ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE
DEPARTMENT OF ELECTRICAL, ELECTRONIC, AND INFORMATION ENGINEERING
"Guglielmo Marconi"

Master's Degree Programme in Electronic Engineering

Master's Thesis in Architectures for Artificial Intelligence

Integration of a Vector Processor into a Mesh-of-Tiles Architecture for Generative AI Acceleration

Supervisor:
Prof. Francesco Conti

Candidate:
Luca Balboni

Co-supervisors:
Yvan Tortorella (Chips.IT)
Alessandro Nadalini (UniBo)

V Session - Academic Year 2025/2026

Abstract

Le crescenti esigenze computazionali dei carichi di lavoro di Intelligenza Artificiale (AI) e High-Performance Computing (HPC) stanno guidando l'adozione di architetture di accelerazione sempre più scalabili e specializzate. In questo contesto, le architetture organizzate in tile interconnessi tramite Network-on-Chip (NoC) si sono affermate come una soluzione efficace per combinare parallelismo, modularità e scalabilità, consentendo di raggiungere elevate prestazioni e un'elevata efficienza energetica. Il presente lavoro, svolto in collaborazione tra l'Università di Bologna e la Fondazione Chips-IT, riguarda l'evoluzione di un tile appartenente a un'architettura open-source sviluppata nell'ambito della piattaforma PULP (Università di Bologna e ETH Zurich).

L'architettura considerata è costituita da una mesh bidimensionale di tile interconnessi tramite NoC. Ogni tile integra diversi moduli hardware, tra cui un processore RISC-V dedicato alle operazioni di controllo, una memoria locale tightly-coupled organizzata in più banchi interleaved per garantire accessi a bassa latenza, un modulo Direct Memory Access (DMA) per il trasferimento dei dati, un modulo di sincronizzazione tra tile, un'interfaccia di comunicazione verso la NoC e un acceleratore hardware dedicato alle moltiplicazioni matrice-matrice.

Il lavoro svolto ha riguardato un aggiornamento complessivo dell'architettura del tile e l'integrazione di diversi nuovi moduli hardware. In particolare, è stato introdotto un meccanismo di controllo memory-mapped per tutti gli acceleratori presenti nel tile, è stata integrata una nuova versione del processore di controllo ed è stato aggiunto un modulo dedicato alla gestione hardware degli eventi. Il contributo principale della tesi consiste nell'integrazione di un coprocessore vettoriale RISC-V dotato dell'estensione Zve32d, progettato per accelerare operazioni su vettori e migliorare l'efficienza nell'esecuzione di kernel numerici non strettamente matrice-matrice.

L'architettura risultante è stata implementata in tecnologia GlobalFoundries a 12 nm, con un target di frequenza di 800 MHz nelle condizioni di processo SSPG (0.72 V, -40°C). Il design è stato validato tramite un flusso completo di sintesi logica con Synopsys Design Compiler e place and route con Cadence Innovus. I risultati di sintesi mostrano che il tile nella configurazione baseline occupa una cell area totale di 1.22 mm², di cui 0.95 mm² occupati da memory macros e 0.29 mm² da logica standard-cell. L'integrazione del coprocessore vettoriale introduce un overhead di area pari a circa il 5.8% (0.07 mm²). Il place and route su un'area complessiva di 1.69 mm² (1300 μm × 1300 μm) raggiunge una frequenza operativa di 769 MHz con un'utilizzazione del silicio del 76%.

Le prestazioni del sistema sono state valutate tramite simulazioni RTL cycle-accurate utilizzando un insieme di microbenchmark in precisione FP16 rappresentativi di diverse tipologie di kernel numerici, tra cui moltiplicazioni matrice-matrice (MatMul), moltiplicazioni matrice-vettore (MatVec), dot product (DotP) e somme vettoriali (VecSum). Il throughput teorico di

picco dell'acceleratore dedicato alle operazioni matriciali è pari a 295.3 GFLOPS, valore raggiungibile in scenari di computazione GEMM-centric, mentre il coprocessore vettoriale raggiunge 12.3 GFLOPS.

Nei benchmark di picco, per MatMul il primo raggiunge 259.1 GFLOPS, mentre il coprocessore vettoriale si ferma a 11.4 GFLOPS. Nel caso MatVec, invece, il coprocessore vettoriale ottiene 10.8 GFLOPS, mostrando uno speedup di 1.33× rispetto all'acceleratore di moltiplicazioni matriciali (8.1 GFLOPS) grazie a un utilizzo più efficiente delle unità MAC. Nei kernel puramente vettoriali, come DotP e VecSum, il coprocessore vettoriale mostra prestazioni superiori: nel DotP raggiunge 3.5 GFLOPS contro 0.4 GFLOPS dell'acceleratore tensoriale, mentre nel VecSum ottiene 1.0 GFLOPS contro 0.6 GFLOPS.

Questi risultati evidenziano come il coprocessore vettoriale risulti particolarmente efficiente nei carichi di lavoro vector-centric, grazie a un migliore utilizzo delle risorse hardware disponibili.

L'efficienza energetica è stata stimata tramite vector-based power analysis basata sull'attività di switching ottenuta da simulazioni gate-level sulla netlist post-layout, considerando il corner TT a 25°C con tensione di alimentazione nominale (0.8 V). I risultati mostrano che, per matrici fortemente rettangolari, il coprocessore vettoriale può raggiungere un'efficienza energetica superiore rispetto all'acceleratore per moltiplicazioni matrice-matrice anche per operazioni MatMul. Per matrici più quadrate e di dimensioni elevate, l'acceleratore tensoriale rimane la soluzione più efficiente. Nei carichi di lavoro puramente vettoriali, come DotP e VecSum, il coprocessore vettoriale mostra invece un'efficienza energetica sistematicamente superiore su tutte le dimensioni testate, con un vantaggio che aumenta al crescere della lunghezza dei vettori.

Questo approccio apre la strada all'integrazione futura di ulteriori unità di accelerazione dedicate ad altri domini computazionali, consentendo di estendere la piattaforma oltre i soli carichi di lavoro di Intelligenza Artificiale. In prospettiva, tale flessibilità architetturale permette di progettare sistemi in cui la composizione eterogenea dei tile all'interno della mesh consente di adattare l'architettura alle caratteristiche specifiche dei diversi workload target, migliorando l'efficienza complessiva del sistema.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rapid growth of Artificial Intelligence (AI) and High-Performance Computing (HPC) workloads is driving the need for increasingly efficient and scalable computing architectures. Modern applications such as deep learning, scientific computing, and large-scale data analytics require extremely high computational throughput while maintaining strict energy efficiency constraints. As a result, heterogeneous architectures combining programmable processors with domain-specific accelerators have become a key design paradigm for modern computing systems.

