schedule, the store microstream C is not valid.

Accodingly, the hardware has to periodically repeat the schedule that is determined during the mapping phase in order to execute the aimed computation. Figure 3.6 demonstrates a longer execution of the considered computation, highlighting the data flow and parallel execution of the several operations across the PEs.

It is important to note that this example considers a latency of each iteration that is longer than the II, causing the iterations to overlap. Hence, this schedule consumes a load microstream element and produces a store microstream element every II cycles, repeating until the computation is completed.

3.1.4 Configuration bitstream

Both the mapping and the data stream descriptors need to be encoded in a configuration bitstream. This bitstream contains all the configuration bits necessary to set up the hardware for the computation. At a high level, the bitstream functions analogously to an executable program for the accelerator.

Hence, to execute a given kernel, the host processor signals the accelerator which bitstream should be loaded. The accelerator then loads the corresponding bitstream, reconfigures itself for the computation, and executes the computation. Different bitstreams can be loaded for different kernels, enabling a program to call various kernels on the accelerator.

Since the bitstream transfer procedure takes some time, it is advantageous for the accelerator to overlap the bitstream loading with the execution of computations. Figure 3.7 illustrates an example of how concurrent bitstream loading and computation execution can enhance performance, when compared to a pure sequential approach. Accordingly, our computational model prescribes the overlap of configuration transfer and computation execution.

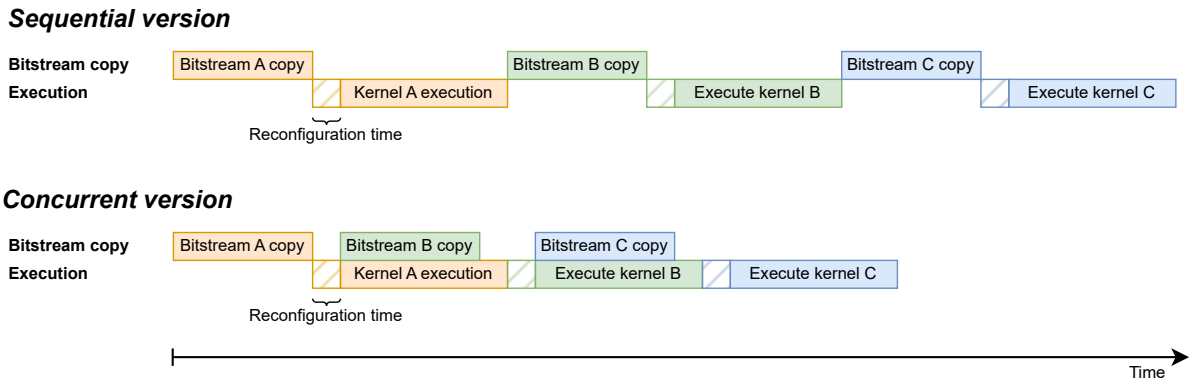


Figure 3.7: Concurrent bitstream loading and computation execution.

3.2 Overall system perspective

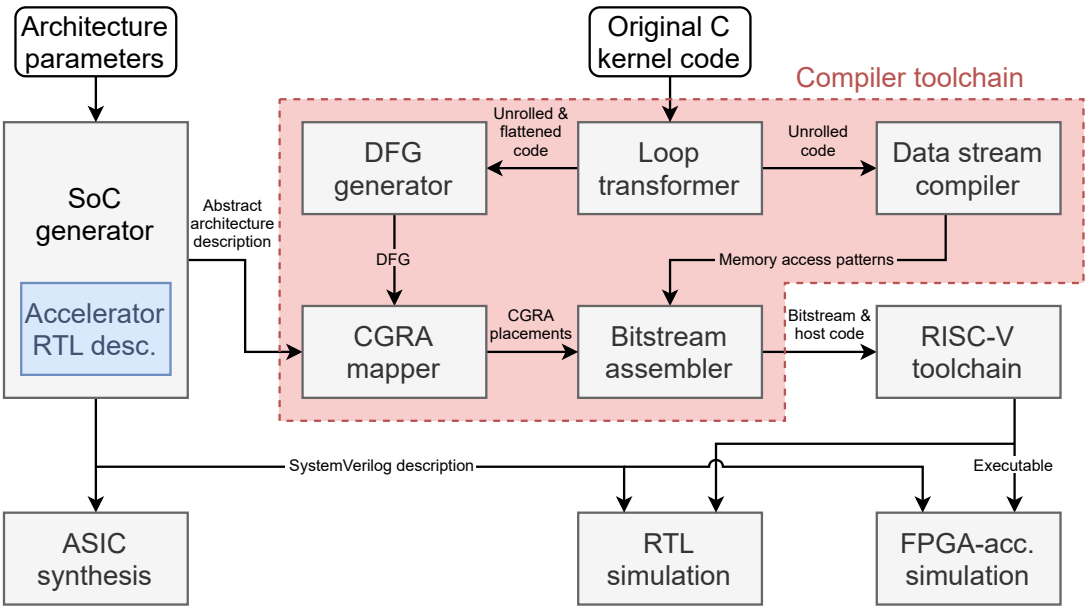


Figure 3.8: Complete system diagram.

Fig. 3.8 depicts the overall system perspective of the proposed architecture and computational model, encompassing both hardware and software components.

The hardware side consists of a parametrizable RTL description of the accelerator architecture, integrated within the Chipyard SoC generator framework. It takes a set of architectural parameters as arguments and generates a SystemVerilog description of a complete SoC containing the accelerator, a RISC-V host processor, memory hierarchy, and other supporting SoC subsystems. Additionally, an abstract architecture representation of the accelerator is produced according to these parameters and will be used by the software side.

The software side comprises a set of tools that take the input kernel (represented as C/C++ source code) and produce an executable file. It consists of several components:

- **Loop transformer**: applies the loop unrolling transformation to the code, in order to vectorize the computation. It also applies a loop flattening pass to produce a linearized version of the code with simplified control flow (no nested loops).
- **DFG generator**: extracts the computation DFG from the unrolled and flattened code.
- **CGRA mapper**: finds a valid space-time mapping of the DFG into the abstract architecture description of the CGRA provided by the hardware generator.
- **Data stream compiler**: extracts the memory access patterns from the unrolled code, generating the corresponding data stream descriptors.
- **Bitstream assembler**: takes the obtained CGRA mapping and data stream descriptors and encodes them within a configuration bitstream. It also generates the host code to load the bitstream.

- **RISC-V toolchain**: compiles the host code into an executable file.

The final SystemVerilog description of the SoC can be directly used for ASIC synthesis or simulation. In the scope of this thesis, we will run two kinds of simulations:

- **RTL simulation**: provides a functional verification of the accelerator and of the executed software.
- **FPGA-accelerated Simulation**: allows a fast and cycle-accurate simulation of the complete system with realistic dynamic random-access memory (DRAM) models.

The implementation of this system will be detailed in the following chapters, with chapter 4 focusing on the architecture of the hardware components, and chapter 5 detailing the software components.

3.3 Summary

This chapter presented the architecture and computational model of the proposed system. It introduced the core abstractions of dataflow graphs and data streams, and showed how they support decoupled computation and memory access. Vectorization was discussed as a key strategy to improve parallelism and arithmetic intensity. The chapter also described the mapping and execution process, as well as the configuration and loading of the bitstream. Finally, it outlined the complete system, including both hardware and software components, that enables the execution of high-level kernels on the proposed accelerator.

4

Microarchitecture

Contents

This chapter provides an in-depth description of the architecture presented in Section 3.1 when integrated as a hardware accelerator of the overall processing system.

The accelerator is implemented as a parameterized RTL description using Chisel. We opted for Chisel because it allows for harnessing the power of Scala language featuring its modern, object-oriented, and functional programming characteristics to describe hardware in a more expressive and modular way when compared to traditional HDLs. Nevertheless, it should be noted that, despite raising the abstraction level of the hardware description, Chisel is not a HLS language, as it still maintains its design intent at the level of cycles, registers, and wires. Along the development toolchain, Chisel is usually compiled into high-quality synthesizable SystemVerilog code, which can be interpreted by most industry-standard electronic design automation (EDA) tools for synthesis, simulation, and verification.

This choice of HDL is also motivated by the ease of integration with the powerful Chipyard framework for SoC generation, which is also written in Chisel. In particular, the accelerator is implemented within a Rocket Chip SoC template, which includes a RISC-V Rocket core, the memory hierarchy, and other SoC subsystems.

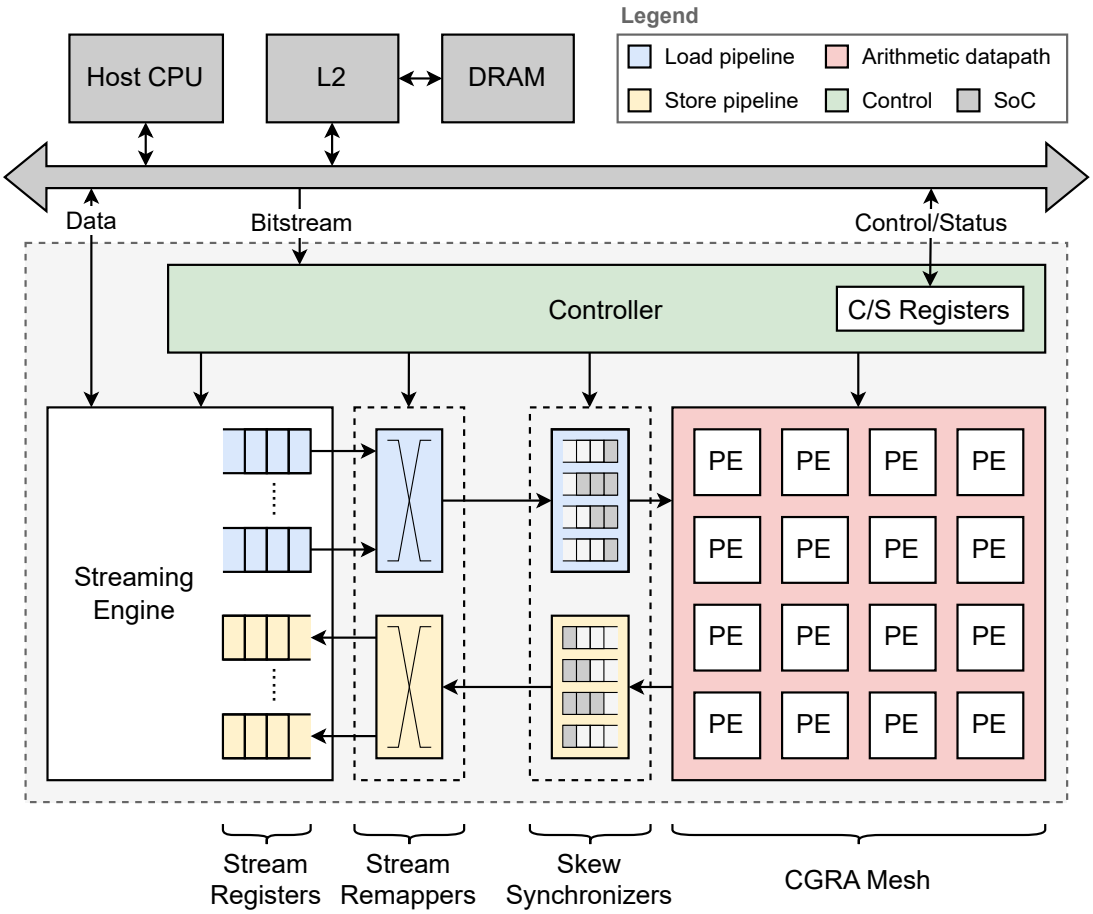


Figure 4.1: Accelerator microarchitecture.

Fig. 4.1 depicts a typical configuration of our SoC architecture implementation, with both the accelerator and the Rocket host processor communicating via shared memory by sharing the bus above the last level cache. This figure also depicts the accelerator itself, composed of four main subsystems:

- The computation mesh, implementing the CGRA-like part of the computation model;
- The streaming engine, implementing the UVE streaming infrastructure;
- The interface between the mesh and the SE, which is responsible for interconnecting the streaming infrastructure with the computation mesh, by routing the data and managing the latency and skew in the dataflow;
- The controller, which manages the state of the accelerator, holds the configuration and handles the reconfiguration process.

4.1 Computation mesh

As in other CGRAs, the heart of the accelerator is a 2D mesh of PEs, interconnected by a network of wires that allow for data transfer between neighbors. Fig. 4.2 shows the architecture of the computation mesh and PEs, with Tab 4.1 listing the parameters taken by the implementation.

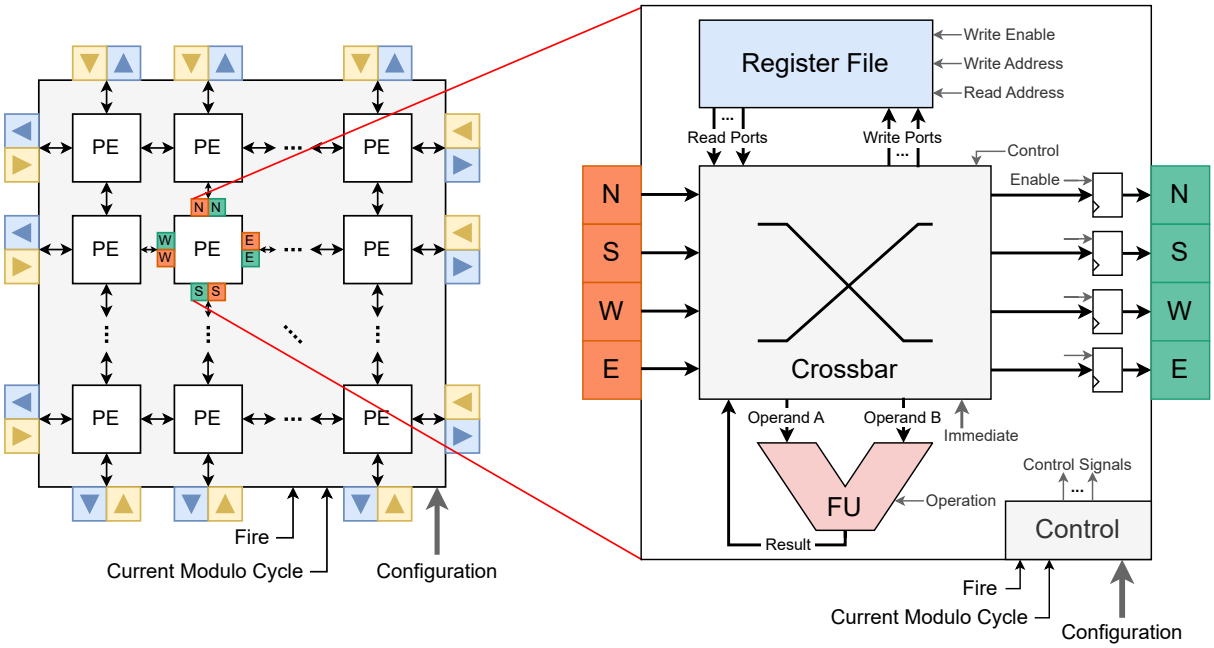


Figure 4.2: Computation mesh and processing element.

The mesh input/output (I/O) consists of the data input and output signals from the PEs at the edges of the mesh and the following control signals:

- `Configuration`: a bundle of configuration words for all PEs.
- `Current modulo cycle`: cycle number in the repeating modulo schedule.
- `Fire`: enable signal that indicates that the current cycle is valid, and computation should be performed.

The `configuration` bundle is split into configuration words for every PE, while the `current modulo cycle` and `fire` signals are broadcast to all PEs.

Table 4.1: Mesh and PE parameters.

Parameter	Description	Example
Data type	Determines the data type in the dataflow	FP32
Mesh rows	Number of PE rows in the mesh	4
Mesh columns	Number of PE columns in the mesh	4
RF size	Number of registers in the RF	4
RF read ports	Number of read ports in the RF	2
RF write ports	Number of write ports in the RF	2

Table 4.2: Operations supported by the FU.

Operation	Function	Result
nop	No operation	-
add	Addition	$A + B$
mul	Multiplication	$A * B$
shl	Shift left	$A \ll B$
lshr	Logical shift right	$A \gg B$
ashr	Arithmetic shift right	$A \ggg B$
and	Bitwise AND	$A \wedge B$
or	Bitwise OR	$A \vee B$
xor	Bitwise XOR	$A \oplus B$

4.1.1 Processing elements

The PE is the base computational unit of the accelerator, having two main functions: perform the arithmetic/logic operations, and route the data between its neighbors.

The PE takes data inputs from orthogonally adjacent neighbor PEs (or, in the case of PEs at the edge of the mesh, from the mesh's external ports). Complementarily, it outputs data to the same neighbor PEs or mesh's external ports. Additionally, it takes the broadcasted `current modulo cycle` and `fire` control signals from the mesh, as well as the configuration word that was targeted for that PE.

Internally, the PE is composed of a FU, a RF, a set of output registers, an internal crossbar switch, and control logic (see Fig. 4.2).

Functional unit

The FU performs arithmetic and bitwise logic operations on two operands, producing a single result. Since our compiler toolchain relies on LLVM to generate the DFG, the operations supported by this implementation are a subset of the LLVM IR primitive binary operations. Tab. 4.2 lists the operations supported by the default implementation of the FU.

The data type parameter determines both the arithmetic type and the width of the operators. Our parameterized implementation supports signed or unsigned integer arithmetic using the standard synthesis tool's implementation of basic integer arithmetic operations. For IEEE-754 floating-point arithmetic,

Table 4.3: PE crossbar allowed data sources per destination type.

Destination	Source
FU operand inputs	Immediate Any RF read port Inputs from neighbor PEs
RF write ports	FU result output Inputs from neighbor PEs
Output registers	FU result output Inputs from neighbor PEs Any RF read port

we use the IP cores from the Berkeley Hardfloat library [82]. Additionally, thanks to the modular and reusable nature of our Chisel description, other data types can be easily supported by simply describing the implementation of their arithmetic operations.

Register file

The RF allows the temporary storage of data between cycles. Its size is parametrized, as well as the number of simultaneous read and write ports. It takes as control signals the read and write addresses, as well as the write enable signals.

Internal Crossbar switch

The internal crossbar allows for the routing of data between the PE's inputs, outputs, and internal components. In order to save on resources, the crossbar is not a true fully connected crossbar switch, but a customized subset, only allowing for some data sources, depending on the destination type, as listed in Tab. 4.3.

The use of such an internal crossbar switch allows the outputs of the PE to be independently selected, not only from the output of the FU or the RF, but also directly from its inputs. Thus, when connected by the external mesh, all the crossbars within the PEs form a NoC, that allows for simultaneous data movement and computation within a PE.

This kind of NoC architecture uses more area than a typical neighbor-to-neighbor (N2N) architecture, previously depicted in Fig. 2.5. In a N2N architecture, every PE output always presents the output of the internal FU, excluding the need for a large crossbar switch. However, this comes at the cost of having to use the FU to perform data routing between neighbors.

Our decision to use such a NoC architecture was also supported by the work of Karunaratne et al., who have shown that NoC-type architectures use less power and are more performant than comparable N2N-type architectures, resulting in a 1.98× better performance-per-watt (on average) [26]. This is explained by the fact that the FUs consume the largest portion of power in a PE, making their use for routing rather inefficient since it leads to more dead cycles where the arithmetic capabilities of FUs are

not being used.

Output registers

Each PE output is buffered through an output register that acts as a pipeline register, ensuring that no combinatorial path extends further than one PE.

Control logic

The control logic takes a configuration word as input, containing a sequence of control signals for all modulo cycles. It selects the appropriate control signals based on the current modulo cycle. Additionally, it also receives the broadcasted `fire` signal, that indicates that the current cycle is valid. This signal is logic ANDed with the `write enable` signals for the RF and output registers, allowing for the state of the PE to remain unchanged in invalid cycles.

4.2 Streaming engine

The streaming engine comprises the implementation of a UVE streaming infrastructure. Since prior work on a UVE SE implementation had already been done in the HPCAS research group, we opted to use the SE implementation developed in Fernandes' master thesis [36]. Although his implementation was originally developed to be used within a general-purpose processor, its parameterizable structure allows the adjustment of some architectural parameters to better fit the design goals of a dataflow accelerator. Additionally, Fernandes' SE implementation is also written as a Chisel RTL description, making the integration within our implementation very easy.