This thesis is developed in the context of a joint research activity between the University of Bologna and the Chips-IT Foundation, within the broader ecosystem of the PULP (Parallel Ultra-Low-Power) platform. The PULP platform is an open-source hardware and software ecosystem jointly developed by the University of Bologna and ETH Zurich. Within this framework, MAGIA (Mesh Architecture for Generative Intelligence Acceleration) has been proposed as a scalable template for mesh-based accelerator architectures targeting emerging AI workloads.

MAGIA is organized as a two-dimensional mesh of compute tiles interconnected through a Network-on-Chip (NoC). Each tile integrates a lightweight RISC-V processor responsible for control operations, a tightly-coupled multi-banked local memory, a DMA engine for efficient data movement, a network interface enabling communication across the Network-on-Chip, and synchronization mechanisms for coordination among tiles, together with specialized hardware accelerators. In the original architecture, the tile includes a dedicated accelerator for matrix-matrix multiplications, which efficiently targets matrix-centric workloads.

While tensor accelerators are highly efficient for GEMM-based computations, many AI and HPC workloads also include operations that are better expressed as vector computations. Examples include reductions, matrix–vector multiplications, vector transformations, and other linear algebra kernels that are not efficiently supported by a matrix multiplication engine. For this reason, the integration of vector processing capabilities represents an important step toward improving the flexibility and overall computational efficiency of the architecture. Moreover, extending the tile with additional compute engines opens the possibility of designing heterogeneous tile configurations within the mesh, where different tiles may integrate different types of

accelerator depending on the target workload. Such heterogeneity can further improve resource utilization and enable the architecture to efficiently support a wider range of computational patterns.

The main objective of this thesis is therefore to extend the original MAGIA tile architecture through the integration of a RISC-V vector coprocessor from the PULP platform, namely the Spatz Core Complex (Spatz CC). Its integration allows vector operations to be executed concurrently with the operation of the matrix multiplication engine already present in the tile, enabling the architecture to efficiently support matrix-centric and vector-centric workloads.

In order to enable the integration of the vector coprocessor, a set of architectural updates to the original MAGIA tile has been introduced. These modifications not only support the integration of the vector unit, but also address several limitations and minor issues present in the previous tile implementation. In particular, the updates include the adoption of a unified memory-mapped control interface for the hardware accelerators, improvements to the event handling infrastructure through the integration of an event unit, and the replacement of the original control core with an alternative RISC-V processor.

The updated architecture is validated through a complete hardware design flow targeting a 12 nm GlobalFoundries technology. The design is synthesized and physically implemented using Electronic Design Automation (EDA) tools, enabling the evaluation of area, timing, and power characteristics. Furthermore, the computational performance and energy efficiency of the system are evaluated through cycle-accurate simulations and a set of representative microbenchmarks.

The thesis is organized as follows.

- **Chapter 2** provides the architectural background required to understand the contributions of this work. Introduces the MAGIA (Mesh Architecture for Generative Intelligence Acceleration) platform and its organization as a mesh of compute tiles interconnected through a Network-on-Chip. The chapter describes the internal structure of the MAGIA tile and its main components, including the control processor, the local memory organization, and the hardware accelerators. Then it discusses different accelerator control paradigms, comparing instruction-based interfaces such as the eXtension Interface (XIF) with memory-mapped programming models. Finally, the chapter presents the architecture of the vector coprocessor integrated in this work.

- **Chapter 3** describes the architectural modifications introduced to enable the integration of the vector coprocessor and improve the overall design of the tile. The chapter first presents the redesign of the tile address space and the introduction of memory-mapped control interfaces for the hardware accelerators, including the tensor accelerator, the DMA engine, and the synchronization module. It then discusses the integration of the event unit and its hardware and software interfaces. The chapter also describes the replacement of the original control processor with an alternative RISC-V core and concludes with the

hardware and software integration of the vector coprocessor within the tile architecture.

- **Chapter 4** presents the physical design of the updated architecture. Introduces the adopted physical design methodology and describes the synthesis and the place and route flows used to implement the design. The chapter reports the post-synthesis and post-layout results, including area occupation and timing closure.
- **Chapter 5** evaluates the performance and energy efficiency of the proposed architecture. The chapter first introduces the benchmark suite used to characterize the system and describes the adopted power analysis methodology. Then it presents the experimental results, comparing the computational performance and energy efficiency of the different compute engines for representative numerical kernels.
- **Chapter 6** summarizes the main contributions of this thesis and discusses possible directions for future work.

Chapter 2

Background

2.1 MAGIA – Mesh Architecture for Generative Intelligence Acceleration

MAGIA is a tile-based scalable accelerator designed for AI workloads. Its architectural philosophy is based on the spatial replication of homogeneous compute tiles interconnected through a two-dimensional (2D) Network-on-Chip (NoC), allowing predictable scalability in both performance and physical implementation [1].

2.1.1 MAGIA Architecture

The system is organized as a regular 2D mesh (XY grid) of tile (Fig. 2.1). Each tile represents a self-contained compute node that integrates computation, local memory, communication interfaces, and synchronization mechanisms. Tiles explicitly orchestrate the data movement and barrier phases, following a structured execution model in which computation and communication are clearly separated. This explicit management of data transfers enables software controlled overlap between computation and communication, which is essential for sustaining high utilization of on-chip accelerators.

Inter-tile communication occurs through an AXI4-based interface [2] connected to the NoC. In the reference implementation, MAGIA adopts FlooNoC, an open source 2D XY mesh interconnect that features AXI4-compatible Network Interfaces (NIs) that translate AXI transactions into network packets. The NoC uses 32-bit physical links with a single physical channel per direction and dimension-ordered routing across the mesh. The choice of a single physical channel per direction simplifies the router microarchitecture (Fig. 2.2) and reduces buffering requirements.

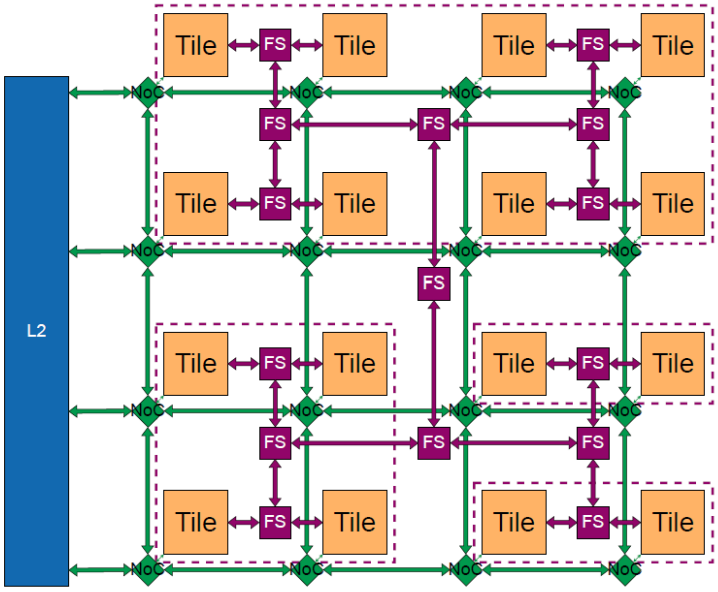

Figure 2.1: Architecture of a 4×4 configuration of MAGIA

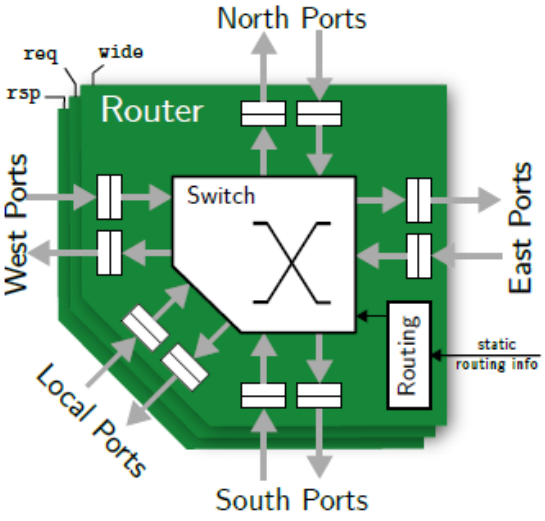

Figure 2.2: Router microarchitecture

Virtual channels are not required, as synchronization traffic is offloaded to a dedicated network, preventing interference between bulk data transfers and barrier control messages. This design choice allows the NoC to be optimized for sustained bandwidth rather than low-latency control signaling.

MAGIA follows the Bulk Synchronous Parallel (BSP) execution model, in which computation phases alternate with global synchronization barriers. Within each superstep, tiles perform local computation and communication, followed by a barrier that ensures global progress before entering the next phase. Synchronization is implemented through FractalSync: Lightweight Scalable Global Synchronization of Massive Bulk Synchronous Parallel AI Accelerators, a ded-

icated H-tree synchronization network layered on top of the mesh [1]. FractalSync implements a recursive divide-and-conquer synchronization scheme that naturally maps to an H-tree topology, enabling logarithmic scaling of barrier latency with respect to the number of tiles. By separating bulk data movement (handled by the NoC) from synchronization (handled by FractalSync), MAGIA achieves a clean architectural decoupling between throughput-oriented communication and latency-sensitive barrier signaling.

2.1.2 MAGIA Tile

At the heart of MAGIA lies the compute tile, which integrates a matrix multiplication accelerator, a DMA engine, a multi-banked L1 scratchpad memory, and a lightweight control core. The tile represents the fundamental unit of parallelism in the architecture and is designed to operate as an autonomous compute node within the larger mesh. Its microarchitecture balances specialization and programmability: computationally intensive kernels are offloaded to dedicated hardware accelerators, while a general-purpose RISC-V core orchestrates execution, data movement and synchronization.

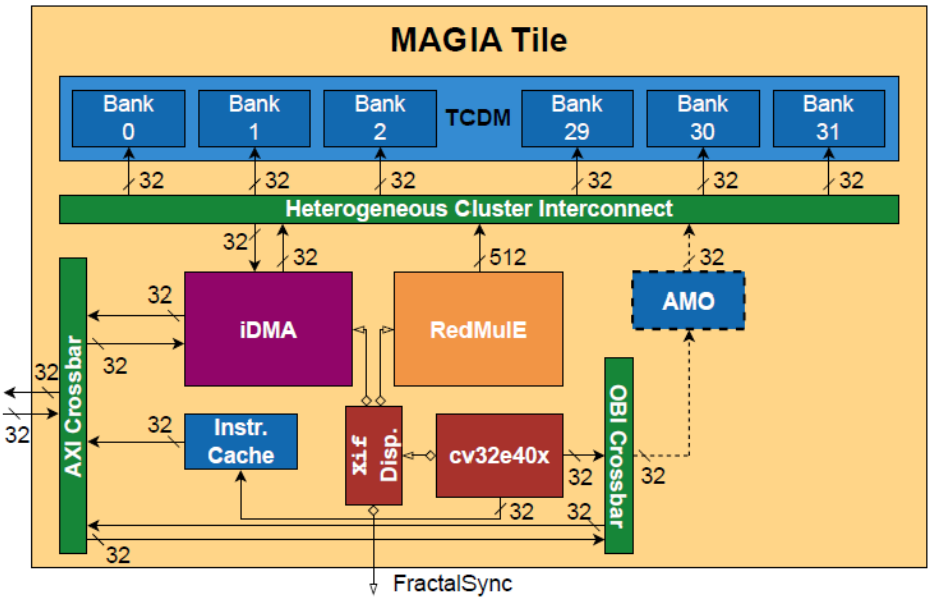

Figure 2.3: MAGIA tile block diagram

L1 Scratchpad Memory (TCDM)

The tile features a 1 MiB L1 Scratchpad Memory (SPM), implemented as a Tightly-Coupled Data Memory (TCDM) composed of 32 interleaved memory banks. Each bank is 32 KiB in size and 32-bit wide. The scratchpad organization ensures single-cycle access latency under non-conflicting conditions and enables high aggregate bandwidth through bank interleaving. Word-level interleaving distributes consecutive memory addresses across different banks, al-

lowing simultaneous access to independent words. This organization significantly reduces structural hazards and enables multiple masters to perform concurrent memory operations, provided that bank conflicts are avoided. The TCDM is accessed through a single-cycle latency Heterogeneous Cluster Interconnect (HCI), which provides arbitration among internal masters (core, DMA, accelerator) [3] [4]. The HCI adopts a distributed arbitration scheme in which contention is resolved locally at the level of each memory bank rather than through a centralized global arbiter. After address decoding determines the target bank, each request is forwarded to the corresponding bank-specific arbiter. If multiple masters simultaneously access the same bank, the local arbiter selects one request, typically using a fair round-robin policy, while temporarily stalling the others. Crucially, accesses targeting different banks proceed independently and without mutual interference. This bank-level arbitration ensures that conflicts remain localized and do not propagate across the entire interconnect. As a result, when memory accesses are evenly distributed across banks, the HCI sustains high aggregate bandwidth while preserving single-cycle access latency under non-conflicting conditions. The arbitration logic is intentionally lightweight and largely combinational, enabling predictable timing behavior. Such a distributed structure is particularly well suited for accelerator-centric workloads, where heterogeneous masters generate memory traffic with diverse access patterns and bandwidth requirements.