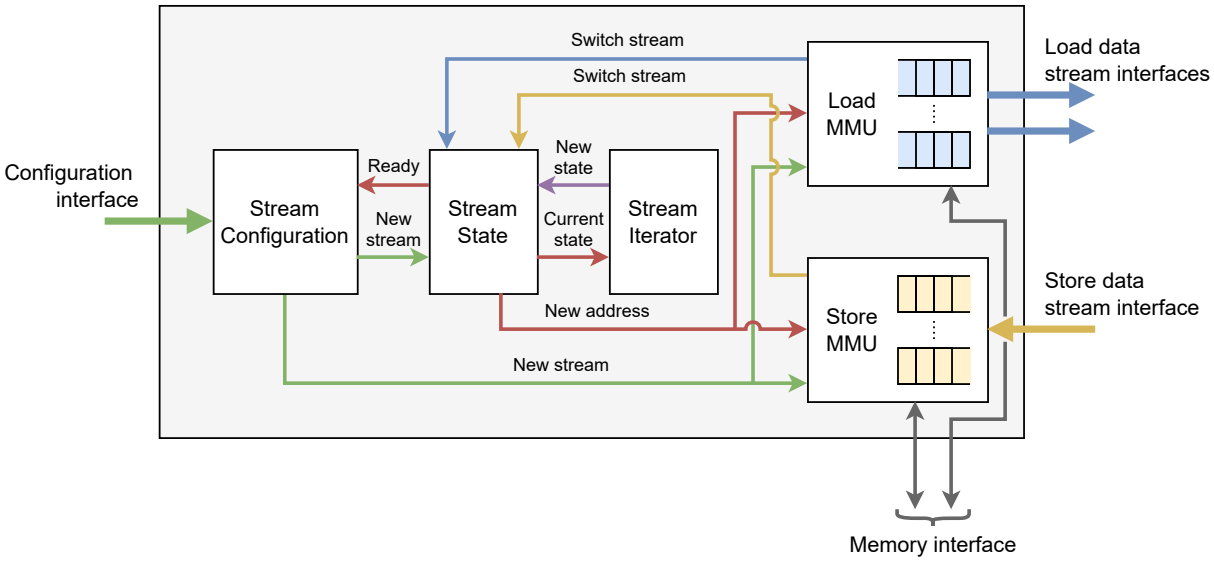


Figure 4.3: Streaming engine internal architecture.

Fig. 4.3 shows a block diagram of the SE architecture. The IO of the SE can be grouped in three main interfaces:

- **Data stream interfaces**: the interfaces that are used to load and store data streams.
- **Memory interface**: consisting of an Advanced eXtensible Interface 4 (AXI4) connection to the memory hierarchy.
- **Configuration interface**: used to program the streaming engine.

Fernandes' master thesis documents his SE's implementation in great detail, but in summary, the SE is composed of the Stream Configuration, the Stream State, the Stream Iterator, the Load memory management unit (MMU), and the Store MMU. The Stream Configuration module receives configuration instructions and decodes them, providing a complete stream description to the Stream State module. This module holds the state of the multiple concurrent data streams and orchestrates which stream to process at each time. When signaled by the Stream State module, the Stream Iterator module performs the address calculations, generating a new memory address and updating the corresponding stream state. Finally, the Load and Store MMUs, take the addresses and make coalesced requests to the main memory, in order to populate the internal vector registers (in the case of load streams) or consume them (in the case of store streams).

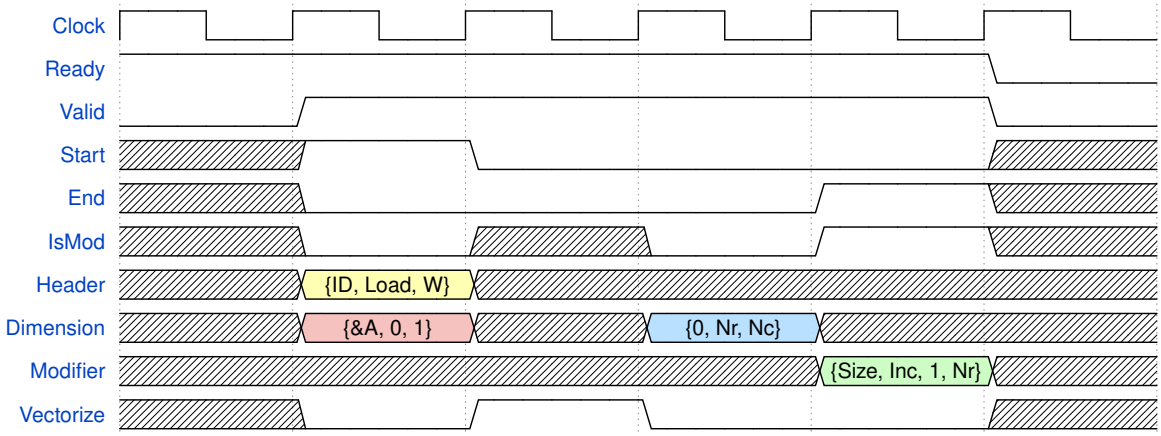
4.2.1 Configuration process

The SE is configured through a sequence of signals on its configuration interface which connects to the accelerator's controller. To ensure synchronization, this interface implements a ready/valid handshake mechanism, where data transfers only occur when both ready and valid are asserted `HIGH`. In this interface, the SE is the consumer and the controller is the producer.

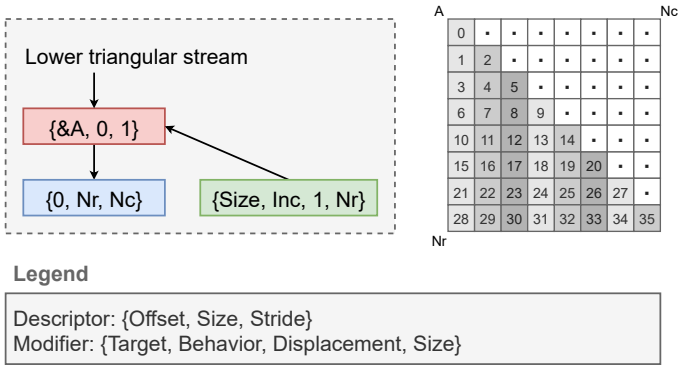
Fig. 4.4 depicts an example of the configuration process. In this example, a load stream with a lower triangular pattern and a data element width of 32 bits is configured. The vectorization point of the data stream is set to the innermost dimension, meaning that every time the innermost dimension starts, a new vector also starts aligned with it. For example, in the depicted lower triangular pattern, if the vector length were set to 2, we would get the sequence of vectors: $\{[0,-],[1,2],[3,4],[5,-],\ldots\}$ where every line starts with a new vector, even if the last vector of the previous line does not contain all valid elements. In contrast, if the vectorization point were set to the outermost dimension, the sequence of vectors would be $\{[0,1],[2,3],[4,5],\ldots\}$, where some vectors contain elements that span different iterations of the innermost dimension.

The configuration protocol obeys the following rules, as depicted in the example waveform in Fig. 4.4a:

- The configuration is iterative, with one dimension or modifier configured per valid cycle.
- In the first cycle, the `start` signal must be set to `HIGH`.
- In the first cycle, the SE reads the stream metadata from the `header` channel. The stream metadata is comprised of an ID field, the stream type (Load or Store), and the data element width (byte, half word, word, or double word).



(a) SE configuration interface waveform.



(b) UVE data stream descriptors.

Figure 4.4: Example of the SE configuration process for a lower triangular pattern data stream. In this example the stream is vectorized in the innermost dimension.

- In the last cycle, the `end` signal must be set to `HIGH`.
- To describe a 1D stream without modifiers, only one cycle is required, and both `start` and `end` signals are set to `HIGH`.
- The `isMod` signal defines if the current cycle configures a new dimension or appends a modifier.
  - When `isMod` is `LOW`, a new dimension is defined by reading the fields from the `dimension` channel, which comprises the offset, size, and stride fields.
  - When `isMod` is `HIGH`, a modifier is defined by reading the fields from the `modifier` channel, which contains the target, behavior, displacement, and size fields. Note that in UVE, modifiers are attached to dimension after the one that they apply to.
- The `vectorize` signal can be set `HIGH` after any dimension definition, to set the vectorization point to the previously configured dimension. In the cycles where the `vectorize` signal is set `HIGH` no dimension or modifier is configured.
  - If the `vectorize` signal is never set `HIGH`, the SE assumes the default vectorization point to the last dimension.

Table 4.4: Load data stream interface.

Signal	Direction	Description
`Ready`	Input	Consumer is ready
`Stream ID`	Input	Consumer sets the ID of the stream it wants to consume
`Valid`	Output	Data is valid
`Vector data`	Output	Stream vector data
`Predicate`	Output	Mask marking what vector bytes are valid data (Unused)
`Done`	Output	Set when all stream elements have been consumed
`Completed`	Output	Set every time a dimension completes (Unused)

Table 4.5: Store data stream interface.

Signal	Direction	Description
`Valid`	Input	Producer holds valid data
`Stream ID`	Input	Producer sets the ID of the stream it wants to store to
`Vector data`	Input	Stream vector data
`Predicate`	Input	Mask marking what vector bytes are valid data
`Ready`	Output	SE is ready to consume a data vector
`Output width`	Output	Data element width (Unused)
`Done`	Output	Set when all stream elements have been consumed by the SE (New)
`Store request successful`	Output	Set after the last store request to memory was successful (New)

4.2.2 Memory interface

As it was referred before, the SE implements an AXI4 [83] master interface to interact with the memory hierarchy. AXI4 is a burst-based protocol, meaning that it is capable of doing multiple data transfers per a single request. This is useful for cases where large data transfers are necessary, such as transferring entire cache lines. The SE takes advantage of this capability since it operates as a prefetcher and as so, it can coalesce memory requests.

4.2.3 Data stream interfaces

The SE produces load stream vectors to the accelerator through the load data stream interface and consumes store stream vectors from the accelerator through the store data stream interface. These channels also use a valid/ready handshake protocol. Tables 4.4 and 4.5 detail the signals that constitute these interfaces.

It is worth noting that our accelerator implementation does not make use of some signals, like the `completed` signal in the load interface and the `output width` in the store interface, since they are not relevant to the execution of our compute model. Additionally, since we do not support predication, the `predicate` signal in the load interface is also ignored. In the store interface, however, we must set the `predicate` signal to mark the valid vector elements according to the configured vector length. To aid in

the control logic of the accelerator, two new signals were newly implemented in the SE store data stream interface: a `done` signal, analogous to the `done` signal in the store interface; and a `store request successful` signal that is helpful to indicate when the last store request to memory was successful, marking that all the data has left the accelerator.

It is also worth noting that the considered SE implementation supports multiple parallel load data stream interfaces, with the number being defined by a parameter. However, only one store data stream interface is currently supported. This is incompatible with our computation model, which requires that multiple store streams are concurrently processed .

To address this limitation, we implemented a time-multiplexing system to multiplex the use of the single store interface by multiple concurrent data streams. While far from ideal, this solution is a viable workaround for most kernels since most do not require concurrent store streams, usually having DFGs that converge to a single result (e.g. reduction).

Additionally, to further promote the simplicity of the controller, we opted to include some control logic to combine all the control signals of the data stream interfaces in more useful and meaningful signals. Furthermore, the load stream multiplexing mechanism and additional control logic are implemented in a wrapper module also containing the SE, as described in the following subsection.

4.2.4 SE wrapper module

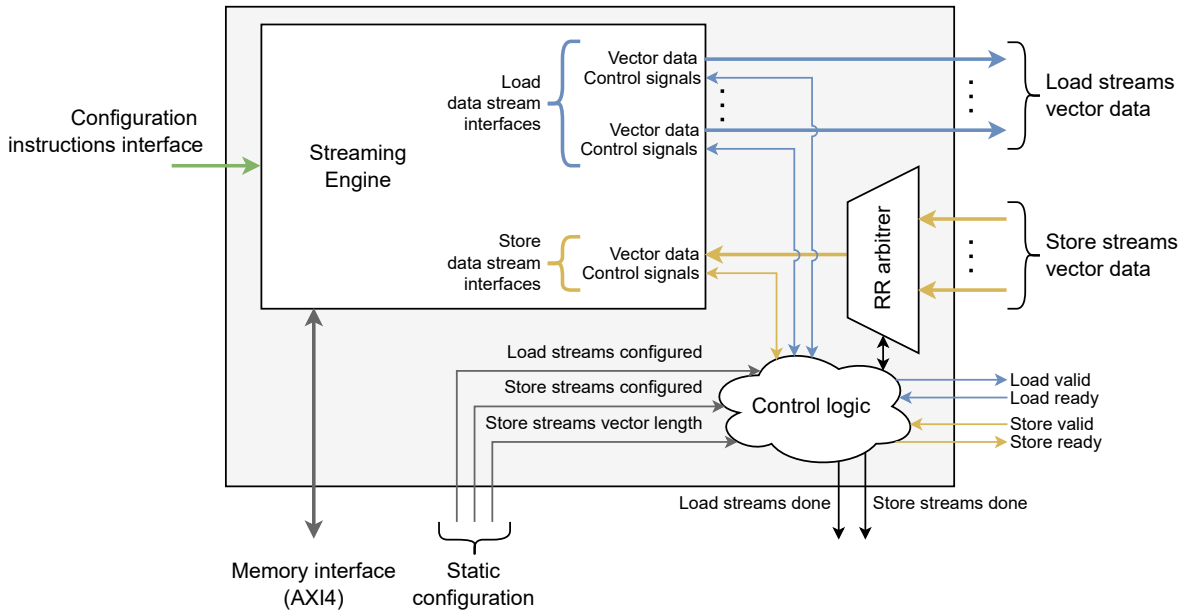


Figure 4.5: Streaming engine wrapper module.

The SE wrapper module, shown in Fig. 4.5, adapts the SE's interface to meet the accelerator's requirements that all streams are always synchronized. It consolidates the individual `ready` and `valid` signals for the load and store interfaces into unified control signals, ensuring that all data streams operate in lockstep. The control logic also takes in the status signals from the SE's data interfaces to generate a single `done` signal for all the store and load streams. Additionally, this logic also generates

Table 4.6: Streaming engine wrapper module control signals.

Signal	Direction	Description
`Load ready`	Input	Load streams consumer is ready
`Load valid`	Output	All load streams data is valid
`Store valid`	Input	Store streams producer has valid data
`Store ready`	Output	All stores streams are ready to be consumed
`Load streams configured`	Input	Signals which load streams are used (obtained from the configuration word)
`Store streams configured`	Input	Signals which store streams are used (obtained from the configuration word)
`Store streams vector length`	Input	Holds the vector length of all store streams (obtained from the configuration word)
`Load streams done`	Output	All load streams have finished
`Store streams done`	Output	All store streams have finished

the appropriate predicate masks for the store streams, according to the configured vector length of each store stream. To generate these signals, the control logic needs to know which load and store streams are to be configured, as well as the vector length of the configured store streams. Consequently, a significant amount of information must be passed into the module from the configuration word. These configuration signals, together with the sequence of signals needed to program the SE, comprise all the configuration data needed by the SE subsystem. Since these configuration fragments differ, in the sense that the SE programming signals are used as instructions to program the SE, and the control logic configuration signals are just directly wired from a configuration word, we refer to the first as SE configuration instructions, while denoting the latter as static configuration. Tab. 4.6 lists all the control signals in the SE wrapper module's I/O.

To implement the time-multiplexing mechanism for the store streams, the wrapper module also includes a round-robin arbiter. The round-robin arbiter selects a store stream at each cycle when the SE's store interface is ready, allowing for the multiple concurrent store streams to sequentially share the single interface.

4.3 SE and mesh interface

According to the adopted computational model, the computation mesh I/O ports handle scalar stream data, hence forward denoted by microstreams. The mapping tools can assign these streams to any port (whether adjacent or non-adjacent) in any modulo cycle. To support this, a dedicated stream remapper module is placed between the SE and the computation mesh ports (see Fig. 4.1) to scatter/coalesce the load/store macrostream vectors into/from microstream scalar elements.

Specifically, the load stream remapper routes macrostream elements from the SE's output registers to a set of registers that hold the mesh inputs for every cycle of the repeating schedule, ensuring that data arrives at the designated data ports in a timely manner. Similarly, the store stream remapper routes

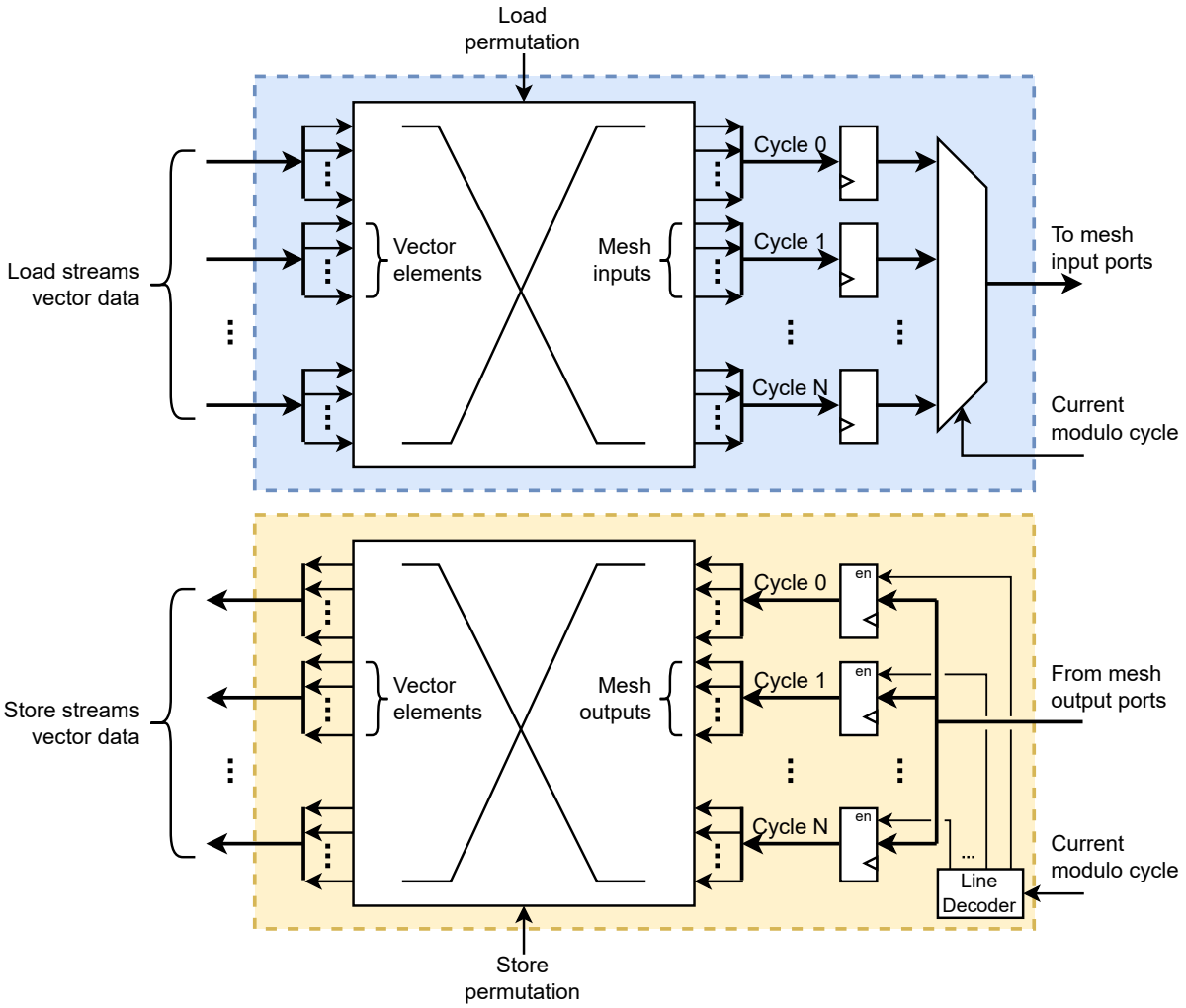


Figure 4.6: Load and store stream remappers.

the mesh output data, gathered into a set of registers over the duration of one iteration, to the appropriate positions in a store macrostream vector word, constructing a vector word to pass back to the SE to be committed to memory.

Fig. 4.6 shows a diagram of the load and store stream remapper modules, featuring crossbars to perform the routing, a set of registers to hold the microstream elements for every cycle of the repeating schedule, and auxiliary logic to select the appropriate register to read from/write to based on the `current modulo cycle` signal.

Conceptually, these remappers act like crossbars between the elements of the stream vector registers and the time-extended I/O of the computation mesh. However, because these elements only require permutation and not multicasting (i.e., the same element is routed to only one destination), the remappers use permutation networks instead of full crossbars, significantly reducing the area requirements.

To implement the permutation network, we considered a design based on the Beneš network [84]. Although many permutation network designs have been proposed in the literature, we chose the Beneš permutation network (BPN), because it is generalized for an arbitrary number of inputs and outputs, it is non-blocking, scalable, and is composed of simple 2×2 switching elements arranged as recursively

larger subnetworks, making it well-suited for hardware description with Chisel.

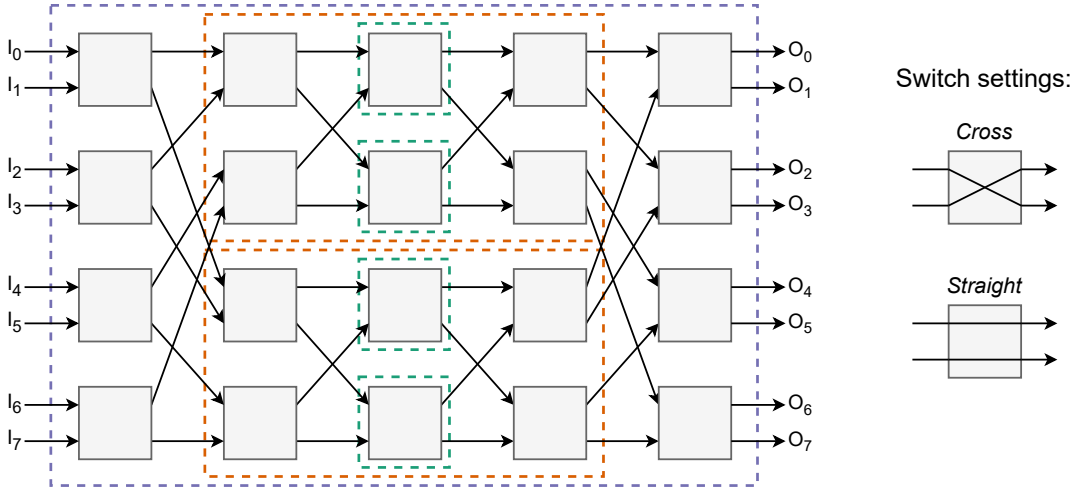


Figure 4.7: Beneš permutation network for 8 signals.