Control Core – CV32E40X

Each tile is controlled by an OpenHW Group CV32E40X RISC-V core [5], coupled with a 16 KiB instruction cache. Within the MAGIA tile, the core does not perform heavy numerical computation directly; instead, it acts as a control processor responsible for coordinating accelerator execution, orchestrating synchronization phases, and explicitly managing data movement across memory levels. The core operates as a master of the internal OBI-based crossbar [6], which connects it to both the local L1 TCDM and the tile's external memory interface. All load and store instructions generated by the core are issued through the OBI interconnect. When the target address falls within the local L1 address space, the request is forwarded to the TCDM banks via the HCI, conversely, when the address maps to the global memory region, the request is routed toward the tile's AXI interface and injected into the NoC, allowing the core to access remote tiles or the shared L2 memory. The core is coupled with a private 16 KiB instruction cache, which supplies instructions during normal execution. Upon an instruction cache miss, a refill request is generated and forwarded through the tile's memory interface to L2 memory.

To support accelerator control, the ISA is extended through the OpenHW eXtension Interface (XIF) [7], which allows custom instructions to be integrated without modifying the processor microarchitecture. The XIF interface enables tightly-coupled accelerator integration by allowing the core to offload recognized custom instructions directly from the decode stage. In the MAGIA tile, the XIF mechanism is used to control and interact with multiple hardware units, including the iDMA engine, the FractalSync synchronization module, the RedMulE matrix mul-

tiplication accelerator and the floating-point unit (FPU). Custom instructions are decoded and dispatched to the appropriate hardware block, enabling the core to program data transfers, trigger synchronization operations, configure and start matrix multiplication kernels, or invoke floating-point computations.

DMA - Direct memory access

To support efficient data movement, the tile integrates a high-performance and energy-efficient modular DMA engine architecture, namely iDMA, an open-source and highly configurable data movement engine. The iDMA adopts a modular and layered design that separates the programming interface from the low-level data transport logic (Fig. 2.4).

At its core, the backend of iDMA implements the AXI-based data transfer mechanism, generating burst transactions and handling responses over the on-chip interconnect. In the MAGIA tile, the DMA connects the local L1 TCDM to the global L2 memory or to remote tiles. The instantiated configuration includes two independent 32-bit transfer channels:

- One dedicated to inbound traffic (external → L1);
- One dedicated to outbound traffic (L1 → external).

This dual-channel organization enables full-duplex communication and supports overlapping read and write transactions. Once programmed by the control core, the DMA operates autonomously, allowing communication to proceed in parallel with computation and enabling efficient double-buffering schemes.

Figure 2.4: iDMA block diagram

RedMulE - Reduced-Precision Matrix Multiplication Engine

Matrix multiplication kernels are offloaded to RedMulE (Fig. 2.5), an open-source accelerator designed for efficient General Matrix-Matrix Multiplication (GEMM) and General Matrix–Matrix Operations (GEMM-Ops). RedMulE is the computational backbone of the MAGIA tile. RedMulE istance in MAGIA Tile implements a 24 × 8 semi-systolic array of Computing Elements (CEs) (Fig. 2.6), for a total of 192 processing units. The semi-systolic organization allows the reuse of operands across rows and columns, reducing memory-bandwidth pressure and increasing arithmetic intensity [8].

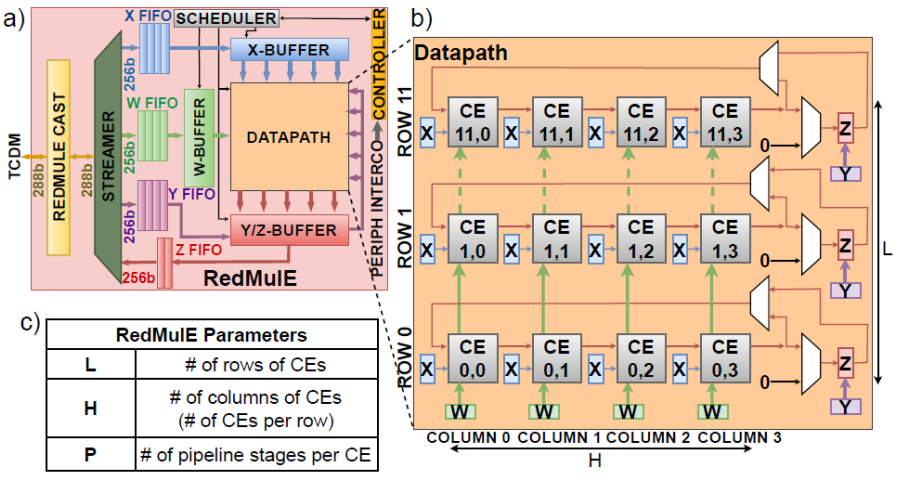

Figure 2.5: Example of RedMulE internal architecture

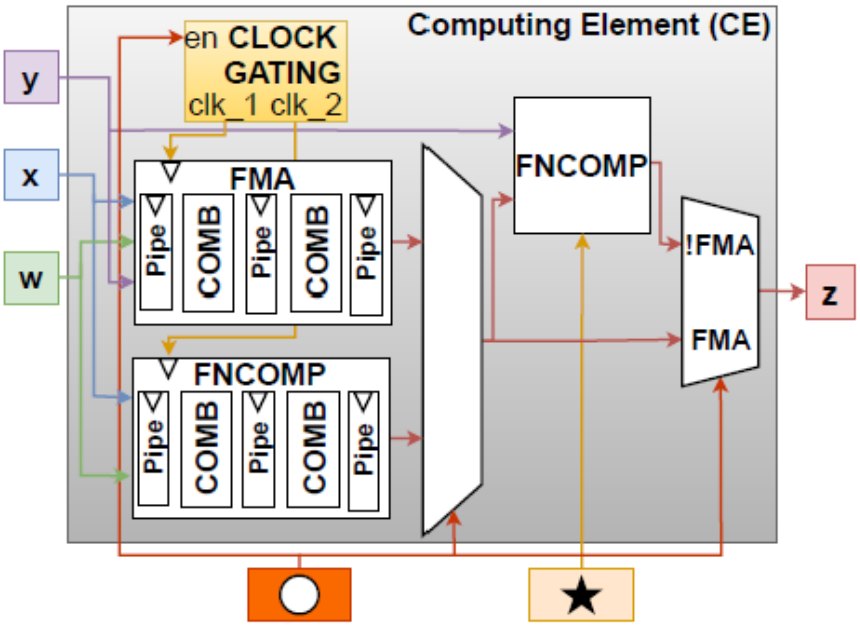

Figure 2.6: Microarchitecture of RedMulE's CE with extensions for GEMM-Ops support

The architecture follows an outer-product data flow in which matrix **X** elements are broadcast along rows, matrix **W** elements are broadcast along columns and partial sums are accumulated locally within each row. This structure increases input reuse and minimizes the number of memory accesses per operation, enabling peak throughput close to full array utilization. Unlike fully systolic arrays, RedMulE adopts a semi-systolic design in which columns forward partial results horizontally, while accumulation feedback is handled locally within rows. The resulting data flow is optimized for tiled GEMM execution, providing a trade-off between hardware complexity, routing cost, and scalability. RedMulE supports mixed-precision floating-point com-

putation with FP16 and hybrid FP8 formats, while using internal FP16 accumulation. Hybrid FP8 storage reduces the memory footprint and increases effective memory bandwidth; inputs can be stored in FP8 and up-casted internally to FP16 for accumulation to preserve numerical stability while reducing memory traffic. Beyond classical GEMM ($\mathbf{Z} = \mathbf{XW}+\mathbf{Y}$), RedMulE supports GEMM-Ops, where the multiplication and/or accumulation operators can be replaced by min/max. This generalization enables acceleration of graph algorithms, dynamic-programming kernels, shortest-path problems, and other non-linear algebraic operators. RedMulE is tightly coupled to the TCDM via a 512 bit wide interface, providing high internal bandwidth between the accelerator and L1 memory.

2.2 Accelerator Control Interface: XIF vs Memory-Mapped Programming

Within the PULP ecosystem, accelerators are typically integrated using one of two paradigms. The first paradigm is the XIF-based integration, where accelerator control is performed through custom RISC-V ISA extensions. In this approach, dedicated instruction encodings are reserved for accelerator operations, allowing hardware modules to be invoked as if they were native execution units of the processor. The second paradigm is the memory-mapped register approach, where accelerators expose a set of configuration and status registers in the processor's address space. In this model, software programs accelerators by performing standard load and store operations to dedicated control registers, and execution is triggered by writing to control registers. The accelerator is therefore treated as a peripheral connected to the system interconnect. As previously discussed, in the original MAGIA Tile architecture, accelerator control is implemented through the eXtension Interface [7], which provides a standardized mechanism to extend the instruction set of the CV32E40X core without modifying its internal microarchitecture, allowing for a tight integration of hardware accelerators at the ISA level. Through the XIF request channel, the core forwards the instruction information to the external unit. Depending on the interface configuration, this may include the instruction word, operand values, and destination register details. From the processor's perspective, the external module behaves as an additional execution unit integrated into the pipeline. The accelerator performs the requested operation and eventually returns the result and completion information via the XIF response channel. The core then writes the result back to the register file, preserving the architectural behavior expected from a standard in-core instruction. The XIF protocol relies on a ready/valid handshake mechanism to guarantee proper synchronization between the core and the external unit. If the accelerator cannot accept a new request or if the result is not yet available, the processor pipeline may stall. This mechanism enables the integration of multi-cycle accelerators while maintaining precise exceptions and in-order execution semantics. However, practical experience with CV32E40X IP highlighted functional bugs and stability concerns associated with XIF-based control. These

issues motivated the adoption of an alternative control strategy. In this work, the CV32E40X core is therefore replaced by the CV32E40P processor, which provides a more mature and stable architecture for accelerator integration. The XIF mechanism is replaced by a memory-mapped programming model. This approach separates accelerator control from the processor instruction pipeline.

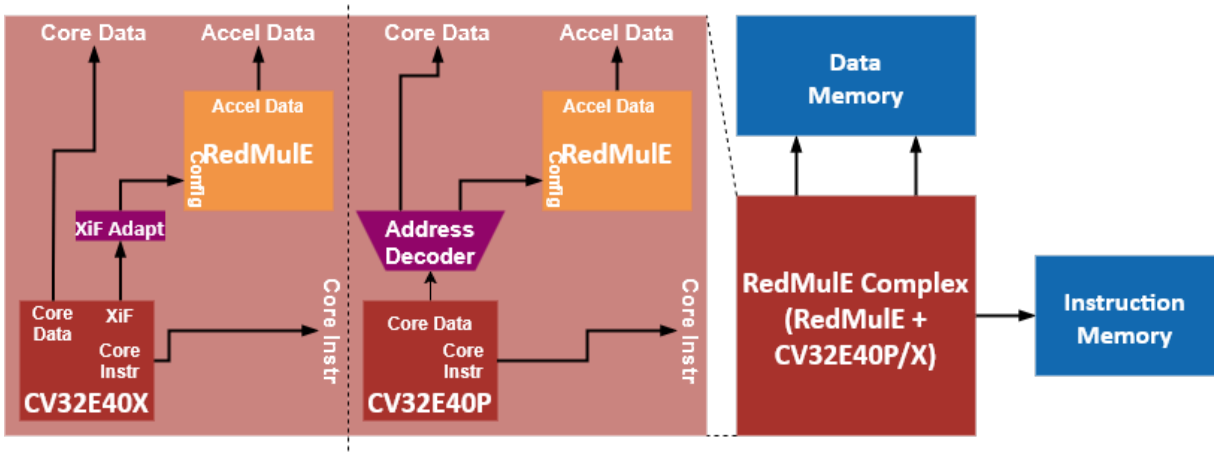

Figure 2.7: RedMulE supports both approaches

2.3 CV32E40P (RI5CY)

The CV32E40P core has been extensively validated within the PULP Platform ecosystem, particularly in the PULPissimo SoC, providing better compatibility with PULP-specific peripherals and programming models. CV32E40P, with the appropriate PULP extensions enabled, supports the p.elw (event load word) instruction, which enables efficient, low-latency event-based synchronization.

The CV32E40P core, also known as RI5CY, is a 4-stage, in-order, 32-bit RISC-V processor core. The core implements the RV32IMFCXpulpv2 instruction set architecture and is specifically designed for energy-efficient embedded applications.