In particular, an n-signal BPN consists of $n\left(\log_2 n - 1/2\right)$ switches arranged in $2\log_2 n - 1$ columns of $n/2$ rows of switches. This configuration yields an asymptotic size of $\mathcal{O}\left(n \log n\right)$ switches, which is more efficient than the quadratic $\mathcal{O}\left(n^2\right)$ scaling of the computation mesh. Consequently, as design size increases, the remappers footprint becomes progressively less significant.

Fig. 4.7 illustrates the internal structure of an 8-signal BPN, highlighting the recursive construction using smaller 4-signal and 2-signal BPNs, with each 2-signal BPN corresponding to an elementary 2×2 switching element.

The setting of the externally controlled switches defines the permutation and can be easily determined by a simple recursive algorithm described in [85].

4.4 Microstreams skew synchronization

Referencing Fig. 3.6 of the computation model, we observe that the computation latency associated to each iteration, often exceeds the II. This implies that certain load or store nodes exhibit initial latencies that surpass the II, meaning they are consumed or produced by the mesh at a later repetition of the repeating schedule than the one when the corresponding iteration started execution. Consequently, from the mesh point of view, load/store microstreams are skewed in their production or consumption.

To better illustrate this, Fig. 4.8 depicts an example where, although each new iteration begins its execution every II, the iteration execution latency spans 5 IIs. In this example, the first load microstream element (L.0) is consumed by the mesh in the first repetition of the schedule (recalling that the schedule repeats each II), while the second element (L.1) is consumed in the second repetition. For the store microstreams, the mesh produces the first element (S.0) in the fifth repetition and the second (S.1) in the fourth repetition of the schedule.

Note, however, that load microstream elements are derived from the same macrostream word at the same point in time, while store microstream elements must be coalesced into a single macrostream

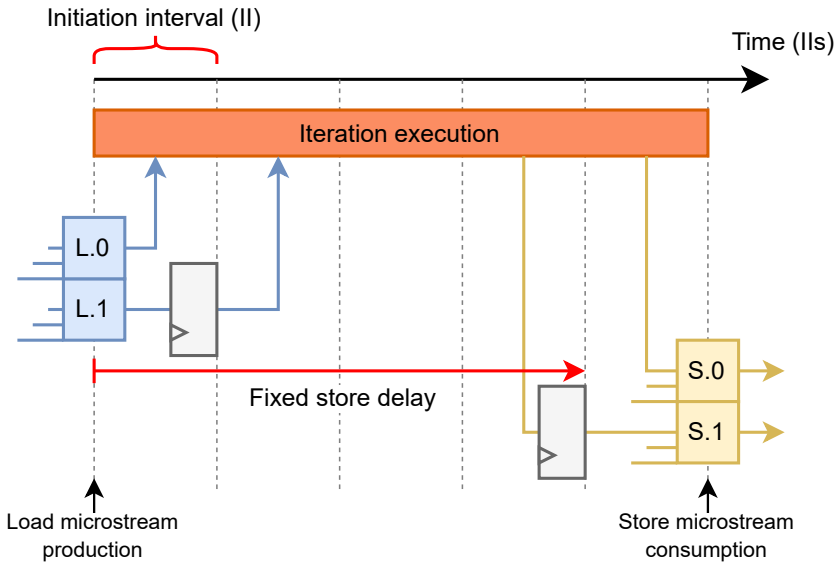


Figure 4.8: Microstreams skew.

word for simultaneous consumption by the streaming infrastructure. This requires the load microstreams to be skewed for sequential consumption by the mesh, while store microstreams are deskewed for synchronized consumption by the streaming infrastructure.

To achieve this skew and deskew synchronization, we introduce an appropriate delay in each microstream, represented by the registers in Fig. 4.8. For load microstreams, the required latency is achieved by directly delaying the signal by the necessary number of IIs. For store microstreams, however, the SE must wait until the element with the longest latency—referred to as the fixed store delay—is produced. Previous elements are delayed by the appropriate number of IIs to align with this longest-latency element, ensuring a simultaneous consumption by the SE.

To apply these targeted delays, skew synchronizers are placed between the outputs of the load stream remapper and the mesh, as well as between the inputs of the store stream remapper and the mesh, as shown in Fig. 4.1. These skew synchronizers add a configurable delay to their input signal.

For load microstreams, the skew synchronizers are configured with the initial latency of each load node, as provided by the mapping tool and measured in IIs units. For store microstreams, each skew synchronizer's configuration specifies the number of IIs to subtract from the fixed store latency.

Fig. 4.9 illustrates the internal implementation of the skew synchronizers. Each synchronizer contains a simple memory with a write pointer that increments every II and a read pointer that trails the write pointer by the specified skew value. If the skew is zero, the output directly matches the input.

4.5 Control

This section outlines the overall system's control, covering the execution control, which manages data movement within the accelerator, and the global control, which handles the configurations and the interface with the host CPU.

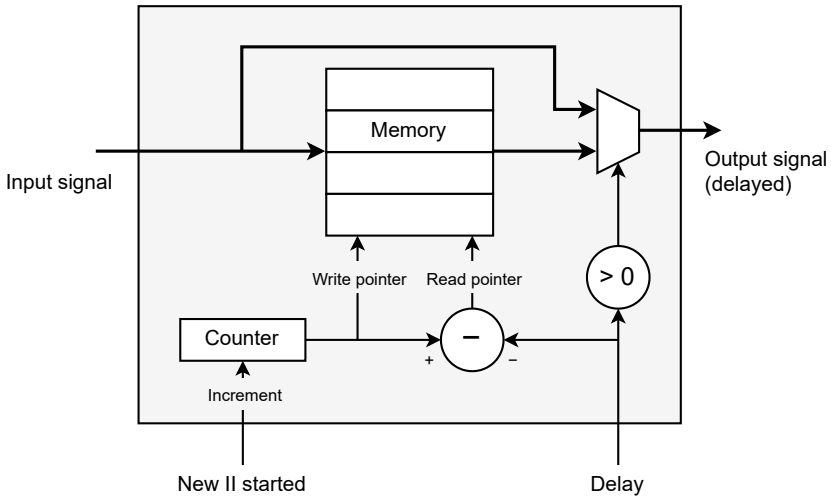


Figure 4.9: Skew synchronizer.

4.5.1 Execution control

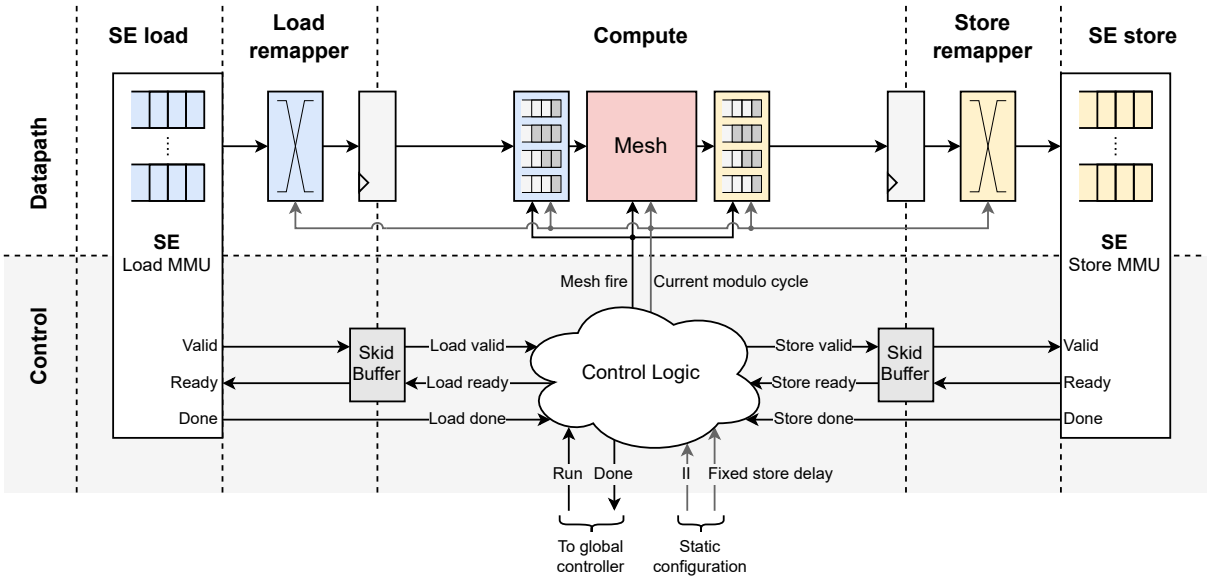


Figure 4.10: Control signals involved in the execution control.

Fig. 4.10 illustrates the control signals used to coordinate the various modules within the datapath.

The implemented execution control mechanism relies on a ready-valid protocol within the SE load and store interfaces to manage the data flow. This protocol accommodates situations where the SE does not have valid output stream registers or is not prepared to receive data in the input stream registers. The protocol operates through two signals: the `ready` signal, which indicates that the consumer is ready to receive data, and the `valid` signal, which indicates that the producer has valid data available for transfer. A data transfer only occurs when both signals are simultaneously asserted.

The SE load and store stream vector data are connected to the remappers, whose registers act as pipeline registers, introducing latency in the data path. To ensure that the `ready` and `valid` signals remain in sync with the vector data, they pass through a skid buffer.

Operating on a static, pre-determined schedule, the mesh executes computations in lockstep—meaning that all PEs must either execute or stall simultaneously. For the mesh to proceed (or ”fire”), both the load stream remapper outputs must be valid, and the store stream remapper inputs must be ready. If either condition is unmet, the entire mesh stalls.

To manage the fixed store delay at the start of the execution, the store pipeline's readiness must be ignored, and the `store valid` signal must be kept `LOW`, as no output data is being produced within that time period. Similarly, once the load streams have concluded, the load pipeline's `valid` signal must be disregarded, allowing the mesh to complete the processing of the final outputs.

As the mesh takes II cycles to consume/produce the entire macrostream vector words, the execution controller must only set its `ready` signal to the load pipeline and `valid` signal to the store pipeline at the last cycle of the II so that data transfers only take place at that time.

Finally, the execution controller receives a high-level `run` signal from the global controller and asserts a `done` signal once all store streams are complete, signaling the end of execution.

4.5.2 Global control and reconfiguration

The global controller is responsible for orchestrating the reconfiguration and the execution of the accelerator, by managing the communication with the host CPU. The accelerator interacts with the host CPU for control and status via MMIO.

A bulk portion of the reconfiguration time is dedicated to transferring the configuration bitstream from the memory system to the accelerator. To offload this task from the host CPU, a DMA is used for the bitstream transfer, allowing the CPU to remain available for other tasks during loading.

The computation model prescribes concurrent bitstream loading and execution, as illustrated in the computation model depicted in Fig. 3.7. To enable concurrent loading and execution, the controller makes use of two distinct data store sets: one for the bitstream being loaded, and a 'hot' set to support the active execution.

The configuration data can be categorized into SE configuration instructions and static configuration (SC). SE configuration instructions represent signal sequences for configuring the SE via the configuration channel, whereas static configuration comprises constant signals used throughout the accelerator to determine the functional roles of its components (e.g., PE operations, crossbar settings). This distinction underscores the distinct treatment of these configuration types.

Fig. 4.11 provides a high-level depiction of the controller's internal structure, including the control signals and the configuration datapath. The configuration-execution process unfolds as follows, with step numbers corresponding to Fig. 4.11's labels:

1. The main memory contains a configuration bitstream for each kernel, preloaded as part of the host CPU's program.

2. The host CPU writes the bitstream's base address of the desired kernel to the ”Bitstream base address” register, and then signals the start of the transfer by writing ”1” to the ”Load bitstream” register.

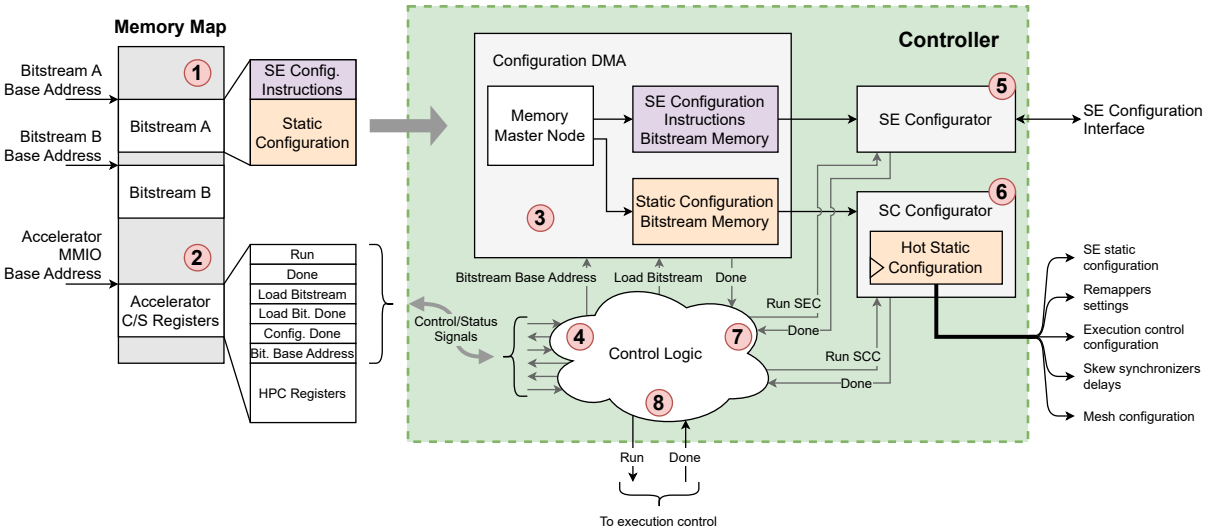


Figure 4.11: Global control and configuration datapath.

3. The configuration DMA controller transfers the bitstream via its memory master interface to designated SE instructions and SC memories, signaling its completion by setting the ”Load bitstream done” status register to ”1”.

4. Upon transfer completion, the host CPU sets the ”Run” register, prompting the accelerator to prepare for execution.

5. The controller activates the SE Configurator, which programs the SE with the required configuration settings.

6. Simultaneously, the SC Configurator loads the static configuration from DMA memory to the static configuration register.

7. When both configurators signal their completion, the controller also flags the completion of its own configuration by setting the ”Configuration done” status register and enables the execution controller by raising the `run` signal.

8. Upon completion of the acceletator’s execution, indicated by the assertion of the `done` signal from the execution controller, the status register is updated to signal the host CPU that the execution has ended.

Fig. 4.12 depicts the concurrency diagram shown previously in Fig. 3.7, annotated with the control and status interactions between the accelerator and the host CPU. Here again, the labels correspond to the steps of the configuration-execution process just described above.

4.6 System-on-chip integration

This section describes the integration of the accelerator within the Rocket Chip SoC template in the Chipyard framework, detailing its implementation and its connections to the memory system.

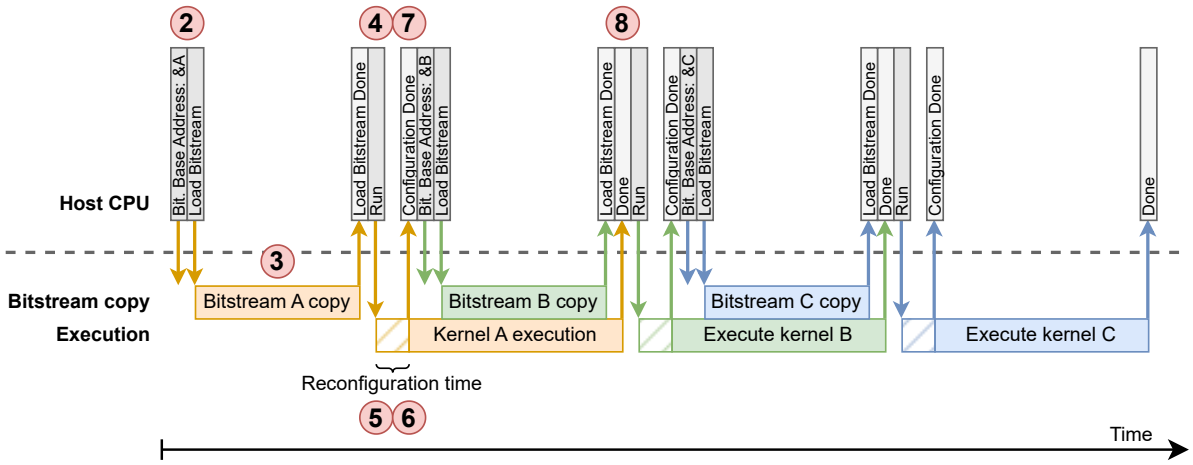


Figure 4.12: Reconfiguration and execution time diagram.

4.6.1 Accelerator–SoC boundary

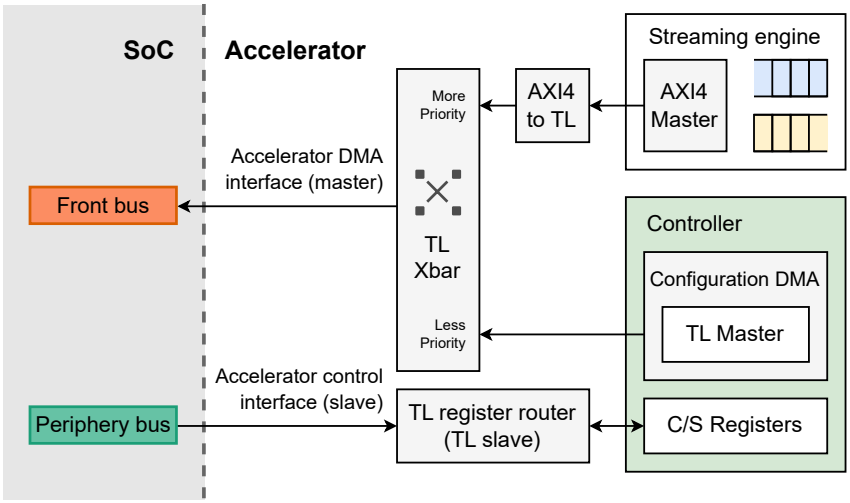


Figure 4.13: Accelerator–SoC boundary.

Fig. 4.13 illustrates the interface between the accelerator and the rest of the SoC. The accelerator implements two bus masters: the SE and the Configuration DMA. Given that the default bus protocol in both the Chipyard framework and Rocket Chip SoC is TileLink (TL) [86], the Configuration DMA of the accelerator is equipped with a TL master interface.

On the other hand, since the SE module implements an AXI4 interface, an AXI4-to-TL widget from the Rocket Chip library is used to bridge the AXI4 protocol to TL. Furthermore, to prioritize the SE data over the Configuration DMA bitstream transfer during concurrent operations, a TL crossbar with an SE data-first arbitration policy is used. This configuration effectively consolidates the accelerator's master memory interfaces into a unified DMA master interface.

The controller's control and status registers are memory-mapped using the TL Register Router, a Rocket Chip component that simplifies the implementation of a MMIO interface. This setup automates much of the interfacing logic, avoiding the need for manually defining a slave TL node and custom logic for register exposure. Accordingly, the accelerator presents an external master TL interface for DMA

operations and a slave TL interface for control.

4.6.2 Memory hierarchy

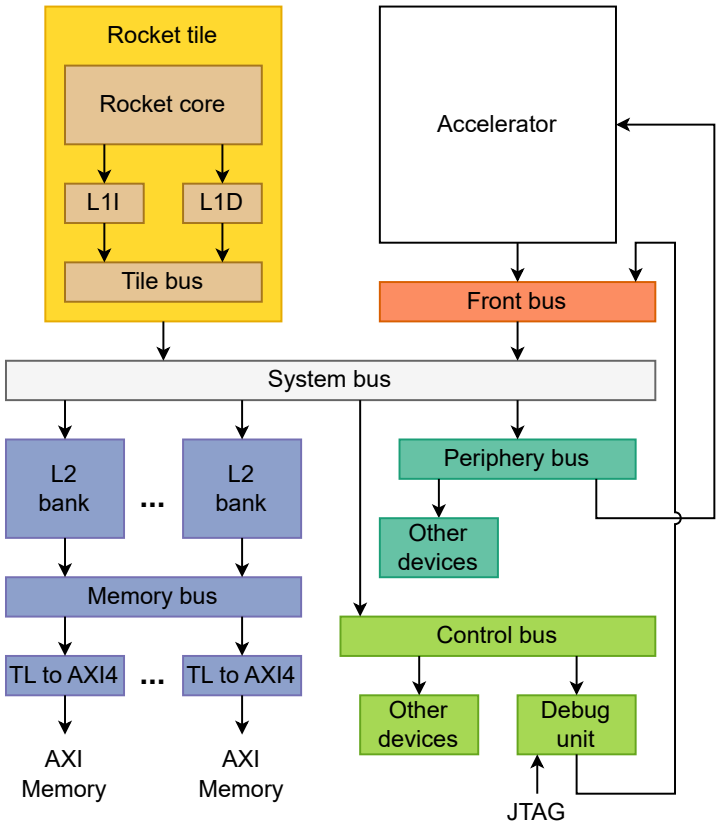


Figure 4.14: SoC memory system.