The CV32E40P, shown in Figure 2.8, features a 4-stage pipeline consisting of: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX) and Write Back (WB). The FPU is integrated directly into the Execute stage, unlike the CV32E40X which uses an external FPU connected via the eXtension Interface (XIF).

The architectural difference in FPU integration represents one of the most significant distinctions between the two cores. In the CV32E40X core, floating-point operations are handled through the XIF interface, which connects external coprocessors but introduces additional latency and complexity in the instruction flow.

The CV32E40P FPU supports single-precision floating-point operations (IEEE 754) including addition, subtraction, multiplication, division, and square root operations. In the MAGIA tile, the FPU is configured with FPU = 1, ZFINX = 1 for dedicated floating-point registers, and

FP_DIVSQRT = 1 to enable division and square root operations.

The CV32E40P core implements the XpulpV2 instruction set extension, which includes hardware loops, post-incrementing load/store operations, extended ALU operations (multiply-accumulate, min/max, bit manipulation), and the Event Load Word (p.elw) instruction.

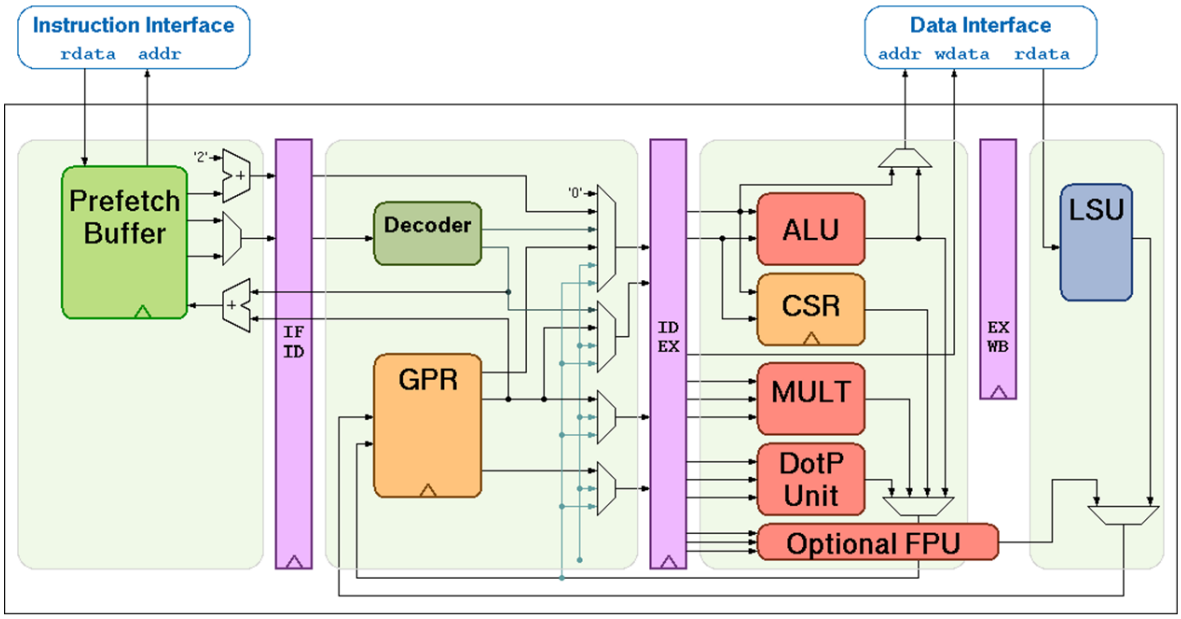

Figure 2.8: RI5CY Block Diagram

2.4 Spatz CC

RedMulE is a highly specialized accelerator tailored for dense matrix–matrix multiplication and GEMM-like workloads. Its microarchitecture is based on a semi-systolic dataflow, which enables high compute utilization and efficient operand reuse when executing structured linear algebra kernels. However, while extremely efficient for GEMM-style computations, a semi-systolic matrix engine does not naturally generalize to other classes of workloads that cannot be expressed as matrix–matrix multiplications. Many AI and scientific applications require efficient execution of vector reductions, convolutions with varying kernel sizes, dot products, Fast Fourier Transforms (FFTs), sparse operations, and element-wise transformations. In such cases, a programmable vector architecture provides greater flexibility and improved adaptability across a wider range of computational kernels. To address this limitation and increase the flexibility of the MAGIA tile, this work integrates the Spatz Core Complex (Spatz CC). The Spatz CC combines a compact vector processing unit, Spatz, with a lightweight scalar core, Snitch [9] [10] (Fig. 2.9).

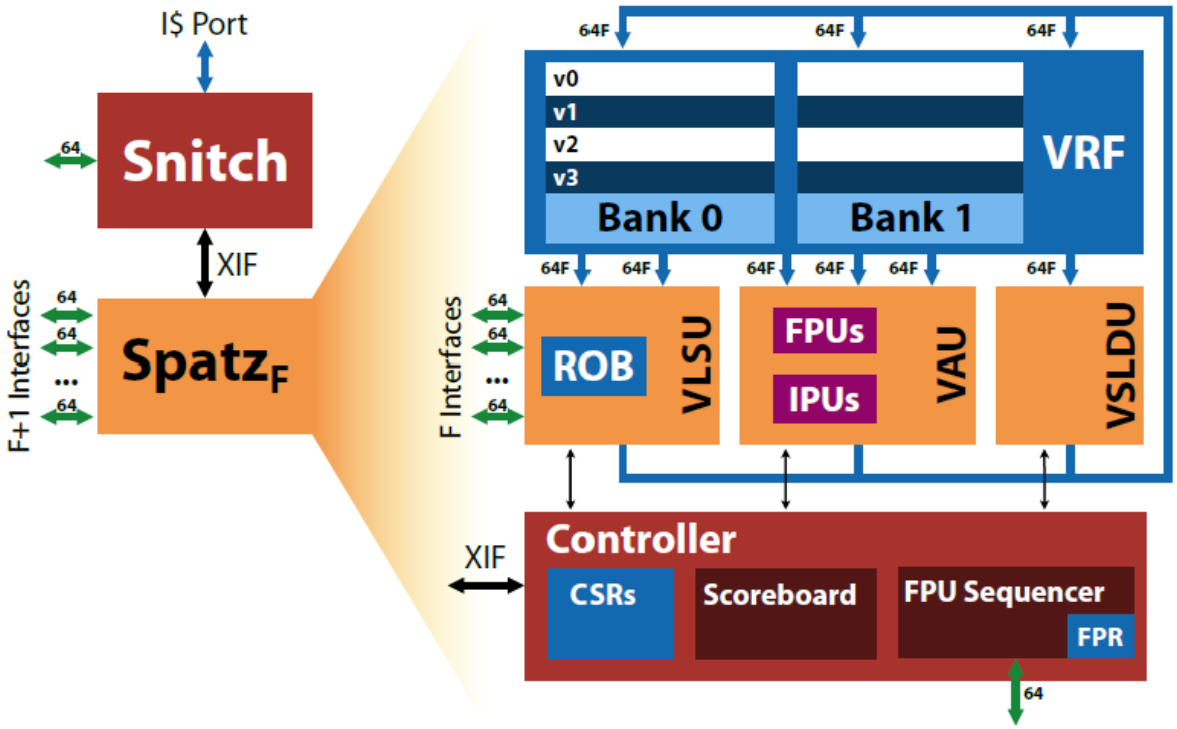

Figure 2.9: Spatz CC composition

Spatz is an open-source, compact, and parametric Vector Processing Unit (VPU) based on the RISC-V vector extension Zve32d [9] [10]. The architecture is explicitly designed to maximize Data-Level Parallelism (DLP) while avoiding complex and energy-hungry Instruction-Level Parallelism (ILP) mechanisms typical of large out-of-order processors. Instead of relying on deep pipelines, speculative execution, or complex scheduling logic, Spatz dedicates the majority of its hardware resources to vector execution units and vector storage structures.

The vector unit works in conjunction with Snitch [11]. Snitch is a tiny pseudo dual-issue processor optimized for area and energy efficient execution of floating-point intensive workloads. Rather than relying on superscalar issue width or complex instruction level parallelism techniques, Snitch follows a minimalist design philosophy and delegates most computational work to tightly coupled execution units such as floating-point or vector accelerators. This approach improves the compute-to-control ratio while keeping the scalar core extremely lightweight. Within the Spatz Core Complex, the scalar core is primarily responsible for orchestration, loop control, and configuration of vector operations, while the Vector Processing Unit (VPU) executes the bulk of arithmetic computation through high-throughput data-parallel execution Spatz implements a subset of RVV 1.0 ISA and provides 32 architectural vector registers. The effective vector length is determined by the implementation parameter VLENB. Through the Vector Length Multiplier (LMUL), multiple architectural registers can be grouped at runtime to form longer logical vectors, trading register availability for increased data reuse and improved computational granularity.

A key architectural feature of Spatz is its latch-based, multi-banked Standard Cell Memory (SCM) Vector Register File (VRF), which acts as a private L0 memory tightly coupled to the vector unit. By providing a high-bandwidth, low-latency storage structure close to the compute units, the VRF increases data reuse and significantly reduces pressure on the L1 Scratchpad Memory interconnect. This design choice alleviates both instruction-side and data-side aspects of the Von Neumann bottleneck: SIMD execution reduces instruction fetch pressure, while the VRF minimizes repeated accesses to shared memory. The Vector Arithmetic Unit (VAU) supports multi-precision floating-point computation, including half-precision (FP16), single-precision (FP32), and double-precision (FP64) formats. Double-precision operations are supported through 64-bit datapaths, while FP32 and FP16 operations exploit narrower arithmetic lanes to increase parallel throughput when lower precision is sufficient. This multi-precision capability allows the architecture to span a wide application domain, from high-accuracy computing kernels requiring FP64 arithmetic to AI inference and training workloads that benefit from reduced precision formats. From a hardware configuration perspective, the vector unit can be instantiated with different datapath widths. While the standard configuration includes 64-bit lanes to support FP64 execution, the architecture can also be synthesized with 32-bit datapaths when double precision is not required.

Chapter 3

MAGIA Tile Architectural Updates

This chapter presents the architectural evolution of the MAGIA tile. The primary contribution of this work is the integration of the Spatz Core Complex (Spatz CC), which extends the original matrix-centric tile into a heterogeneous compute node by introducing support for vector processing. In parallel to the integration of Spatz CC, the tile control model has been redesigned to adopt a unified memory-mapped architecture for all accelerators. In the updated design, accelerators are exposed as memory-mapped peripherals and are programmed exclusively through standard load and store operations. This transition removes the dependency on custom ISA extensions and establishes a uniform, ISA-independent programming model across the tile. The adoption of a memory-mapped control scheme is accompanied by the substitution of the original CV32E40X core with the CV32E40P variant. The CV32E40X core natively supports the OpenHW XIF interface, which had previously been employed to control hardware accelerators through custom instruction extensions. However, during system-level validation, the XIF-based integration exhibited stability issues and revealed several corner case issues. In addition, the CV32E40X core is no longer actively maintained within the PULP ecosystem, raising concerns about its long-term viability and support. Consequently, the dependency on XIF-based control was eliminated and the design was migrated to the CV32E40P core, which does not provide eXtension interface support.

Finally, an Event Unit has been introduced to improve synchronization robustness across multiple concurrent accelerators. During early validation, race conditions were observed when multiple accelerators signaled completion simultaneously or when the core was executing non-interruptible critical sections. The Event Unit provides centralized event buffering, storing completion notifications in hardware registers and decoupling event generation from event handling. This mechanism improves robustness and enables safe orchestration of concurrent operations involving Spatz CC, RedMulE, and iDMA.

3.1 Memory Mapping

3.1.1 Address Space Modification

The new architecture reserves the lower 64 KB of the address space as a structured peripheral control region. The complete organization of the addressable space is reported in Table 3.1.

Address Range	Size	Region	Description
0x0000_0000 – 0x0000_00FF	256 B	Guard	Null pointer protection
0x0000_0100 – 0x0000_01FF	256 B	RedMulE	Matrix multiplication control
0x0000_0200 – 0x0000_05FF	1 KB	iDMA	DMA engine control
0x0000_0600 – 0x0000_06FF	256 B	FractalSync	Hierarchical synchronization
0x0000_0700 – 0x0000_16FF	4 KB	Event Unit	Event management
0x0000_1700 – 0x0000_17FF	256 B	Spatz Control	Spatz CC config
0x0000_1800 – 0x0000_FFFF	~58 KB	Reserved	Future peripheral expansion
0x0001_0000 – 0x0001_FFFF	64 KB	Stack	Thread stack memory
0x0002_0000 – 0x000F_FFFF	~896 KB	L1 SPM	Tile-local scratchpad
0xC000_0000 – 0xFFFF_FFFF	~1 GB	L2	Shared memory

Table 3.1: Updated MAGIA tile memory map

The peripheral control region, which spans 0x0000_0100 to 0x0000_17FF, contains memory-mapped control interfaces. RedMulE, iDMA, and FractalSync occupy the lower portion (0x0000_0100 to 0x0000_06FF). Each peripheral is allocated a contiguous region sized according to its configuration requirements: RedMulE and FractalSync occupy 256 bytes each, while iDMA requires 1 KB to support its dual-channel configuration. Above these, the Event Unit (4 KB at 0x0000_0700) provides unified interrupt management, and Spatz Control (256 B at 0x0000_1700) handles vector processor configuration.