The conceived SoC follows the default Rocket Chip configuration, which includes a hierarchical, coherent bus topology comprising five main TL buses: the system bus, the periphery bus, the front bus, the control bus, and the memory bus. These buses provide all the commonly used attachment points for the devices, each with tailored characteristics such as operating frequency, word-width, and coherence support suited to their respective device types.

The accelerator's DMA interface connects to the front bus, optimized for DMA devices, and resides above the system bus and L2 cache. Its control interface connects to the periphery bus, the recommended bus for MMIO devices. Fig. 4.14 shows a diagram of the SoC's memory hierarchy.

To meet the accelerator's substantial bandwidth requirements, the SoC configuration includes multiple L2 banks, each paired with a DRAM channel. Positioned alongside the accelerator, the Rocket core interfaces above the system bus and includes private L1 data and instruction caches.

For cache coherency, the L2 caches feature a banked coherence manager, which maintains coherence with the Rocket core's L1 caches through the TileLink Cached (TL-C) protocol, the highest level of TL protocol conformity. However, although the SE within the accelerator acts as an L1 cache during kernel execution, the accelerator does not implement any coherence protocol, as the host core does not access the kernel data during this execution. Instead, it adheres to the TileLink Uncached Lightweight

(TL-UL) level, directly interfacing with the L2 and deferring to the L2 coherence manager to handle all the necessary L1 cache block invalidations.

4.7 Summary

This chapter detailed the hardware microarchitecture that implements the abstract computational model introduced in the previous chapter. The accelerator was described as a parameterizable RTL design written in Chisel and integrated into a Rocket Chip-based SoC using the Chipyard framework.

We presented the accelerator's four main components: the CGRA-style computation mesh, the streaming engine implementing the UVE-based memory access model, the interface coordinating the dataflow between them, and the controller responsible for reconfiguration and execution.

Together, these components form a self-contained accelerator, whose integration into a full SoC was also described.

5

Compilation toolchain

Contents

Referring back to the computational model presented in Fig. 3.8, this chapter details the implementation of the proposed compiler toolchain. This compilation flow encompasses a series of tools and intermediate formats in order to achieve end-to-end automation from a kernel described as annotated C/C++ source code to an executable to be run by the target architecture, as depicted in Fig. 5.1.

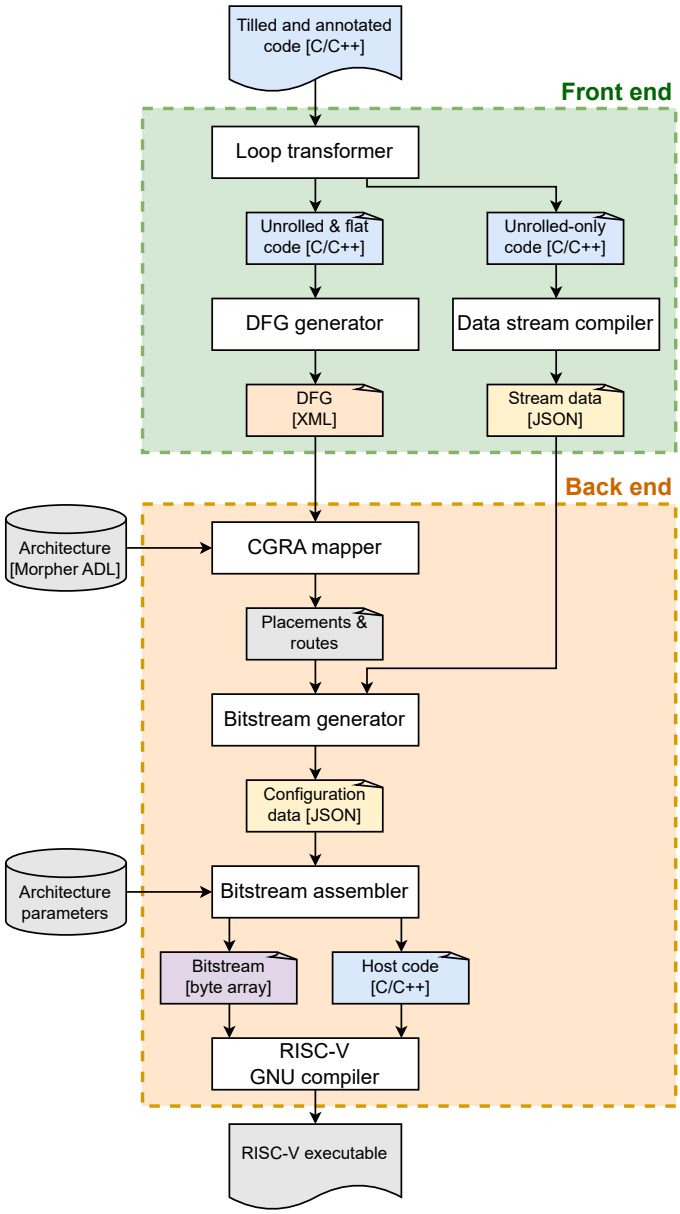


Figure 5.1: Overview of the end-to-end compilation flow.

The several phases that integrate this compilation flow can be grouped into a front end and a back end, each with distinct responsibilities. The front-end phases focus on interpreting and analyzing the original code in an architecture-agnostic perspective. For the proposed system, this includes a pre-processing phase of loop transformations, followed by DFG extraction and memory access pattern extraction. On the other hand, the back-end focuses on code/configuration generation for the specific architecture. This involves the mapping process, configuration generation and assembly, and host code generation.

To automate the procedure depicted in Fig. 5.1, a Python script was developed to seamlessly link each tool, enabling end-to-end compilation with a single command.

5.1 Front end

This section details the front-end steps required to transform the original code into the abstractions of the computational model: the DFG and the data stream descriptors. To support these explanations, the following subsections will use an example based on a 2D Jacobi stencil kernel, whose base source code is presented in Fig. 5.2.

```
void kernel_jacobi_2d() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      B[i][j] =
          0.2f * (A[i][j] + A[i][j - 1] + A[i][j + 1] + A[i - 1][j] + A[i + 1][j]);
    }
  }
}
```

Figure 5.2: Original C source code of a 2D Jacobi stencil.

5.1.1 Code annotation

The input to the compilation flow must be a C/C++ source code annotated with a custom `cgra_map` pragma, which identifies the kernel targeted for acceleration, with explicit tiling applied. This pragma is required to be placed within a perfect loop nest. Then, following the considered computation model, loops at higher levels will be linearized, while loops nested below the pragma represent the tile that will be unrolled to form the DFG. Fig. 5.3 illustrates the original kernel's source code, annotated with the pragma and organized using a 2D tiling strategy. In this structure, the inner loops produce a 2D tile, while the outer loops increment by the tile size, corresponding to the number of tiles to be executed.

Currently, explicit tiling is required from the programmer. Nevertheless, with the potential future implementation of an automatic tiling pass, the pragma could also accept tiling parameters (e.g., dimensions and tiling factors) for the compiler to automatically apply tiling transformations.

5.1.2 Loop transformer

The loop transformer step performs three main transformations: loop unrolling, loop flattening, and some instrumentation to prepare the code for the next tool. Implemented using Clava, this source-to-source transformation tool (described in Section 2.4.3), provides some extra flexibility by enabling direct manipulation of the source code, bypassing the complexity of developing custom LLVM passes.

The considered Clava transformations are scripted in JavaScript. In the loop unrolling step, the transformer identifies the custom pragma and locates an inner loop at the same AST level as the pragma.

```c
#define N 1024
#define TILE_SIZE 2

void kernel_jacobi_2d() {
  for (int i = 0; i < N; i += TILE_SIZE) {
    for (int j = 0; j < N; j += TILE_SIZE) {
      #pragma cgra_map
      for (int ii = 0; ii < TILE_SIZE; ii++) {
        for (int jj = 0; jj < TILE_SIZE; jj++) {
          B[i + ii][j + jj] =
              0.2f * (A[i + ii][j + jj] + A[i + ii][j + jj - 1] +
                      A[i + ii][j + jj + 1] + A[i + ii - 1][j + jj] +
                      A[i + ii + 1][j + jj]);
        }
      }
    }
  }
}
```

Figure 5.3: Tiled and annotated C source code.

Then, the unrolling process extracts the loop's induction variable, initialization, end-condition, and step-value. It then generates all the unrolled loop bodies by substituting the induction variable with actual values, determined through the execution of a simulated JavaScript loop configured with the same parameters as the original C loop. This transformation is applied recursively to effectively handle nested inner loops. Fig. 5.4 shows an example of the output from the loop unrolling transformation of the code in Fig. 5.3.

In the loop flattening step, the transformer makes use of a straightforward approach: it identifies loops external to the pragma and replaces them with a single loop whose iteration count equals the product of the original inner loop sizes. To maintain the functional equivalency of the unrolled body, the flattener also declares the original induction variables and defines each with an expression that computes its value based on the new linearized induction variable `idx`. This strategy ensures that the transformed code retains the intended behavior while optimizing the loop structure for the DFG generator.

Finally, an additional instrumentation step replaces the pragma by a call to the `please_map_me()` function for compatibility with the next tool in the flow. Fig. 5.5 depicts the code after the unrolling, flattening, and instrumentation transformations.

5.1.3 DFG generator

To generate the DFGs from the transformed C source code, the implemented flow makes use of the DFG generator tool available in the Morpher Framework [70]. Compiled with the experimental `REMOVE_AGI` define, this configuration of this tool removes all the address generation instructions (AGIs) and control flow elements, resulting in a computation-only DFG, as required by the proposed computational model.

The DFG generator starts by invoking *Clang* to produce the LLVM IR. Then, it runs the *opt* (the LLVM optimizer), which applies a custom LLVM pass for DFG extraction. The DFG is directly extracted

```
void kernel_jacobi_2d() {
  for (int i = 0; i < 1024; i += 2) {
    for (int j = 0; j < 1024; j += 2) {
      #pragma cgra_map
      {
        {
          B[i + 0][j + 0] = 0.2f * (A[i + 0][j + 0] + A[i + 0][j + 0 - 1] +
                                    A[i + 0][j + 0 + 1] + A[i + 0 - 1][j + 0] +
                                    A[i + 0 + 1][j + 0]);
        }
        {
          B[i + 0][j + 1] = 0.2f * (A[i + 0][j + 1] + A[i + 0][j + 1 - 1] +
                                    A[i + 0][j + 1 + 1] + A[i + 0 - 1][j + 1] +
                                    A[i + 0 + 1][j + 1]);
        }
      }
      {
        {
          B[i + 1][j + 0] = 0.2f * (A[i + 1][j + 0] + A[i + 1][j + 0 - 1] +
                                    A[i + 1][j + 0 + 1] + A[i + 1 - 1][j + 0] +
                                    A[i + 1 + 1][j + 0]);
        }
        {
          B[i + 1][j + 1] = 0.2f * (A[i + 1][j + 1] + A[i + 1][j + 1 - 1] +
                                    A[i + 1][j + 1 + 1] + A[i + 1 - 1][j + 1] +
                                    A[i + 1 + 1][j + 1]);
        }
      }
    }
  }
}
```

Figure 5.4: Unrolled code.

from the IR, leveraging its static single assignment (SSA) form, which inherently represents a DFG. Post-extraction, control flow and AGIs are removed, isolating only the computation operations.

The preceding loop-flattening transformation ensures that memory accesses dependent on outer loop induction variables are treated as regular memory accesses rather than loop-carried dependencies. This adjustment is crucial, as the CGRA datapath is not responsible for any loop control; from the CGRA datapath's perspective, all outer loops are effectively linearized, with loop control implicitly managed by the SE.

The output of this step is a quasi-XML file format containing the DFG and additional information, including constants and inter-iteration recurrence relationships. Additionally, a DOT graph file is generated, which can be rendered as an image, as shown in Fig. 5.6. In this graph, the four `STORE` nodes correspond to the four store operations in the code from Fig. 5.5, which represent four store microstreams. Fig. 5.6a depicts a snippet of the XML output file, which represents two DFG nodes and their respective inputs and outputs.

```
void kernel_jacobi_2d() {
  for (int idx = 0; idx < 262144; idx++) {
    int j = (idx) % 512 * 2;
    int i = (idx / (512)) % 512 * 2;
    please_map_me();
    {
      {
        B[i + 0][j + 0] = 0.2f * (A[i + 0][j + 0] + A[i + 0][j + 0 - 1] +
                                  A[i + 0][j + 0 + 1] + A[i + 0 - 1][j + 0] +
                                  A[i + 0 + 1][j + 0]);
      }
      {
        B[i + 0][j + 1] = 0.2f * (A[i + 0][j + 1] + A[i + 0][j + 1 - 1] +
                                  A[i + 0][j + 1 + 1] + A[i + 0 - 1][j + 1] +
                                  A[i + 0 + 1][j + 1]);
      }
    }
    {
      {
        B[i + 1][j + 0] = 0.2f * (A[i + 1][j + 0] + A[i + 1][j + 0 - 1] +
                                  A[i + 1][j + 0 + 1] + A[i + 1 - 1][j + 0] +
                                  A[i + 1 + 1][j + 0]);
      }
      {
        B[i + 1][j + 1] = 0.2f * (A[i + 1][j + 1] + A[i + 1][j + 1 - 1] +
                                  A[i + 1][j + 1 + 1] + A[i + 1 - 1][j + 1] +
                                  A[i + 1 + 1][j + 1]);
      }
    }
  }
}
```

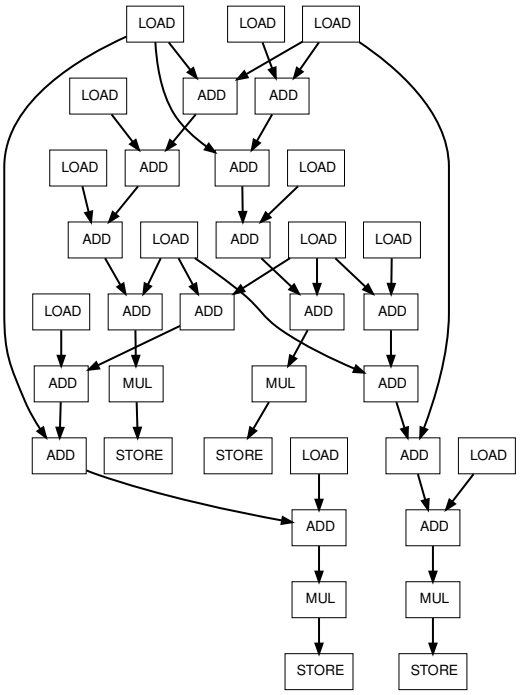
Figure 5.5: Unrolled and flattened code.

5.1.4 Data streaming compiler

The data streaming compiler was designed to automatically extract and encode UVE memory access pattern descriptors directly from C source code, following the approach detailed in [68].

In the proposed flow, this compiler operates on the unrolled-only code, as the flattening pass removes the outer loops that encode the memory access patterns. At its current stage, the compiler, still under active development at HPCAS, generates descriptors for each individual microstream from the unrolled code (i.e. for each load/store operation in the code in Fig. 5.5), resulting in an excessive number of microstreams. To follow the considered computation model, the compiler should consolidate these individual microstreams into a single vector macrostream, establishing a clear correspondence between microstreams and the encompassing macrostream. This shortcoming also impacts other applications of the tool in UVE-based GPP applications, especially for stencil computations, where each stencil point would become its own data stream.

Future improvements of this tool aim to address this limitation by implementing automated coalescing, allowing the compiler to directly generate optimized macrostream descriptors. Once resolved, the tool will be able to autonomously produce both the coalesced macrostream and the corresponding rela-



(a) Simple DFG representation as DOT graph.

```
...
<Node idx="23" ASAP="4" ALAP="12" BB="for.body">
<OP>ADD</OP>
<Inputs>
        <Input idx="22"/>
        <Input idx="29"/>
</Inputs>
<Outputs>
        <Output idx="24" nextiter="0"
        ↪  type="I1"/>
</Outputs>
<RecParents>
</RecParents>
</Node>

<Node idx="45" ASAP="5" ALAP="13" BB="for.body"
↪  CONST="1045220557">
<OP>MUL</OP>
<Inputs>
        <Input idx="44"/>
</Inputs>
<Outputs>
        <Output idx="15" nextiter="0"
        ↪  type="I1"/>
</Outputs>
<RecParents>
</RecParents>
</Node>
...
```

(b) Snippet of the DFG XML file.

Figure 5.6: Sample DFG XML output with a corresponding graph representation.

tionships with individual microstreams.

In the interim, the programmer is required to manually provide coalesced macrostream descriptors and define their correspondence with microstreams within a JSON file. An example of this stream description format is illustrated in Fig. 5.7. In this format, the `macro_description` key encapsulates a sequence of descriptors that define the coalesced macrostream, while the `micro_pattern` key specifies, for each macrostream vector word, the corresponding microstream nodes within the DFG for each element. This JSON file is subsequently used as input by downstream tools within the compilation flow.

5.2 Back end

This section focuses on the back-end steps that translate the computational model abstractions generated by the front end into the architecture-specific configurations, encompassing mapping, configuration generation, and bitstream assembly.

5.2.1 CGRA mapper

The compilation flow makes use of the CGRA mapper tool provided by the Morpher framework [70] to map the DFG onto the target architecture, represented as an abstract architecture model described in Morpher's custom ADL. The Morpher ADL models CGRA architectures using modules, ports, and

```
...
"stores": [
  {
    "macro_description": [
      {"offset": "0x82000000", "size": 2, "stride": 1},
      {"offset": "0", "size": 2, "stride": 1024},
      {"vectorize": true},
      {"offset": "0", "size": 512, "stride": 4},
      {"offset": "0", "size": 512, "stride": 2048}
    ],
    "micro_pattern": [26, 18, 15, 7]
  }
]
...
```

Figure 5.7: Snippet of the JSON file containing macrostream descriptions and microstream correspondence patterns.

connections. Modules, representing elements like FUs and RFs, are configurable and may contain submodules. Ports manage the data flow, while connections define module links. FU modules can model either ALUs or load/store units (LSUs) based on a list of supported operations and operation latency, while RF modules model register files. Multiplexers are inferred through connections, and a specialized syntax simplifies the PE interconnections. Fig. 5.8 depicts an example of how a two-input two-output PE can be modeled with Morpher ADL.

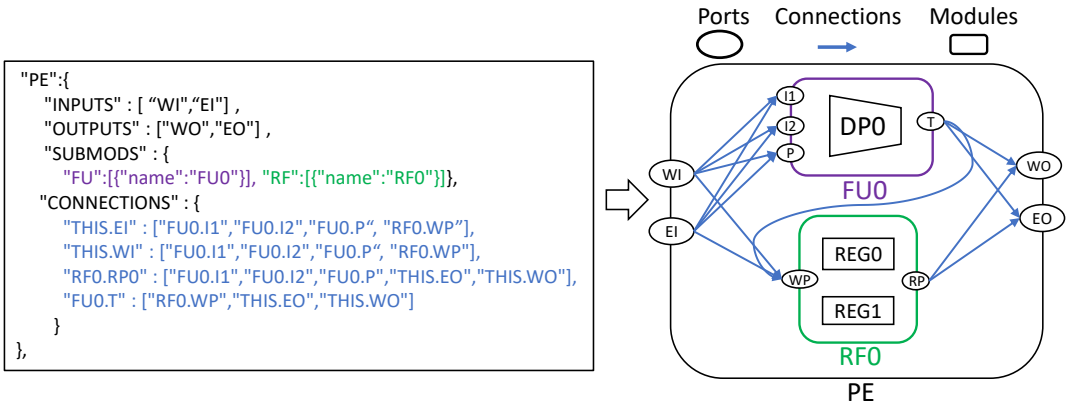


Figure 5.8: Example of the Morpher ADL to descirbe a simple two-input two-output PE [70].

The remaining modeling of the proposed architecture follows a straightforward approach, defining a grid of PEs with characteristics similar to those of the target architecture. To represent the I/O ports of the mesh for load and store microstreams, phony PEs that can only perform load and store operations are introduced. This setup takes advantage of the CGRA mapper to assign load and store microstreams to the designated mesh ports.

The adopted mapping procedure makes use of a heuristic-based pathfinder algorithm based on [55], which, when successful, generates a file detailing the placements of DFG nodes onto spatio-temporal coordinates, and the routes between nodes mapped to architectural locations such as PEs, registers and ports. A snippet of this file is shown in Fig. 5.9. Additionally, a separate file is produced that lists the

```
...
node=22,mapped=PE_X4|_Y1|-T0
        parent:21
                CGRA_Ins.PE_X4|_Y2|-T1.FU0.DP0_T
                CGRA_Ins.PE_X4|_Y2|-T1.NR_R0
                CGRA_Ins.PE_X4|_Y2|-T0.NR_RI
                CGRA_Ins.PE_X4|_Y2|-T0.NORTH_O
                CGRA_Ins.PE_X4|_Y1|-T0.SOUTH_I
                CGRA_Ins.PE_X4|_Y1|-T0.FU0.DP0_I2
                CGRA_Ins.PE_X4|_Y1|-T0.FU0.DP0.I2
        parent:53
                CGRA_Ins.PE_SE_NORTH_X2|_Y0|-T1.LU0.DP0_T
                CGRA_Ins.PE_SE_NORTH_X2|_Y0|-T1.SOUTH_O
                CGRA_Ins.PE_X2|_Y1|-T1.NORTH_I
                CGRA_Ins.PE_X2|_Y1|-T1.RF0.WP0
                CGRA_Ins.PE_X2|_Y1|-T1.RF0.R1_R0
                CGRA_Ins.PE_X2|_Y1|-T0.RF0.R1_RI
                CGRA_Ins.PE_X2|_Y1|-T0.RF0.RP1
                CGRA_Ins.PE_X2|_Y1|-T0.ER_R0
                CGRA_Ins.PE_X2|_Y1|-T1.ER_RI
                CGRA_Ins.PE_X2|_Y1|-T1.EAST_O
                CGRA_Ins.PE_X3|_Y1|-T1.WEST_I
                CGRA_Ins.PE_X3|_Y1|-T1.ER_R0
                CGRA_Ins.PE_X3|_Y1|-T0.ER_RI
                CGRA_Ins.PE_X3|_Y1|-T0.EAST_O
                CGRA_Ins.PE_X4|_Y1|-T0.WEST_I
                CGRA_Ins.PE_X4|_Y1|-T0.FU0.DP0_I1
                CGRA_Ins.PE_X4|_Y1|-T0.FU0.DP0.I1
...
```

Figure 5.9: Snippet of the CGRA mapper output containing the placements and routes.

latency of each node, which is later used to calculate the microstream skews.

5.2.2 Configuration generator

The configuration generator, implemented in Scala, processes the Morpher route file, analyzing each connection in the route and setting the appropriate configuration bits across the accelerator. This procedure configures various components, including registers and multiplexers, ensuring the correct data flow paths and functionality. In addition to establishing connections, the configuration generator sets the FU operations, the sets constants, and uses the latency information to configure the skew synchronizers. It also extracts the descriptors from the streams JSON file, translating them into configuration instructions to program the SE. Additionally, it processes the macrostream-microstream correspondence, generating the switch settings for the BPNs within the remappers.

The output is a human-readable JSON file containing all configuration values for the accelerator's repeating schedule, dictating the exact behavior of each component within the accelerator. Fig. 5.10a presents a snippet of this file, highlighting the configuration of a specific PE during a particular cycle of the repeating schedule.

```
...
{
  "op": "mul",
  "outRegsSel": {
    "north": "result",
    "south": "result",
    "west": "north",
    "east": "result"
  },
  "outRegsEn": {
    "north": false,
    "south": true,
    "west": true,
    "east": false
  },
  "rfWritePortsSel": {
    "ports": ["west", "result"]
  },
  "fuSrcSel": {
    "a": "rfPort0",
    "b": "immediate"
  },
  "rfReadAddr": [3, 0],
  "rfWriteRegSourceSel": [0, 0, 0, 0],
  "rfWriteEn": [false, true, false, false],
  "immediate": "1045220557"
}
...
```

(a) Snippet of the JSON configuration file.

```
unsigned char jacobi_2d_bitstream[] __attribute__
↪  ((aligned(64))) = {
  0x02, 0x00, 0x00, 0x00, 0x32, 0x00, 0x00, 0x00,
  0x40, 0x08, 0x00, 0x08, 0x54, 0x00, 0x00, 0x00,
  0x12, 0x01, 0x04, 0x14, 0x04, 0x12, 0x10, 0x14,
  0x32, 0x16, 0x10, 0x31, 0x10, 0x36, 0x70, 0x35,
  0x10, 0x3b, 0x0a, 0x1b, 0x39, 0x1a, 0x23, 0x00,
  0xc5, 0x00, 0xc5, 0x00, 0x2a, 0x04, 0x95, 0x06,
  0x88, 0xf8, 0xc7, 0x00, 0x6c, 0x26, 0x4e, 0x00,
  0xf6, 0x3c, 0x78, 0x68, 0x20, 0x48, 0x00, 0x00,
  ...
  ...
};
```

(b) Header file with the bitstream as a C static byte array.

Figure 5.10: Configuration representations.

5.2.3 Bitstream assembler

The configuration JSON file acts as a human-readable representation of the accelerator's configuration, much like assembly code serves as a readable intermediary before machine code. Each entry in the configuration JSON file has a 1:1 correspondence with a specific setting in the bitstream, creating an assembly-to-machine-code-like relationship. The bitstream assembler tool's role is to convert this JSON configuration into a bitstream that can be directly loaded into the accelerator's configuration memories.

At the top level of the Chisel architecture description, the configuration signals are grouped within a Chisel bundle. The assembler, implemented in Scala, constructs a Chisel bundle literal of the same type as this configuration bundle and then takes advantage of ChiselSim to perform a reinterpretation cast of it as a Chisel `UInt` literal, effectively converting it to a raw array of bytes.

To optimize the bitstream loading process, padding is added to the bundle to ensure that the resulting bitstream size aligns with the cache line size, assuring that the bitstream loading process happens in cache-line-sized blocks.

At the end, the tool outputs the bitstream as a C static array definition, allowing it to be easily included in source code and embedded by the compiler in the final executable without requiring custom linker scripts or assembly. Fig. 5.10b shows an example of the assembled configuration bitstream.

Additionally, a minimal example of host code is generated to demonstrate the execution of the kernel on the accelerator. Fig. 5.11 provides this host code, with appropriate comments detailing the control interactions with the accelerator, directly mapping to the process outlined in Fig. 4.12.

Finally, the host generated code is cross-compiled using the RISC-V GNU compiler.

```c
/* Write the bitstream base address */
reg_write64(BITSTREAM_BASE_ADDR, (uint64_t) &jacobi_2d_bitstream);
/* Start the bitstream loading */
reg_write8(LOAD_BITSTREAM, 1);
/* Bitstream is being loaded */
while (reg_read8(LOAD_BITSTREAM_DONE) == 0) ;
/* Bitstream is loaded */

/* Start the accelerator */
reg_write8(RUN, 1);
/* The accelerator is configuring itself with the bitstream */
while (reg_read8(CONFIGURATION_DONE) == 0) ;
/* The accelerator is configured and will run */

/* The accelerator is running */
/* A new bitstream could be loaded now, while the accelerator is running */
while (reg_read8(DONE) == 0) ;
/* The accelerator has finished running */
```

Figure 5.11: Minimal host code.

5.3 Summary

This chapter presented the full compilation flow that targets the proposed architecture. The toolchain transforms an annotated C/C++ kernel into a RISC-V executable, generating all necessary configuration artifacts along the way.

The front end performs loop transformations, extracts the computation as a DFG, and derives memory access patterns for the UVE streaming model. The back end handles mapping onto the CGRA, configuration generation, and bitstream assembly. A minimal host program is also produced and compiled, enabling direct execution on the target system.

6

Experimental evaluation

Contents

This chapter evaluates the proposed computational system in terms of performance, hardware resources utilization, and energy efficiency. The methodology includes comparisons against both open-source IP and a commercial-off-the-shelf (COTS) system to establish a robust baseline.

To ensure a fair comparison, the considered open-source references are synthesizable under similar conditions, including an equivalent memory hierarchy and VLSI fabrication process. The chosen references are:

- **Rocket**: a scalar GPP.
- **Saturn**: a vector unit, attached to a Rocket core that implements the RISC-V Vector Extension (RVV).
- **Gemmini**: a fixed-dataflow DSA for deep-learning.

For real-world validation, a COTS system has also been included: an **Arm** Cortex-A53 CPU with the Neon SIMD extension.

The proposed architecture was implemented as a parametric RTL template. In particular, concrete parameters used for the evaluation were selected to balance practical implementation constraints and benchmark requirements. The most relevant key parameters are listed in Tab. 6.1.

Table 6.1: Key architecture parameters.

	Parameter	**Value**
Accelerator	Data type	FP32
	Mesh dimensions	4×4
	Max. II	8/16
	RF size	4
	Max. skew	2
	SE LRQ tables	32
	SE LRQ requests	16
	SE LLB tables	4
SoC	System bus width	256 bit
	Cache block size	64 B
	L2 banks	1
	L2 ways	8
	L2 capacity	512 KiB
	DRAM channels	1

The accelerator's 4×4 mesh configuration was considered the largest array size for which valid mappings could be generated by the mapping tool within a reasonable timeframe (~24 hours). Other accelerator parameters were selected based on resource utilization estimates derived from benchmark requirements. The SE parameters were configured according to the authors' recommendations for the original implementation. For the SoC, the considered parameter values were adjusted to accommodate the accelerator's high memory bandwidth demands, namely, the width of the system bus and L2.

Two configurations were used during the considered evaluation: one allowing a maximum II of 8 cycles and another allowing a maximum II of 16 cycles. While most benchmarks fit within the II-8 configuration, two required the II-16 configuration due to mapping constraints. Importantly, the larger II did not affect performance but required additional area. Future work refinements to be discussed in section 6.4.3 would enable the system to handle larger IIs (such as 16) without requiring additional area compared to the current II-8 configuration. As such, the II-8 configuration will be emphasized throughout this chapter, as it aligns more closely with the expected area efficiency of the refined implementation.

This chapter begins by evaluating the attained performance based on the simulation results. It then analyzes area utilization and energy efficiency using VLSI synthesis outcomes. The obtained findings reveal some limitations in the current implementation, prompting some envisaged improvements and an estimated evaluation of a system incorporating these enhancements.

6.1 Benchmarks and mapping

The implemented architecture was evaluated using a diverse set of benchmarks drawn from various application domains. These benchmarks are as follows:

- **Jacobi 1D**: A 3-point stencil computation from the PolyBench benchmark suite [87], representing a linear algebra workload.
- **Jacobi 2D**: A 5-point stencil computation from PolyBench, also focused on linear algebra.
- **Heat 3D**: A 7-point stencil computation in a three-dimensional space, derived from computational physics applications and included in PolyBench.
- **Black-Scholes**: Option pricing in quantitative finance models from PARSEC [88] benchmark suite, conveniently adapted to replace the division operation with the iterative Newton-Raphson method.
- **FIR**: A finite impulse response filter benchmark sourced from CGRA-Bench [89], highly applied in the signal processing domain.
- **GEMM**: A general matrix-matrix multiplication workload, representing linear algebra computations and obtained from PolyBench.

Table 6.2 provides a detailed characterization of each benchmark, including their computational and memory access patterns. This table also summarizes the mapping characteristics obtained using the CGRA mapping tool, alongside the resulting computational parameters derived from the mapped benchmarks.

The table employs various metrics used to characterize the benchmarks, with those that are non-obvious and non-self-explanatory described here:

- **Tile size**: Defines the tiling factor applied to each dimension. A linear tile size indicates the tiling factor for one single dimension, while the dimensional tile size represents the product of the tiling

factors across all dimensions. For instance, in a 2D Jacobi kernel with a linear tile size of 2, the dimensional tile size is $2^2 = 4$. Refer to Fig. 5.3, which illustrates the tiling process for the Jacobi 2D kernel.

- **Quality of mapping (QoM)**: Measures how closely the mapping solution approximates the optimal mapping (as calculated by the mapper tool), calculated as:

$$\text{QoM} = \frac{\text{Minimum II}}{\text{Mapped II}}$$

- **Array utilization**: Indicates the percentage of utilized PEs, calculated as:

$$\text{Array utilization} = \frac{\text{DFG operations}}{\text{Number of PEs} \times \text{Mapped II}}$$

- **Arithmetic intensity**: Quantifies the ratio of computational operations to memory bandwidth requirements, expressed as:

$$\text{Arithmetic intensity} = \frac{\text{DFG operations}}{\text{DFG memory positions} \times \text{Element size}} \quad [\text{op/B}]$$

- **Arithmetic rate**: Represents the rate of performed operations per cycle, calculated as:

$$\text{Arithmetic rate} = \frac{\text{DFG operations}}{\text{Mapped II}} \quad [\text{op/cycle}]$$

- **Load rate and Store rate**: Represents the required memory bandwidth for data loads and stores, calculated as:

$$\text{Load/Store rate} = \frac{\text{DFG loads/stores} \times \text{Element size}}{\text{Mapped II}} \quad [\text{B/cycle}]$$

- **Total memory rate**: Represents the total memory bandwidth, calculated as:

$$\text{Total memory rate} = \text{Load rate} + \text{Store rate} \quad [\text{B/cycle}]$$

The used tile sizes were chosen as the minimum values that provided a reasonable array utilization. Although for some specific benchmarks, such as FIR and GEMM, larger tile sizes could have improved the array utilization, the mapping tool already encountered significant difficulty in achieving acceptable QoM for these sizes within a reasonable run time. As a result, particularly in the GEMM benchmark, the performance is limited primarily by the quality of the mapping rather than the inherent characteristics of the benchmark.

Table 6.2: Benchmark characterization.

	Benchmark	Jacobi 1D	Jacobi 2D	Heat 3D	Black-Scholes	FIR	GEMM
Dim.	Dimensions	1	2	3	1	1	3
	Tile size (linear)	30	4	2	1	3	4
	Tile size (dimensional)	30	16	8	1	3	64
DFG	Operations	75	80	104	103	117	128
	Loads	32	32	32	5	22	48
	Stores	30	16	8	1	3	16
	Memory total	62	48	40	6	25	64
Mapping	Minimum II	5	5	7	7	8	8
	Mapped II	5	6	8	7	11	16
	Tool run time (h)	1.1	3.0	3.6	17.2	36.3	11.1
	QoM	100%	83%	88%	100%	73%	50%
	Array utilization	94%	83%	81%	92%	66%	50%
Computation	Arithmetic intensity (op/B)	0.30	0.42	0.65	4.29	1.17	0.50
	Arithmetic rate (op/cycle)	15.00	13.33	13.00	14.71	10.64	8.00
	Load rate (B/cycle)	25.60	21.33	16.00	2.86	8.00	12.00
	Store rate (B/cycle)	24.00	10.67	4.00	0.57	1.09	4.00
	Total memory rate (B/cycle)	49.60	32.00	20.00	3.43	9.09	16.00

6.1.1 Discussion

To conclude, it is worth noting that current toolchain relies on simplistic mapping strategies constrained by abstractions that are too closely tied to the imperative code style and instruction models of traditional GPPs. This reliance particularly limits the abstraction capabilities within our computational model and limits the maximum efficiency and performance attainable by the system. For instance, supporting certain operations (such as accumulations), which are trivial from a microarchitectural perspective, is constrained by the compiler's ability to represent and map these operations onto the architecture. Instead, accumulations were implemented as load-store recurrences, resulting in excessive memory accesses, larger DFGs, and unnecessarily high IIs. Subsequently, the presence of recurrences forced the minimum II to accommodate inter-iteration dependencies. To compensate for this and to achieve a reasonable array utilization, larger unrolling factors were employed, leading to significantly larger DFGs. These larger DFGs are more challenging to map and contribute to unnecessarily increased IIs.

6.2 Performance

This section presents a performance evaluation of the proposed accelerator, examining its execution metrics and comparing them with other systems.

6.2.1 Simulation and execution

The evaluation of the proposed architecture relied on executing the set of benchmarks described in section 6.1 by using both FPGA-accelerated RTL simulations and a real-world COTS system.

For the synthesizable IP architectures, including the proposed accelerator, FPGA-accelerated simulations were conducted using FireSim [38], targeting a clock frequency of 1 GHz. FireSim provides the capability of running significantly larger simulations than conventional RTL simulations, enabling the evaluation of datasets that exceed cache sizes and utilize realistic DRAM models. In this evaluation, DDR3 memory models running at 1 GHz were used, resulting in transfer rates similar to the DDR3-2133 specification, which achieves comparable rates at a 1066 MHz frequency. The dataset sizes for the benchmarks were chosen to be approximately 16 MiB, ensuring a balance between representing meaningful workloads and maintaining the feasibility of the RTL simulations, even if FPGA-accelerated. For the COTS systems, the Arm core integrated into a Xilinx development board operated at 1.2 GHz.

The benchmarks' code was adapted to the specificities of each architecture to ensure an accurate evaluation. For architectures supporting vector instructions, such as the Saturn vector unit and the ARM Cortex-A53 system, the workloads relied on the compiler's automatic vectorization. Notably, Gemmini, as a fixed-dataflow DSA, was restricted to the GEMM benchmark due to its specialization in matrix-matrix multiplication.

Hardware performance counters were implemented in the proposed accelerator to collect key execution metrics. For the other architectures, cycle and time counters were used to measure the execution time, ensuring consistent metric collection for comparison.

6.2.2 Results and discussion

The performance results obtained from the execution of the benchmarks on the proposed architecture, as simulated using FireSim, are presented in Table 6.3. This table includes metrics collected through hardware performance counters, offering detailed insights into the execution performance. Fire cycles are the cycles during which computation is executed without stalls. The fire ratio represents the proportion of fire cycles to the total number of cycles.

An analysis of the obtained counts of load and store stalls reveals that, despite the implementation of a robust streaming infrastructure, memory accesses remain a limiting factor for most benchmarks. Nonetheless, the streaming infrastructure effectively coalesces memory accesses, reducing the total number of requests. This is evident when comparing the data size of the benchmarks with the number of memory requests observed during execution.

Figure 6.1 provides further insight into this aspect by plotting benchmark metrics on a roofline model, which relates arithmetic performance to arithmetic intensity. The upper bound, representing the computational limit, indicates the peak performance achievable by the architecture's execution resources (computed as the number of PEs multiplied by the operating frequency). The memory bound, relevant at lower arithmetic intensities, is shown as a line with a slope equal to the peak memory bandwidth, calculated as the L2 bus width times the frequency. This assumes ideal memory behavior and does not

Table 6.3: Accelerator performance results from FPGA-accelerated RTL simulation.

Benchmark	Jacobi 1D	Jacobi 2D	Heat 3D	Black-Scholes	FIR	GEMM
Data size	4.19 M	2048^2	160^3	4.19 M	4.19 M	2048^2
Total cycles	25.2 M	30.5 M	57.5 M	70.7 M	51.8 M	40.2 G
Fire cycles	699 k	1.57 M	4.10 M	29.4 M	15.4 M	2.15 G
Load stalls	28.3 k	2.08 M	42.0 M	32.6 M	27.9 M	38.0 G
Store stalls	24.4 M	26.9 M	11.4 M	8.73 M	8.56 M	13.7 M
Load requests	279 k	1.13 M	3.23 M	1.31 M	1.30 M	865 M
Store requests	262 k	786 k	1.28 M	262 k	262 k	268 M
Fire ratio	2.78 %	5.15 %	7.13 %	41.5 %	29.7 %	5.35 %
Real array utilization	2.61 %	4.29 %	5.79 %	38.2 %	19.7 %	2.67 %
Execution time (s)	0.0252	0.0305	0.0575	0.0707	0.0518	40.2

account for inefficiencies such as memory access latencies or cache misses.

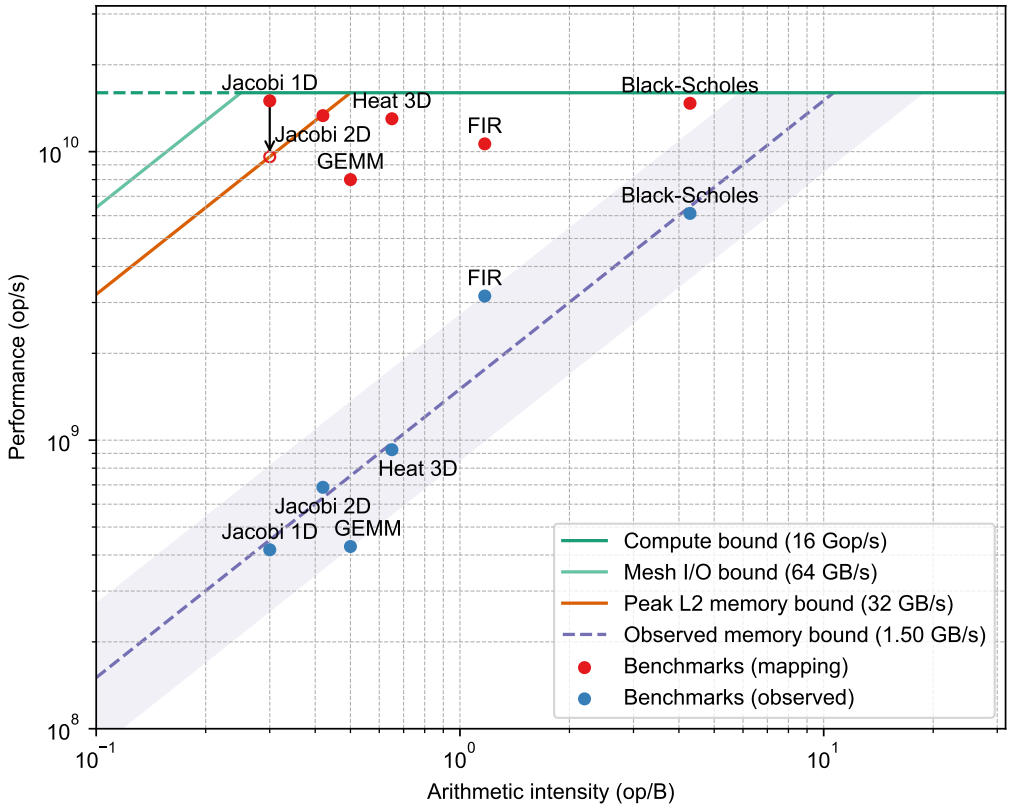


Figure 6.1: Considered benchmarks plotted in a roofline model chart.

In this plot, the red points represent the theoretical peak performance of each benchmark based on their mapped arithmetic rate. The Jacobi 1D benchmark mapping requires a bandwidth that exceeds the memory bound for its arithmetic intensity, indicating that, even in the absence of overheads, it could never achieve the mapped rate. The blue points reflect the observed performance, calculated by applying the measured fire ratio to the arithmetic rate. As expected, the observed points fall below their theoretical counterparts, with benchmarks exhibiting lower arithmetic intensity showing a more pronounced drop.

By performing a least squares regression to fit an additional memory boundary to the observed results, it becomes evident that the current SE-L2 system achieves an effective bandwidth averaging 1.50 GB/s for the tested benchmarks. This value is significantly lower than the theoretical maximum bandwidth of the L2 cache, highlighting the system's limitations under practical conditions. Hence, and despite the implemented streaming solution, this result points to still existing persistent inefficiencies in the streaming mechanism. To help identify the source of these inefficiencies, Table 6.4 presents the hardware performance counter values collected during the accelerator simulations.

Table 6.4: Performance metrics from the SE performance counters.

Benchmark	**Jacobi 1D**	**Jacobi 2D**	**Heat 3D**	**Black-Scholes**	**FIR**	**GEMM**
Total cycles	25.15 M	30.53 M	57.48 M	70.70 M	51.79 M	40.16 G
Fire cycles	699 k	1.57 M	4.10 M	29.36 M	15.38 M	2.15 G
Higher iterations	838.9 k	8.65 M	30.75 M	25.16 M	8.39 M	6.98 G
LMMU commits	4.47 M	8.39 M	16.38 M	20.97 M	30.76 M	6.44 G
LMMU stalls	7.82 M	6.20 M	4.69 M	10.18 M	4.23 M	18.44 G
LF stalls	7.82 M	6.20 M	4.69 M	10.18 M	4.23 M	18.44 G
LRQ stalls	0	0	0	0	0	0
LLB stalls	1.19 M	18.76 M	31.16 M	0	0	38.37 G
SMMU commits	4.19 M	4.19 M	4.10 M	4.19 M	4.19 M	2.15 G
SMMU stalls	7.82 M	3.10 M	1.56 M	10.18 M	4.23 M	6.15 G
Load requests	279 k	1.13 M	3.23 M	1.31 M	1.30 M	865.3 M
Store requests	262 k	786 k	1.28 M	262 k	262 k	268.4 M

Altough this level of detail of the inner workings of the integrated SE modules exceeds the scope of this thesis, Appendix A includes a brief description of the load memory management unit (LMMU) equipping such SE [36] for context to the following analysis.

The presented experimental results demonstrate that load FIFO (LF) stalls and load line buffer (LLB) stalls are the main contributors to the observed performance issues. LLB stalls occur because the buffer becomes full when the LF does not consume its data fast enough. Similarly, LF stalls happen because the LF itself is full, either due to the stream registers not consuming data at a sufficient rate or because the LF buffers are still being filled with data from the LLB.

Given the relatively few fire cycles compared to the total cycles, it is unlikely that the stream registers are not being consumed, ruling out this scenario as a possible cause. In fact, both cases suggest that the primary issue is a slow data transfer from the LF to the LLB. This hypothesis is supported by an analysis of the current SE implementation [36], which shows that only one data element per cycle can be transferred from the LLB to the LF (see Appendix A).

A further evidence comes from the absence of load request queue (LRQ) stalls, which would occur if memory was the bottleneck. LRQ stalls would indicate that the SE is producing more requests than the memory can handle, leading to a backup in the LRQ. Since this is not observed, it reinforces that memory latency is not the limiting factor.

Although the store memory management unit (SMMU) does not provide detailed counters, analyzing

the simulation waveforms revealed similar causes. Another concern is that even if stalls were eliminated, the minimum number of cycles for the current SE implementation [36] would still equal the sum of all LMMU commits, SMMU commits, and all cycles spent iterating higher dimensions (i.e., descriptors that do not generate addresses). This is because the current SE implementation processes only one iteration of the stream descriptors per cycle, including higher-dimension descriptors that do not yield addresses.

This limitation explains the poor performance of higher-dimensionality benchmarks like Heat 3D and GEMM, where the high number of iteration cycles dominates the execution time.

However, despite the observed limitations in memory performance, the accelerator still demonstrates an improved overall performance, when comparing with other systems. Fig. 6.2 illustrates the obtained speedup (in terms of execution cycles) of the accelerator when compared to other systems, normalized to the performance of the scalar Rocket core, the lowest-performing baseline. The speedup is defined as:

$$\text{Speedup} = \frac{\text{Total execution cycles (evaluated system)}}{\text{Total execution cycles (Rocket)}}$$

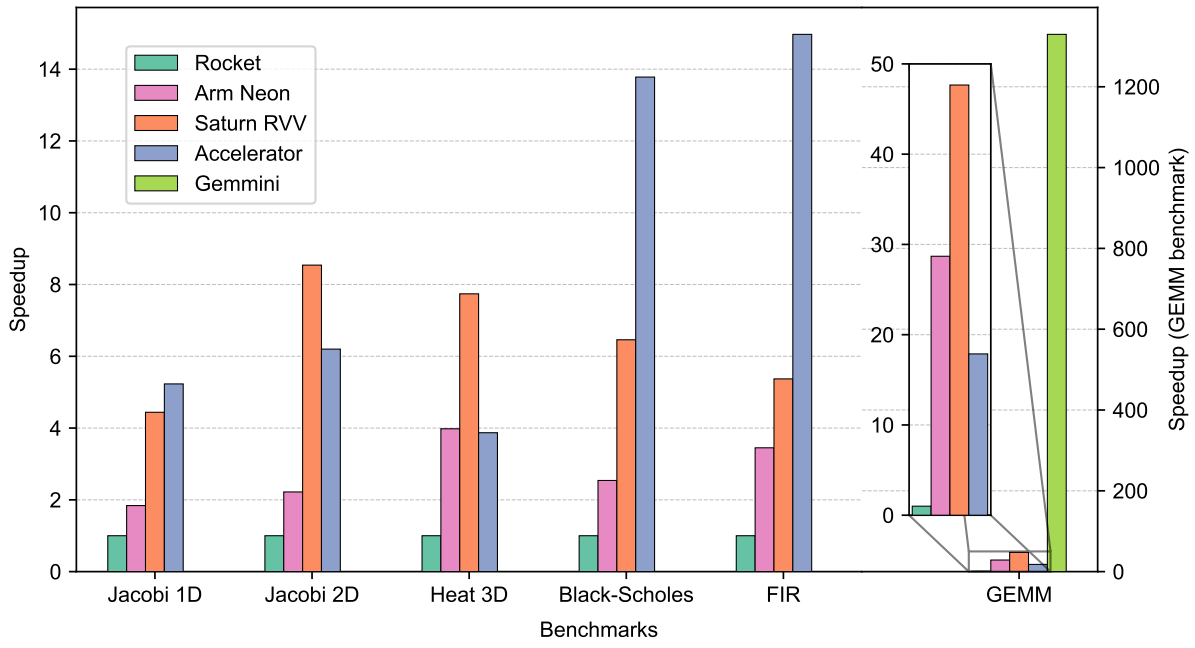


Figure 6.2: Execution cycles speedup normalized to Rocket core.

On average, the implemented accelerator achieved a speedup of 10.32× relative to Rocket, 2.83× against Arm Neon, but only 1.28× compared to Saturn. For benchmarks with the best performance gains, the proposed accelerator surpassed by 17.87×, 5.43×, and 2.79× the performance of Rocket, Arm Neon, and Saturn, respectively. As expected, and as a result of the described limitations of the SE, the accelerator maximized its performance when executing benchmarks with more arithmetic intensity (i.e. requiring lower data bandwidth) like the Black-Scholes and FIR. Conversely, when executing the higher dimensionality benchmarks, such as Heat 3D and GEMM, the acceelerator achieved a speedup lower than the Arm Neon and Saturn systems.

A direct comparison with Gemmini, which is a fixed-dataflow accelerator specialized in matrix multi-

plication, reveals that the proposed accelerator current implementation is still far from the performance of such a dedicated structure, only achieving a 17.87x speedup while the Gemmini achieved 1329× speedup on the GEMM benchmark.

6.3 Silicon area and power

This section presents the silicon area and power evaluation of the proposed accelerator, based on VLSI synthesis results. It discusses the contributions of the various components to the overall SoC area and power and compares the energy efficiency of the accelerator to other reference systems.

6.3.1 VLSI synthesis

To evaluate the silicon area and power characteristics of the proposed accelerator and provide comparative power values for other synthesizable IP architectures, the VLSI synthesis was performed using the Hammer VLSI flow [90], integrated into the Chipyard framework. Hammer is a physical design framework that abstracts vendor-specific technologies and tools to present a unified application programming interface (API) for ASIC development.

The synthesis used the ASAP7 process design kit (PDK) [91], a predictive 7-nm technology designed for academic research. Cadence Genus, version 21.15, was employed as the synthesis tool. To improve the accuracy of the synthesis process, sequential memory elements, represented as first-class primitives in Chisel, were extracted using Chipyard's MacroCompiler tool. These primitives were then passed to the ASAP7 SRAM compiler to generate technology-specific SRAM cells.

This workflow replaced all flip-flop-based memory arrays with SRAMs, providing a more realistic representation of memory structures in typical tapeout scenarios. By using SRAMs, instead of flip-flop arrays, area efficiency and power consumption metrics are significantly more accurate and realistic, aligning the synthesis process with normal design practices in VLSI technologies.

6.3.2 Results and discussion

Table 6.5 presents the area and power breakdown obtained from the synthesis of a complete SoC with the parameters described in Table 6.1. Additionally, Fig. 6.3 provides a visual representation of the synthesized area through a treemap chart, where the proportional size of each region corresponds to the synthesized area of its components.

The presented results also show that the accelerator accounts for approximately 32% of the total SoC area. Within the accelerator, the SE represents the largest area contributor, occupying 53% of the accelerator's footprint, followed by the mesh with 18%. Other significant components include the stream remappers with 15%, the skew synchronizers with 7%, and the controller, including its configuration memories, with 4%.

When considering the use of the ASAP7 process, the accelerator uses 0.149 mm^2 of area and consumes 50 mW of power. The full SoC occupies 0.460 mm^2 and has a power consumption of 70 mW.

Table 6.5: Area and power breakdown of the implemented SoC.

Component			Area (µm²)	Power (mW)
SoC	TOTAL		479 380	374.07
	Accelerator	TOTAL	174 343	311.14
		Mesh	41 302	156.25
		SE	87 386	114.98
		Stream remappers	22 045	25.22
		Skew synchronizers	10 475	8.67
		Controller	5 466	4.21
	L2		233 797	22.26
	Rocket Core	TOTAL	33 166	20.51
		L1D	10 333	4.07
		L1I	8 592	2.20

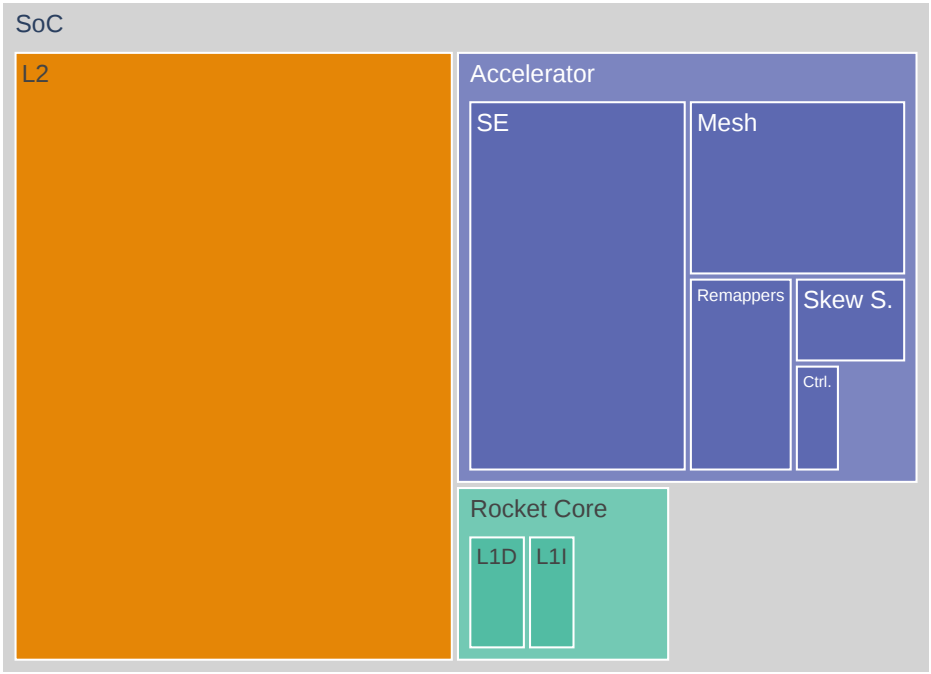


Figure 6.3: Synthesized area treemap.

Energy consumption for executing each benchmark was estimated as the product of the execution time and the synthesis power values. This approach simplifies the analysis by using the synthesized power values instead of performing post-place-and-route dynamic power simulations. For the proposed system, the used power value is the entire SoC's value, which includes the contributions from the Rocket core, as it handles the accelerator control, despite being available for computation during most of the accelerator's execution.

For the remaining considered systems implemented as synthesizable IPs, identical synthesis con-

ditions were applied to derive power values. The only exception is the Arm system, fabricated using a different technology than that used for synthesizing the accelerator, which was excluded from these energy efficiency comparisons. Including it would predominantly highlight differences arising from the fabrication technology rather than the architectural impact.

Figure 6.4 illustrates the energy improvement provided by the accelerator over the other reference systems, normalized to the Rocket system, which was the least energy-efficient baseline. The energy improvement is calculated as:

$$\text{Energy improvement} = \frac{\text{Energy consumption (Rocket)}}{\text{Energy consumption (evaluated system)}}$$

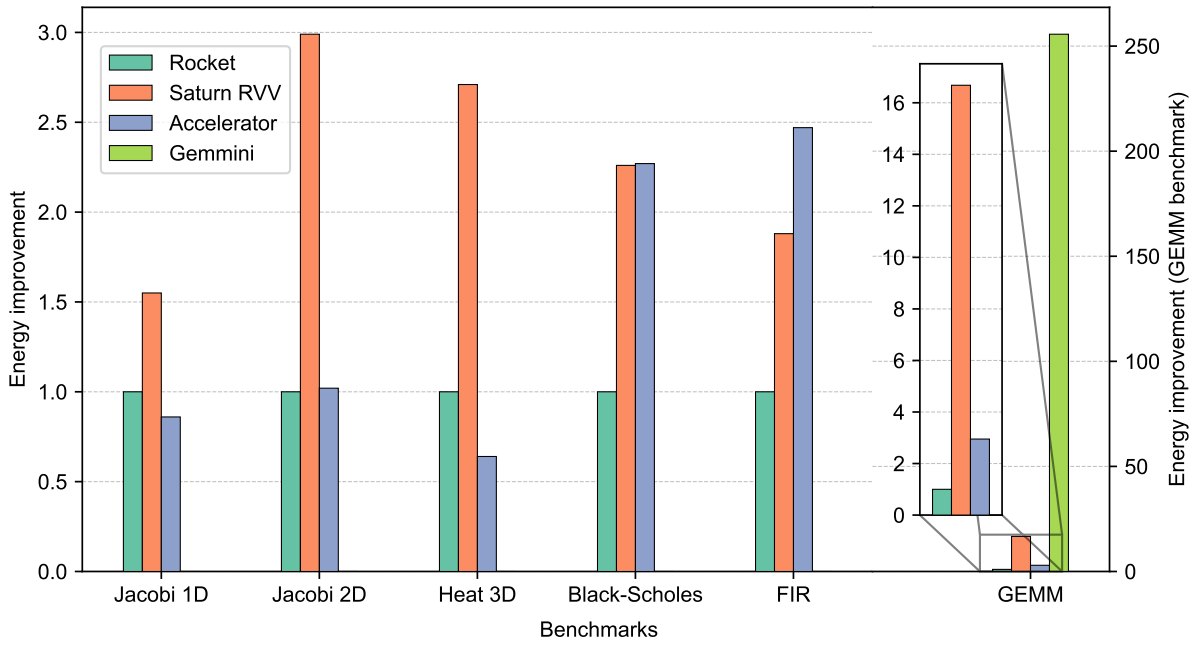


Figure 6.4: Energy improvement normalized to Rocket.

Since energy usage is highly related to the execution time, the energy improvement results exhibit similar trends to the speedup results discussed in the previous section. However, they are further impacted by the accelerators' higher power requirements. When stalls occur, they are particularly costly, as the infrastructure continues consuming power without producing any useful work, leading to a greater energy waste.

According to Fig. 6.4, the accelerator achieves an average energy improvement of 1.70× when compared to the scalar Rocket core, but only 0.61× compared to the Saturn system, indicating that the current implementation is less efficient than Saturn. Even in the best-performing benchmarks, the accelerator achieves a 2.95× energy improvement over Rocket but just 1.31× over Saturn. When compared to Gemmini running in the GEMM benchmark, the proposed accelerator delivers only a 2.95× improvement over Rocket, whereas Gemmini, with its specialization in matrix multiplication, achieves a 256× energy improvement. This stark contrast underscores the substantial impact of the identified overheads on the energy efficiency of the accelerator.

Despite these results, the accelerator SoC managed to use roughly the same area as the Gemmini SoC (0.48 mm^2 vs. 0.52 mm^2) and demonstrated a comparable power consumption (374 mW vs. 321 mW). This indicates that the added flexibility does not come at a significant cost in terms of resource utilization. If the highlighted limitations can be addressed without substantial increases in area or power consumption, the accelerator will deliver a much greater flexibility while maintaining rather competitive physical constraints.

6.4 Proposed improvements

This section explores potential solutions to mitigate the overheads identified in prior sections, specifically addressing challenges in the mapper and SE as well as hardware resource inefficiencies. The potential impact of these improvements on performance, area, and energy efficiency is also estimated.

6.4.1 Compilation and mapping

The complexity of the mapping process still remains a critical challenge, constraining the efficiency of the compilation flow for CGRAs, as previously identified.

Specifically in the case of GEMM benchmarks, if the compiler could recognize accumulation patterns and map them directly onto the architecture, it would eliminate the memory operation overheads caused by recurrences between iterations. In fact, instead of storing the accumulation state at each iteration and reloading it in the next, the data could remain within the computational array throughout iterations. This improvement would significantly increase the arithmetic intensity of benchmarks exhibiting reduction patterns.

However, the current reliance on LLVM intermediate representation introduces strict limitations due to its strong orientation toward traditional CPU architectures, making it less effective to represent the dataflow paradigms inherent to CGRAs. Addressing this will require the development of CGRA compilers that operate at higher levels of abstraction, potentially utilizing modern intermediate representations like MLIR [92]. Such advancements could provide a standardized and flexible foundation for CGRA compilation, enabling diverse mapping algorithms to build on a shared representation rather than reinventing fundamental components.

6.4.2 Streaming engine

The main current bottleneck of the SE is concerned with the fact that it can only deal with (at most) one element per cycle. To address this, the following improvements are proposed:

1. **Parallel address generation**: The SE should be able to simultaneously generate multiple addresses in parallel, solving multiple iterations of the descriptors in a single cycle.

2. **Parallel descriptor iteration**: Iterating over higher dimensions of a stream descriptor should happen in parallel with the processing of lower dimensions, avoiding any extra cycles dedicated to

iterating higher dimensions where no addresses are generated.

3. **Faster vector register population**: The SE must provide a faster population of the vector registers by processing multiple data elements concurrently from the LLB (containing the cache lines) to the LF (containing the stream vectors).

4. **Multiple in-flight memory requests**: The current SE implementation issues the memory requests sequentially, waiting for replies before sending new requests. This approach limits the utilization of the available memory bandwidth and prevents the SE from fully saturating the L2 cache bandwidth, due to the latency in the responses. Enhancing the SE to support multiple concurrent in-flight memory requests would alleviate this bottleneck.

It is important to recall that most SE resources are actually composed by its tables and buffers. Since none of these changes would require significantly more table or buffer space, but rather some additional parallel logic, it is not anticipate any significant resource overheads.

6.4.3 Hardware resources

In the current implementation, the design of certain hardware components, such as remappers and skew synchronizers, scales linearly with the maximum II, which is not strictly necessary. In the current implementation, the stream remappers can remap all stream elements for an entire II in a single cycle. However, this process could instead span the entire II duration, meaning the hardware does not need to depend on the II but rather on the mesh I/O.

For the skew synchronizers, most of the microstreams in a kernel do not require delays, and many mesh ports are left unused. Instead of having dedicated synchronizers for each time-extended mesh port, which scales with the II, a small pool of shared skew synchronizers could be used. Delays would only be introduced where necessary, leveraging the existing permutation network to redirect the microstreams through these shared synchronizers.

By decoupling their resource requirements dependence from the II, the design could support larger computational kernels without increasing the hardware cost, thereby improving scalability and area efficiency.

6.4.4 Estimated results

To estimate the performance of a hypothetical system implementing the proposed improvements, the roofline model shown in Fig. 6.5 was utilized. The performance improvement resulting from the proposed mapping is indicated by the figure's light arrows, which represent the transition from the benchmark metrics achieved by the current mapping methods to what is theoretically possible. These improvements are achieved in two ways. First, a vertical shift in performance is assumed by considering that the improved mapping achieves a perfect QoM. Second, a rightward shift in arithmetic intensity is granted by the proposed elimination of load-store recurrences through the added support for reductions, as previously proposed, and by increasing tile sizes in benchmarks where arithmetic intensity depends on

this factor. This increase in tile size is considered feasible with the implementation of a more efficient mapping algorithm.

Two scenarios are considered to estimate the bandwidth achieved by an improved SE. The first assumes the L2's theoretical maximum bandwidth of 32 GB/s. Although unrealistic for most benchmarks, this scenario could be achievable under ideal conditions for highly regular workloads, such as GEMM, as demonstrated by Gemmini. The second scenario, which is more realistic, assumes that the improved SE and memory system achieve half the theoretical maximum bandwidth, or 16 GB/s. This more realistic scenario accounts for hypothetical overheads, such as cache misses and memory latency.

In the roofline model depicted with these improved mappings, only the Jacobi 1D, Jacobi 2D, and Heat 3D benchmarks are predicted to remain memory-bound. This is illustrated in Fig. 6.5 by the blue dashed line and the black arrows.

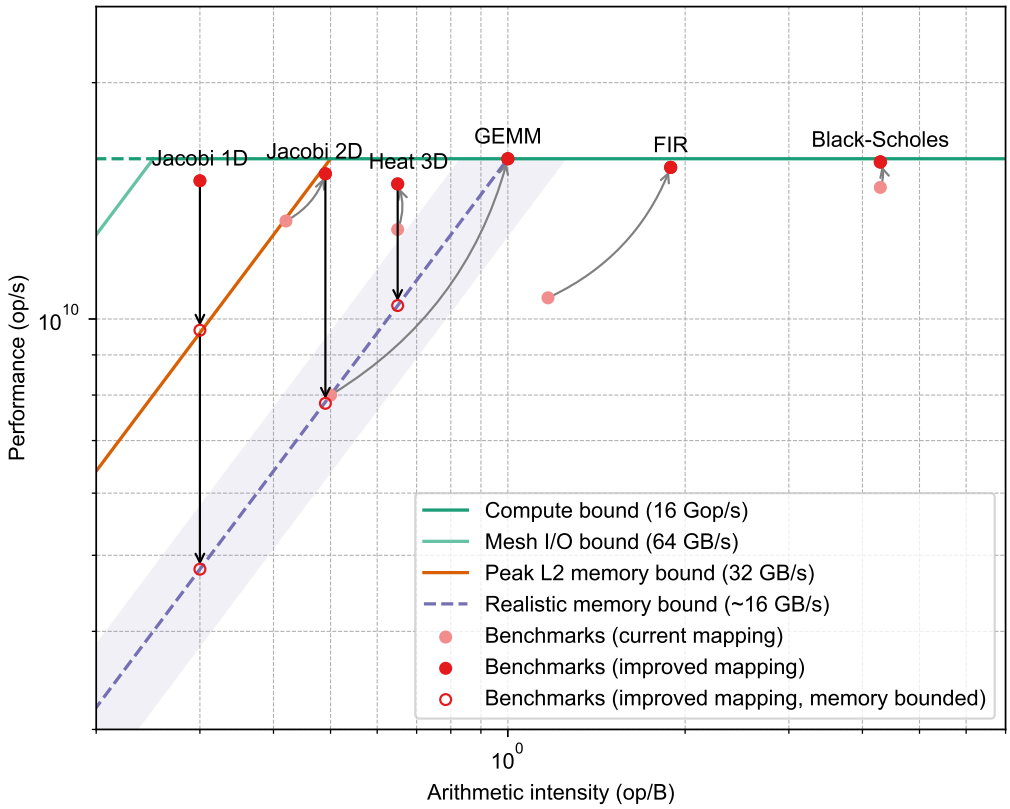


Figure 6.5: Benchmarks in a roofline model, according to estimated improvements in the mapping and the acceelerator.

Fig. 6.6 compares the improved system's estimated performance by considering the same baselines used for evaluating the current implementation. This chart includes three configurations: the proposed accelerator running the current benchmark mappings but assuming the maximum memory bandwidth of 32 GB/s; the proposed accelerator with improved mappings and the same maximum bandwidth; and a more realistic version of the accelerator with improved mappings but achieving only half the theoretical maximum bandwidth, or 16 GB/s.

For the more realistic 16 GB/s-bounded version, the accelerator is estimated to achieve an average speedup of 952× relative to Rocket, 61× compared to Saturn, 134× over Arm Neon, and 79× over the

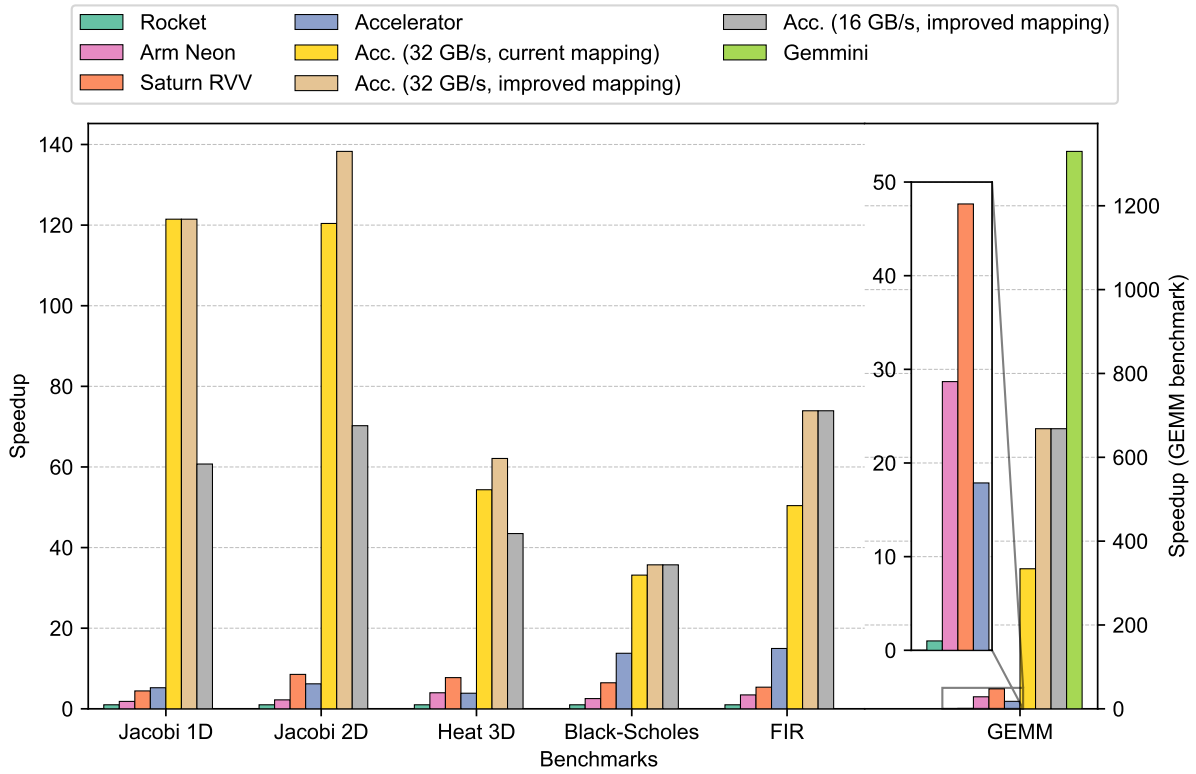


Figure 6.6: Execution cycles speedup including the proposed accelerator improvements, normalized to Rocket.

current accelerator implementation. In benchmarks with the highest performance gains, the estimated speedups are 668×, 14×, 33×, and 37× relative to Rocket, Saturn, Arm Neon, and the current accelerator implementation, respectively.

Hence, the presented estimates showcase that despite its generalized nature, the accelerator demonstrates rather competitive performance even against highly specialized systems, such as the Gemmini matrix-multiplication DSA. For example, in the GEMM benchmark, a realistic version of the improved accelerator is estimated to achieve a 668× speedup over Rocket, which is already half the speedup achieved by Gemmini. Notably, Gemmini achieves its performance using fused multiply-accumulate operations that execute both a multiplication and an addition in a single cycle. If a compilation system supporting the representation of these kinds of fused operations were developed and the trivial microarchitectural changes required to support it were implemented, achieving similar results would be entirely plausible. This estimated result highlights the potential of the proposed architecture to support a broader range of applications with minimal compromise in domain-specific performance.

For the energy improvement estimation, the same power values as the current implementation were assumed, as the SE improvements are not expected to result in significant power or area overheads. Furthermore, the other proposed microarchitectural improvements are anticipated to reduce hardware resource requirements.

Fig. 6.7 illustrates the estimated energy improvements for the 16 GB/s-bounded version of the improved accelerator, normalized to Rocket, which serves as the least energy-efficient baseline. On aver-

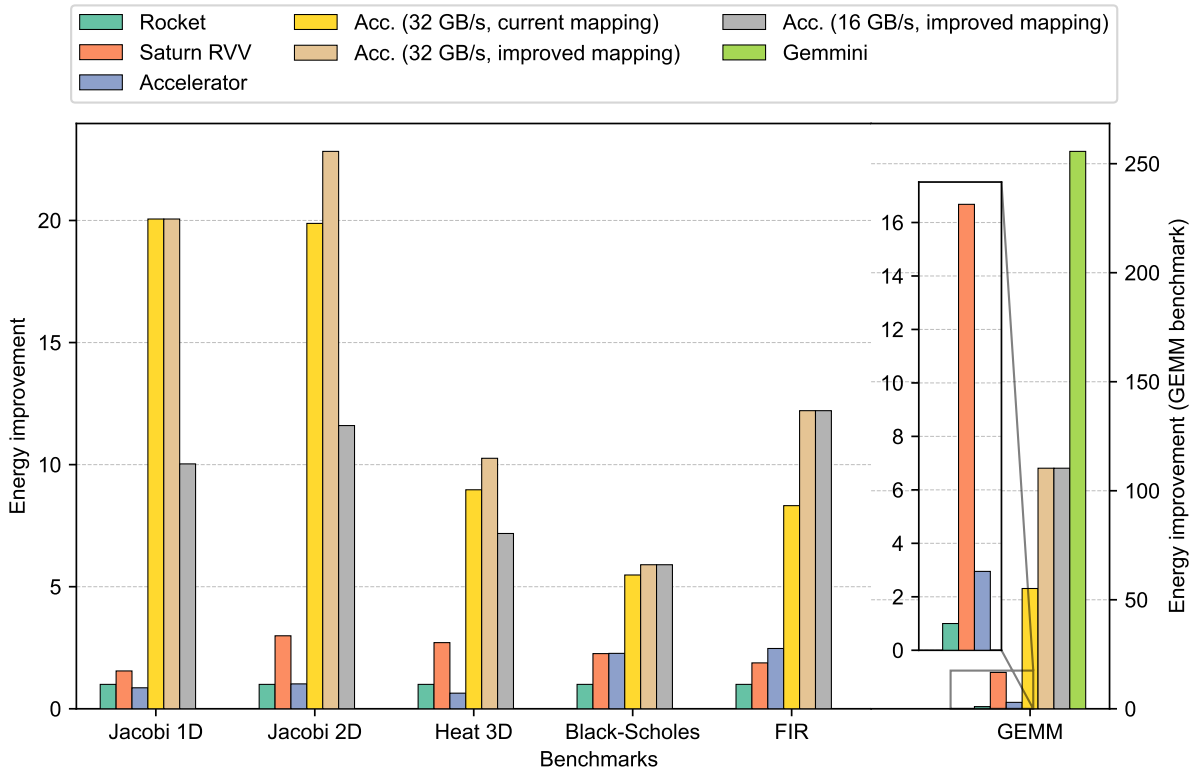


Figure 6.7: Energy improvement including estimated accelerator improvements, normalized to Rocket.

age, the improved accelerator is estimated to achieve 26× energy improvement over Rocket, 4.8× over Saturn, and 13× over the current accelerator implementation. For benchmarks with the best energy gains, the improvements are 110×, 6.6×, and 37× over Rocket, Saturn, and the current accelerator implementation, respectively. In the GEMM benchmark, the improved accelerator is estimated to achieve about 43% of the Gemmini's energy gain, resulting in only 2.3× more energy usage despite Gemmini's specialization in the matrix multiplication operation.

6.5 Summary

This chapter presented a comprehensive evaluation of the proposed computational accelerator in terms of performance, hardware resource utilization, and energy efficiency. The system was compared to both open-source and commercial references under common conditions, using a range of benchmarks to characterize different computational and memory access patterns. Key experimental results highlighted the accelerator's strengths in compute-intensive workloads, as well as the impact of the mapping and SE's bottlenecks on overall efficiency.

Analysis of performance counters and synthesis outcomes provided insight into the main sources of overhead, particularly in the SE's address generation rate. The chapter also explored potential architectural and toolchain improvements, with estimated results illustrating how targeted changes could further enhance system performance and resource efficiency. These observations inform subsequent discussion regarding future directions and refinements.

7

Conclusions

Contents

7.1 Work summary and contributions

This work presented the design, implementation, and evaluation of a novel reconfigurable accelerator architecture supported by an integrated streaming solution. A corresponding computational model was also developed, along with a companion end-to-end software framework to implement it. The accelerator was integrated within a complete SoC built using the Chipyard framework, providing a realistic evaluation environment. This integration facilitated detailed comparisons of performance, area, and energy efficiency against other solutions, including both general-purpose and domain-specific architectures.

Experimental results obtained from a synthesized 7 nm implementation revealed performance and energy efficiency gains compared to conventional scalar and vector GPP architectures. However, these results also highlighted some existing limitations in the implementation of the streaming solution and the mapping process. These limitations were identified, and corresponding solutions were proposed. An estimated evaluation of a system incorporating these improvements further validated the stream-driven model's adaptability in high-demand application domains. Comparisons with fixed-dataflow DSA architectures, such as Gemmini, demonstrated that the proposed architecture can achieve a comparable performance and energy efficiency, while retaining the flexibility inherent to reconfigurable designs. These findings also suggest that future systems could significantly reduce the number of specialized blocks or accelerators by adopting a stream-supported, dataflow-driven reconfigurable architectures. Such an approach would allow significant reductions of the silicon area and complexity costs, while mitigating the growing verification overhead associated with integrating an increasing number of specialized functional units in modern systems.

7.2 Existing limitations

Since the identified limitatations fall outside the scope that is possible in a single Masters dissertation, the proposed solutions had to be left for further exploration.

Compiler challenges

A key challenge is the low level of abstraction that is considered by current tools for CGRAs. The toolchain relies on traditional GPP-oriented models, which limit its ability to efficiently represent and map operations like reductions and accumulations. As a result, these operations are implemented inefficiently, leading to excessive memory accesses, larger DFGs, and higher initiation intervals (IIs), all of which reduce performance.

Although more sophisticated mapping methods and tools already exist, they are not readily available or easily integrated into mainstream toolchains. The current tools are in a crude state, with limited accessibility, documentation, and support, which hinders their adoption and usability. Frameworks like MLIR [92] offer a promising path toward higher abstraction levels and better support for dataflow paradigms. However, significant effort is still needed to develop mature and user-friendly compiler toolchains that can optimize the CGRAs, simplify their programming, and seamlessly integrate them into existing workflows.

Streaming engine limitations

The current SE [36], while functional, is not fully optimized for the demands of high-performance accelerators. It is limited to processing only one element per cycle, which significantly restricts data throughput and prevents the full utilization of the available memory bandwidth. Additionally, the SE issues the memory requests sequentially, waiting for replies before sending new ones. This approach significantly limits its ability to saturate the L2 cache bandwidth, making it a bottleneck for the overall system performance.

To address these limitations, several improvements are proposed. Parallel address generation is needed to handle multiple iterations of stream descriptors in a single cycle. Iterations over higher-dimensional stream descriptors should occur in parallel with processing lower dimensions, avoiding wasted cycles where no addresses are generated. Faster vector register population would also enable the SE to process multiple data elements simultaneously, accelerating data transfers. Finally, supporting of multiple concurrent in-flight memory requests would eliminate the sequential request-response bottleneck, improving memory handling and fully utilizing the available bandwidth. Together, these enhancements would significantly improve the data throughput and allow the accelerator to better achieve its performance potential.

Hardware resource utilization

In the current implementation, the design of certain hardware components, such as stream remappers and skew synchronizers, scales unnecessarily with the maximum initiation interval (II). This scaling introduces inefficiencies in hardware resource utilization that could be addressed with alternative design approaches.

The stream remappers can currently remap all stream elements for an entire II in a single cycle. This operation could instead be spread across the II duration, making it independent of the II and reliant only on the mesh input-output (I/O) requirements, reducing the hardware demands.

Similarly, skew synchronizers are assigned to each time-extended mesh port, even though most microstreams do not require delays and many ports remain unused. A shared pool of synchronizers could be introduced to handle delays only when necessary, using the existing permutation network to direct the microstreams as needed.

By decoupling the resource requirements of these components from the II, the design could support larger computational kernels without increasing the hardware costs. This would significantly improve scalability and area efficiency, enhancing the architecture's adaptability to a wide range of workloads.

7.3 Recommendations for future research

To address these limitations, future efforts should focus on:

1. **Advanced compiler development**: Create compilers supporting improved abstraction levels and better mapping algorithms for CGRAs. MLIR stands out as a strong common standard, offering

the right level of abstraction and portability to make compiler and mapping improvements easily reusable across projects.

2. **Improved streaming engine**: Redesign the SE to handle multiple in-flight memory requests, enable parallel address generation, and improve data throughput to fully utilize memory bandwidth.

3. **Hardware optimization**: Revise some components design, such as the remappers and synchronizers, to eliminate unnecessary dependencies on II, reducing resource usage and improving scalability.

4. **Improved toolchains**: Create more accessible and feature-complete tools to enhance usability and broaden adoption.

5. **Expanded benchmarking**: Test the architecture on a broader range of workloads to validate performance and efficiency across diverse applications.

7.4 Final remarks

This thesis demonstrated the potential of a flexible and reconfigurable accelerator architecture that integrates data streaming with CGRA acceleration. The proposed system achieves substantial improvements in performance and energy efficiency, while offering versatility levels that are not found in traditional DSAs. By addressing the outlined limitations, future iterations can further capitalize on these advantages to meet the growing computational demands of modern applications.

Through the envisaged advancements in compiler technology, hardware optimization, and streaming efficiency, this work provides a foundation for the widespread adoption of stream-based reconfigurable accelerators. The findings underscore the viability of stream-driven architectures as a compelling alternative to specialized fixed-function accelerators, enabling greater adaptability without sacrificing efficiency.

Bibliography

[1] G. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86 (1):82–85, Jan. 1998. ISSN 1558-2256. doi: 10.1109/JPROC.1998.658762. Conference Name: Proceedings of the IEEE.

[2] S. E. Thompson and S. Parthasarathy. Moore's law: the future of Si microelectronics. *Materials Today*, 9(6):20–25, June 2006. ISSN 1369-7021. doi: 10.1016/S1369-7021(06)71539-5. URL https://www.sciencedirect.com/science/article/pii/S1369702106715395.

[3] M. M. Waldrop. The chips are down for Moore's law. *Nature News*, 530(7589): 144, Feb. 2016. doi: 10.1038/530144a. URL http://www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338. Cg_type: Nature News Section: News Feature.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, June 2011. Association for Computing Machinery. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. URL https://doi.org/10.1145/2000064.2000108.

[5] B. Peccerillo, M. Mannino, A. Mondelli, and S. Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives. *Journal of Systems Architecture*, 129:102561, Aug. 2022. ISSN 1383-7621. doi: 10.1016/j.sysarc.2022.102561. URL https://www.sciencedirect.com/science/article/pii/S1383762122001138.

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, and A. Borchers. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[7] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, June 2019. ISSN 2156-3365. doi: 10.1109/JETCAS.2019.2910232. Conference Name: IEEE Journal on Emerging and Selected Topics in Circuits and Systems.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture*

News, 42(1):269–284, Feb. 2014. ISSN 0163-5964. doi: 10.1145/2654822.2541967. URL https://dl.acm.org/doi/10.1145/2654822.2541967.

[9] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774, Dec. 2021. doi: 10.1109/DAC18074.2021.9586216. ISSN: 0738-100X.

[10] Samsung. Exynos 990 Mobile Processor. URL https://semiconductor.samsung.com/processor/mobile-processor/exynos-990.

[11] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. ShiDianNao: shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 92–104, New York, NY, USA, June 2015. Association for Computing Machinery. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2750389. URL https://dl.acm.org/doi/10.1145/2749469.2750389.

[12] Y. Turakhia, G. Bejerano, and W. J. Dally. Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly. *SIGPLAN Not.*, 53(2):199–213, Mar. 2018. ISSN 0362-1340. doi: 10.1145/3296957.3173193. URL https://dl.acm.org/doi/10.1145/3296957.3173193.

[13] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally. Darwin-WGA: A Co-processor Provides Increased Sensitivity in Whole Genome Alignments with High Speedup. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 359–372, Feb. 2019. doi: 10.1109/HPCA.2019.00050. URL https://ieeexplore.ieee.org/abstract/document/8675186?casa_token=iUp7JKTSCuYAAAAA:dhBUWekbyZGJuW0e6YIRw_MBH7pnLOlgGVyIwJXSvIubVzrhPA3fTxXsivjylL3IZJMup7EsCQ. ISSN: 2378-203X.

[14] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564, Feb. 2017. doi: 10.1109/HPCA.2017.29. URL https://ieeexplore.ieee.org/abstract/document/7920855?casa_token=TBHNUg2Q_VMAAAAA:T_NsDEM_P-Z44dtXLgLU6cDs1CnVBCJjBrak336pR08hA1i_z-VtncTnHLKcFhZNKlLGZwJe2Q. ISSN: 2378-203X.

[15] NVIDIA. NVIDIA Deep Learning Accelerator. Technical report, 2018. URL https://nvdla.org/primer.html.

[16] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin,

R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 145–158, May 2020. doi: 10.1109/ISCA45697.2020.00023. URL https://ieeexplore.ieee.org/document/9138986.

[17] WikiChip. FSD Chip - Tesla. URL https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip#NPU.

[18] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):48–60, Jan. 2018. ISSN 1937-4151. doi: 10.1109/TCAD.2017.2682138. URL https://ieeexplore.ieee.org/abstract/document/7878541?casa_token=7mTGT4Tc2yQAAAAA:FE2uX3Ov46qhdhL7jsDRD__UBpcj61yDi_hCaxoQRhS0BzmFM-LNFJdgGluUHZQ8097ZR3ywFw. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[19] Wafer-Scale Deep Learning. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–31, Aug. 2019. doi: 10.1109/HOTCHIPS.2019.8875628. URL https://ieeexplore.ieee.org/document/8875628/citations?tabFilter=papers#citations. ISSN: 2573-2048.

[20] M. Khazraee, L. Zhang, L. Vega, and M. B. Taylor. Moonwalk: NRE Optimization in ASIC Clouds. *SIGARCH Comput. Archit. News*, 45(1):511–526, Apr. 2017. ISSN 0163-5964. doi: 10.1145/3093337.3037749. URL https://dl.acm.org/doi/10.1145/3093337.3037749.

[21] J. Dean, D. Patterson, and C. Young. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro*, 38(2):21–29, Mar. 2018. ISSN 1937-4143. doi: 10.1109/MM.2018.112130030. Conference Name: IEEE Micro.

[22] J. L. Hennessy and D. A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, Jan. 2019. ISSN 0001-0782. doi: 10.1145/3282307. URL https://doi.org/10.1145/3282307.

[23] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–429, June 2017. doi: 10.1145/3079856.3080255.

[24] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, June 2017. doi: 10.1145/3079856.3080256.

[25] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In P. Y. K. Cheung and G. A. Constantinides, editors, *Field Programmable Logic and Application*, pages 61–70, Berlin, Heidelberg, 2003. Springer. ISBN 978-3-540-45234-8. doi: 10.1007/978-3-540-45234-8_7.

[26] M. Karunaratne, A. K. Mohite, T. Mitra, and L.-S. Peh. HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 1–6, New York, NY, USA, June 2017. Association for Computing Machinery. ISBN 978-1-4503-4927-7. doi: 10.1145/3061639.3062262. URL https://dl.acm.org/doi/10.1145/3061639.3062262.

[27] D. Wijerathne, Z. Li, M. Karunarathne, A. Pathania, and T. Mitra. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Transactions on Embedded Computing Systems*, 18(5s):50:1–50:26, Oct. 2019. ISSN 1539-9087. doi: 10.1145/3358177. URL https://dl.acm.org/doi/10.1145/3358177.

[28] N. Neves, P. Tomás, and N. Roma. A Reconfigurable Posit Tensor Unit with Variable-Precision Arithmetic and Automatic Data Streaming. *Journal of Signal Processing Systems*, 93(12):1365–1385, Dec. 2021. ISSN 1939-8115. doi: 10.1007/s11265-021-01687-7. URL https://doi.org/10.1007/s11265-021-01687-7.

[29] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1027–1040, June 2021. doi: 10.1109/ISCA52012.2021.00084. URL https://ieeexplore.ieee.org/abstract/document/9499726?casa_token=cL02Jz0KfEkAAAAA:BUwJ3kFAZ55W-vDsbDaDM4Br_h2yHXi5jE2ul2UixqRTNdFE5DTjEuj2157v-fi5Lr4ZwnEnZg. ISSN: 2575-713X.

[30] M. Abbaszadeh, T. S. Abdelrahman, R. Azimi, T. S. Czajkowski, and M. Goudarzi. Efficient Data Streaming for a Tightly-Coupled Coarse-Grained Reconfigurable Array. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 435–443, May 2023. doi: 10.1109/IPDPSW59300.2023.00075.

[31] Y. Qiu, Y. Mao, X. Gao, S. Chen, J. Li, W. Yin, and L. Wang. FDRA: A Framework for a Dynamically Reconfigurable Accelerator Supporting Multi-Level Parallelism. *ACM Transactions on Reconfigurable Technology and Systems*, 17(1):4:1–4:26, Jan. 2024. ISSN 1936-7406. doi: 10.1145/3614224. URL https://dl.acm.org/doi/10.1145/3614224.

[32] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. *ACM Trans. Embed. Comput. Syst.*, 22(2):35:1–35:34, Jan. 2023. ISSN 1539-9087. doi: 10.1145/3534933. URL https://dl.acm.org/doi/10.1145/3534933.

[33] J. Deng, L. Zhang, L. Wang, J. Liu, K. Deng, S. Tang, J. Gu, B. Han, F. Xu, L. Liu, S. Wei, and S. Yin. Mixed-granularity parallel coarse-grained reconfigurable architecture. In *Proceedings of the 59th*

ACM/IEEE Design Automation Conference, DAC '22, pages 343–348, New York, NY, USA, Aug. 2022. Association for Computing Machinery. ISBN 978-1-4503-9142-9. doi: 10.1145/3489517.3530454. URL https://dl.acm.org/doi/10.1145/3489517.3530454.

[34] C. Tan, N. B. Agostini, T. Geng, C. Xie, J. Li, A. Li, K. J. Barker, and A. Tumeo. DRIPS: Dynamic Rebalancing of Pipelined Streaming Applications on CGRAs. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 304–316, Apr. 2022. doi: 10.1109/HPCA53966.2022.00030. URL https://ieeexplore.ieee.org/document/9773269. ISSN: 2378-203X.

[35] J. M. Domingos, N. Neves, N. Roma, and P. Tomás. Unlimited Vector Extension with Data Streaming Support. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 209–222, June 2021. doi: 10.1109/ISCA52012.2021.00025. ISSN: 2575-713X.

[36] X. B. Fernandes. Data-Streaming Engine Architecture for the Unlimited Vector Extension. Master's thesis, Instituto Superior Técnico, 2023.

[37] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, July 2020. ISSN 1937-4143. doi: 10.1109/MM.2020.2996616. Conference Name: IEEE Micro.

[38] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 29–42, June 2018. doi: 10.1109/ISCA.2018.00014. URL https://ieeexplore.ieee.org/abstract/document/8416816. ISSN: 2575-713X.

[39] J. Maia, A. Silveira, G. Midões, N. Neves, P. Tomás, and N. Roma. Stream-Driven Acceleration for Embedded RISC-V SoCs. In *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2025. doi: 10.1109/ISCAS56072.2025.11043244. URL https://ieeexplore.ieee.org/abstract/document/11043244.

[40] NVIDIA. NVIDIA Tesla V100 GPU Architecture. White paper, 2017. URL https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[41] Train and run machine learning models faster | Cloud TPU. URL https://cloud.google.com/tpu.

[42] Kung. Why systolic architectures? *Computer*, 15(1):37–46, Jan. 1982. ISSN 1558-0814. doi: 10.1109/MC.1982.1653825. Conference Name: Computer.

[43] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, volume 1, pages 256–282. Society for industrial and applied mathematics Philadelphia, PA, USA, 1979.

[44] NVIDIA. NVIDIA H100 Tensor Core GPU Architecture. White paper, 2022. URL https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[45] Gemmini, Oct. 2018. URL https://github.com/ucb-bar/gemmini.

[46] Berkeley Architecture Research. *Chipyard Documentation, Release 1.11.0*. Jan. 2024. URL https://chipyard.readthedocs.io/en/1.11.0/.

[47] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Computing Surveys*, 52(6):118:1–118:39, Oct. 2019. ISSN 0360-0300. doi: 10.1145/3357375. URL https://doi.org/10.1145/3357375.

[48] J. L. Gustafson and I. T. Yonemoto. Beating Floating Point at its Own Game: Posit Arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, Apr. 2017. ISSN 2313-8734. doi: 10.14529/jsfi170206. URL https://www.superfri.org/index.php/superfri/article/view/137. Number: 2.

[49] Posit Working Group. *Posit Standard Documentation, Release 3.2*. June 2018.

[50] S. Das, K. Martin, T. Peyret, and P. Coussy. An Efficient and Flexible Stochastic CGRA Mapping Approach. *ACM Transactions on Embedded Computing Systems*, 22(1):8:1–8:24, Oct. 2022. ISSN 1539-9087. doi: 10.1145/3550071. URL https://dl.acm.org/doi/10.1145/3550071.

[51] B. R. Rau. Iterative Modulo Scheduling. *International Journal of Parallel Programming*, 24(1):3–64, Feb. 1996. ISSN 1573-7640. doi: 10.1007/BF03356742. URL https://doi.org/10.1007/BF03356742.

[52] D. Wijerathne, Z. Li, M. Karunaratne, L.-S. Peh, and T. Mitra. Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA [Presentation slides], Nov. 2022. URL https://woset-workshop.github.io/PDFs/2022/12-Wijerathne-presentation.pdf.

[53] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings - Computers and Digital Techniques*, 150(5):255, 2003. ISSN 13502387. doi: 10.1049/ip-cdt:20030833. URL https://digital-library.theiet.org/content/journals/10.1049/ip-cdt_20030833.

[54] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele. HiMap: Fast and Scalable High-Quality Mapping on CGRA via Hierarchical Abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(10):3290–3303, Oct. 2022. ISSN 1937-4151. doi: 10.1109/TCAD.2021.3132551. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[55] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *Proceedings of the ACM/SIGDA international symposium on Field*

programmable gate arrays, FPGA '09, pages 191–200, New York, NY, USA, Feb. 2009. Association for Computing Machinery. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508158. URL https://dl.acm.org/doi/10.1145/1508128.1508158.

[56] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng. Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–15, Limassol Cyprus, Nov. 2018. ACM. ISBN 978-1-4503-5986-3. doi: 10.1145/3243176.3243212. URL https://dl.acm.org/doi/10.1145/3243176.3243212.

[57] S. Dave, M. Balasubramanian, and A. Shrivastava. URECA: Unified register file for CGRAs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1081–1086, Mar. 2018. doi: 10.23919/DATE.2018.8342172. URL https://ieeexplore.ieee.org/abstract/document/8342172. ISSN: 1558-1101.

[58] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pages 166–173, Dec. 2002. doi: 10.1109/FPT.2002.1188678.

[59] S. Dave, M. Balasubramanian, and A. Shrivastava. RAMP: resource-aware mapping for CGRAs. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 1–6, New York, NY, USA, June 2018. Association for Computing Machinery. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3196101. URL https://dl.acm.org/doi/10.1145/3195970.3196101.

[60] Z. Li, D. Wijerathne, X. Chen, A. Pathania, and T. Mitra. ChordMap: Automated Mapping of Streaming Applications Onto CGRA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(2):306–319, Feb. 2022. ISSN 1937-4151. doi: 10.1109/TCAD.2021.3058313. URL https://ieeexplore.ieee.org/abstract/document/9351547. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[61] L. Chen and T. Mitra. Graph Minor Approach for Application Mapping on CGRAs. *ACM Transactions on Reconfigurable Technology and Systems*, 7(3):21:1–21:25, Sept. 2014. ISSN 1936-7406. doi: 10.1145/2655242. URL https://dl.acm.org/doi/10.1145/2655242.

[62] M. Hamzeh, A. Shrivastava, and S. Vrudhula. REGIMap: register-aware application mapping on coarse-grained reconfigurable architectures (CGRAs). In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 1–10, New York, NY, USA, May 2013. Association for Computing Machinery. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488756. URL https://dl.acm.org/doi/10.1145/2463209.2488756.

[63] S. A. Chin and J. H. Anderson. An architecture-agnostic integer linear programming approach to CGRA mapping. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18,

pages 1–6, New York, NY, USA, June 2018. Association for Computing Machinery. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3195986. URL https://dl.acm.org/doi/10.1145/3195970.3195986.

[64] Z. Li, D. Wu, D. Wijerathne, and T. Mitra. LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 444–459, Apr. 2022. doi: 10.1109/HPCA53966.2022.00040. ISSN: 2378-203X.

[65] D. Wijerathne, Z. Li, T. K. Bandara, and T. Mitra. PANORAMA: divide-and-conquer approach for mapping complex loop kernels on CGRA. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 127–132, New York, NY, USA, Aug. 2022. Association for Computing Machinery. ISBN 978-1-4503-9142-9. doi: 10.1145/3489517.3530429. URL https://dl.acm.org/doi/10.1145/3489517.3530429.

[66] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 11th international conference on Supercomputing*, pages 317–324, 1997.

[67] A. López-Lagunas and S. M. Chai. Memory bandwidth optimization through stream descriptors. In *Proceedings of the 2005 workshop on MEmory performance: DEaling with Applications , systems and architecture*, MEDEA '05, pages 57–64, USA, Sept. 2005. IEEE Computer Society. doi: 10.1145/1152779.1147360. URL https://dl.acm.org/doi/10.1145/1152779.1147360.

[68] N. Neves, J. M. Domingos, N. Roma, P. Tomás, and G. Falcao. Compiling for Vector Extensions With Stream-Based Specialization. *IEEE Micro*, 42(5):49–58, Sept. 2022. ISSN 1937-4143. doi: 10.1109/MM.2022.3173405. Conference Name: IEEE Micro.

[69] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson. The VTR project: architecture and CAD for FPGAs from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 77–86, New York, NY, USA, Feb. 2012. Association for Computing Machinery. ISBN 978-1-4503-1155-7. doi: 10.1145/2145694.2145708. URL https://dl.acm.org/doi/10.1145/2145694.2145708.

[70] D. Wijerathne, Z. Li, M. Karunaratne, L.-S. Peh, and T. Mitra. Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*, 2022.

[71] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220 (4598):671–680, May 1983. doi: 10.1126/science.220.4598.671. URL https://www.science.org/doi/abs/10.1126/science.220.4598.671. Publisher: American Association for the Advancement of Science.

[72] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a Scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221, June 2012. doi: 10.1145/2228360.2228584. ISSN: 0738-100X.

[73] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr. 2016. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[74] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[75] F. Zaruba and L. Benini. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov. 2019. ISSN 1557-9999. doi: 10.1109/TVLSI.2019.2926114. URL https://ieeexplore.ieee.org/abstract/document/8777130. Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[76] Ibex RISC-V Core, May 2024. URL https://github.com/lowRISC/ibex. original-date: 2017-08-08T12:16:36Z.

[77] Sodor Processor Collection, May 2024. URL https://github.com/ucb-bar/riscv-sodor. original-date: 2013-07-17T22:10:42Z.

[78] Shuttle: A Rocket-based Superscalar In-order RISC-V Core, May 2024. URL https://github.com/ucb-bar/shuttle. original-date: 2023-05-17T02:30:07Z.

[79] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanovic. The Hwacha vector-fetch architecture manual, version 3.8. 1. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262*, 2015. URL https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2016/02/EECS-2015-262.pdf.

[80] J. Bispo and J. M. P. Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, July 2020. ISSN 2352-7110. doi: 10.1016/j.softx.2020.100565. URL https://www.sciencedirect.com/science/article/pii/S2352711019302122.

[81] P. Pinto, T. Carvalho, J. Bispo, and J. M. P. Cardoso. LARA as a language-independent aspect-oriented programming approach. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1623–1630, New York, NY, USA, Apr. 2017. Association for Computing Machinery. ISBN 978-1-4503-4486-9. doi: 10.1145/3019612.3019749. URL https://dl.acm.org/doi/10.1145/3019612.3019749.

[82] Berkeley Hardware Floating-Point Units, 2018. URL https://github.com/ucb-bar/berkeley-hardfloat. original-date: 2014-09-08T04:13:04Z.

[83] ARM. AMBA AXI and ACE Protocol Specification Version E, Feb. 2013. URL https://developer.arm.com/documentation/ihi0022/e.

[84] V. E. Beneš. *Mathematical theory of connecting networks and telephone traffic*. Academic press, 1965.

[85] A. D. Ruslanov and J. R. Johnson. An FPGA implementation of bene" permutation networks. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 245–245, Monterey California USA, Feb. 2004. ACM. ISBN 978-1-58113-829-0. doi: 10.1145/968280.968317. URL https://dl.acm.org/doi/10.1145/968280.968317.

[86] SiFive. SiFive TileLink Specication Version 1.8.1, Jan. 2020. URL https://starfivetech.com/uploads/tilelink_spec_1.8.1.pdf.

[87] L.-N. Pouchet and T. Yuki. PolyBench 4.2.1 (pre-release). Technical report, May 2016. URL https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1/blob/master/polybench.pdf.

[88] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, Oct. 2008. Association for Computing Machinery. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454128. URL https://dl.acm.org/doi/10.1145/1454115.1454128.

[89] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo. OpenCGRA: An Open-Source Unified Framework for Modeling, Testing, and Evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 381–388, Oct. 2020. doi: 10.1109/ICCD50377.2020.00070. URL https://ieeexplore.ieee.org/abstract/document/9283606?casa_token=5IrfJ9B2L04AAAAA:x6f0aVVp_efmBP1MoXyF2dgPYAr9XUI2TkPlBP00i8uiqDy-VTNOh3SyZF2YyU7EMOUmsDNw4FJk. ISSN: 2576-6996.

[90] H. Liew, D. Grubb, J. Wright, C. Schmidt, N. Krzysztofowicz, A. Izraelevitz, E. Wang, K. Asanović, J. Bachrach, and B. Nikolić. Hammer: a modular and reusable physical design flow tool: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 1335–1338, New York, NY, USA, Aug. 2022. Association for Computing Machinery. ISBN 978-1-4503-9142-9. doi: 10.1145/3489517.3530672. URL https://dl.acm.org/doi/10.1145/3489517.3530672.

[91] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, July 2016. ISSN 1879-2391. doi: 10.1016/j.mejo.2016.04.006. URL https://www.sciencedirect.com/science/article/pii/S002626921630026X.

[92] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, Feb. 2021. doi: 10.1109/CGO51591.2021.9370308. URL https://ieeexplore.ieee.org/abstract/document/9370308.

[93] G. P. Midões. UVE Streaming Engine: Hardware Implementation and Evaluation. Master's thesis, Instituto Superior Técnico, 2024.

Appendix A

Load MMU

The load memory management unit (LMMU), shown in Fig. A.1, consists of three main modules: the load request queue (LRQ), the load line buffer (LLB), and the load FIFO (LF). When the streaming engine generates a memory address, it is first registered in the LRQ, which groups addresses by memory row for coalescing. The LRQ issues requests for the required cache lines to the memory system, and the LLB receives and stores these lines as they arrive. The LF then extracts the data elements needed for each stream address from the LLB, assembles them in the correct stream order, and delivers the data to the processing unit. This architecture enables efficient memory access and optimized data flow to the processor.

More information about the LMMU can be found in the theses by Fernandes [36] and Midões [93].

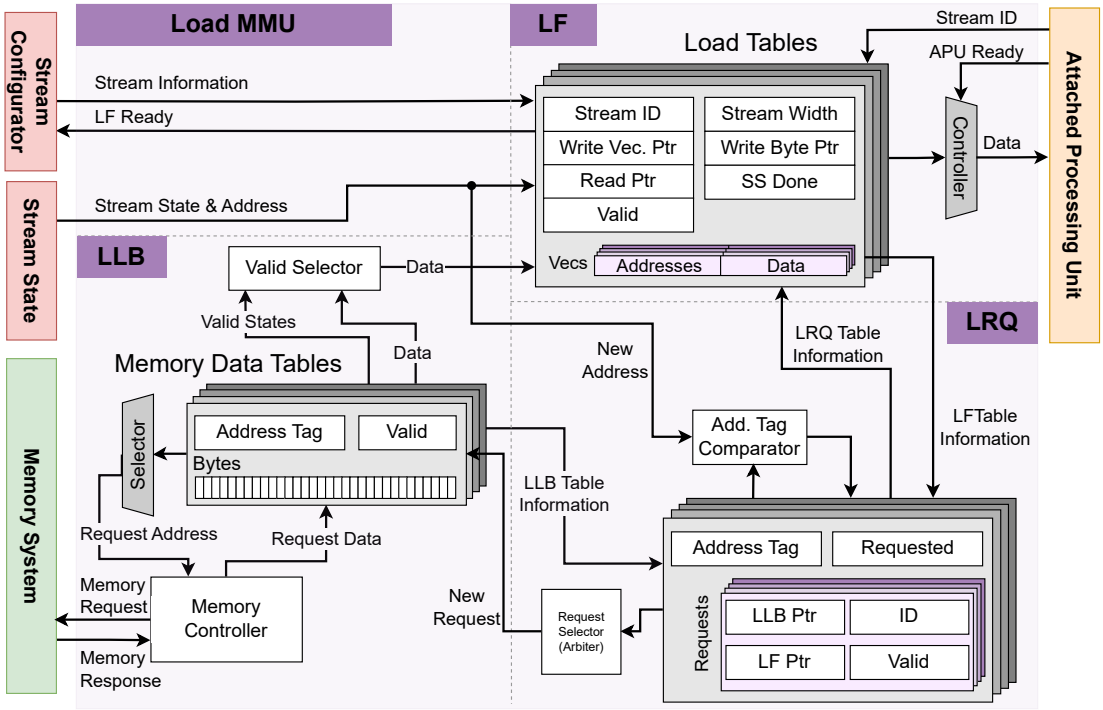


Figure A.1: Load MMU architecture [93].