RedMulE is mapped at 0x0000_0100, iDMA at 0x0000_0200, FractalSync at 0x0000_0600, Event Unit at 0x0000_0700, and Spatz Control at 0x0000_1700, while the remaining space up to 0x0000_FFFF is reserved for future extensions.

Above the peripheral region, the software stack is mapped at 0x0001_0000 and occupies 64 KB. The L1 scratchpad memory spans from 0x0002_0000 to 0x000F_FFFF, providing approximately 896 KB of tile-local storage.

In multi-tile configurations, each tile's L1 region is offset by 1 MB according to the tile identifier, preventing address conflicts. For example, tile 0 occupies 0x0002_0000–0x000F_FFFF, while tile 1 is mapped at 0x0012_0000–0x001F_FFFF.

The shared L2 memory, reachable through the NoC, begins at 0xC000_0000 and extends to 0xFFFF_FFFF.

3.1.2 RedMulE Memory-Mapping

The updated design enables RedMulE in HWPE-CTRL mode by disabling the XIF extension parameter (X_EXT) in the accelerator configuration. In this mode, the accelerator exposes its

control registers through a standardized HWPE-CTRL slave interface, which implements a simple request-response protocol for register access. This interface provides the same register file structure as the XIF-based design but makes it accessible through memory-mapped transactions rather than custom instructions.

The hardware integration connects RedMulE's HWPE-CTRL slave port to the tile's OBI crossbar. A dedicated routing rule in the OBI crossbar ensures that all memory transactions targeting the address range 0x0000_0100–0x0000_01FF are directed exclusively to RedMulE's control port. This address-based routing allows the processor to access RedMulE registers using standard load and store instructions with the appropriate addresses, and the interconnect automatically directing these transactions to the correct destination.

Since OBI and HWPE-CTRL use different signaling protocols, a protocol bridge module (obi2hwpe_ctrl) performs the necessary conversion. This bridge translates the OBI address and data phases into HWPE-CTRL request and grant signals, handling differences in transaction semantics, timing, and flow control. The register space is organized into two functional groups: control registers for job management and synchronization (offsets 0x00–0x14), and configuration registers for specifying matrix operation parameters (offsets 0x40–0x54). Table 3.2 summarizes the complete register map.

Offset	Register	Access	Function
0x00	TRIGGER	W	Initiate computation
0x04	ACQUIRE	R	Acquire accelerator (job ID or -1)
0x08	EVT_ENABLE	W	Enable event signal generation
0x0C	STATUS	R	Current operation status (0 = idle)
0x10	RUNNING_JOB	R	Active job ID or -1
0x14	SOFT_CLEAR	W	Reset control state
0x40	X_PTR	W	Input matrix X address
0x44	W_PTR	W	Weight matrix W address
0x48	Z_PTR	W	Output matrix Z address
0x4C	MCFG0	W	Matrix configuration: M and K dimensions
0x50	MCFG1	W	Matrix configuration: N dimension
0x54	ARITH	W	Operation type and numeric format

Table 3.2: RedMulE memory-mapped register map

The control registers implement a lock-based access protocol to prevent conflicts in multi-threaded environments. The ACQUIRE register serves as an atomic lock mechanism: reading it returns zero if the accelerator is available and simultaneously locks it for exclusive use, or returns -1 if already locked by another thread. Once acquired, the TRIGGER register initiates computation and the STATUS register allows software to poll for completion. The STATUS register returns zero when the accelerator is idle, providing a simple synchronization primitive. The RUNNING_JOB register identifies the currently executing job (when STATUS is non-zero), while SOFT_CLEAR allows software to reset the control state machine.

The configuration registers specify the complete matrix multiplication operation. The three pointer registers (X_PTR, W_PTR, Z_PTR) contain memory addresses for the input matrix, the weight matrix, and the output matrix, respectively. The matrix dimensions are split into two configuration registers: MCFG0 encodes both M and K dimensions in a single 32-bit word, while MCFG1 contains the N dimension. The ARITH register specifies the type of operation and the numeric format.

For software control, a set of helper functions from RedMulE's original HAL [12] simplifies the accelerator interaction. These functions provide memory-mapped register access through standard C interfaces, replacing the original XIF custom instructions while maintaining the same programming model. The interface includes functions for lock acquisition (hwpe_acquire_job()), accelerator configuration (redmule_cfg()), execution triggering (hwpe_trigger_job ()) and completion polling (hwpe_get_status()). The typical usage sequence follows four phases: acquire exclusive access by polling until ACQUIRE returns zero, configure all matrix parameters through redmule_cfg() which handles register encoding internally, trigger computation by writing to the TRIGGER register, and wait for completion by polling the STATUS register until it returns zero (idle state).

3.1.3 iDMA Memory-Mapping

Memory-mapped DMA control is implemented through idma_ctrl_mm, a dedicated controller module. This module builds on the previous module by Victor Isachi: idma_ctrl.

The iDMA control interface occupies 1 KB of address space (0x0000_0200–0x0000_05FF), the largest allocation among the tile's accelerators, reflecting the complexity of its dual-channel architecture.

The hardware integration connects the iDMA controller to the tile's OBI crossbar through a dedicated routing rule. All memory transactions targeting the iDMA address range are directed to an OBI decoder module (idma_obi_ctrl_decoder), which routes transactions to the appropriate channel-specific control registers. This decoder implements address-based demultiplexing, ensuring that register accesses reach the correct channel configuration space.

The controller architecture implements two independent DMA channels to support bidirectional data movement between memory hierarchies. The first channel (AXI-to-OBI) handles transfers from L2 memory to L1 scratchpad, converting AXI read transactions into OBI write transactions. The second channel (OBI-to-AXI) performs the inverse operation, transferring data from L1 to L2 by reading from OBI and writing to AXI. This dual-channel design supports common compute patterns in which input data is staged from global L2 memory into tile-local L1 before computation, and the results are written back to L2 after processing. The channels operate independently, enabling simultaneous bidirectional transfers that can pipeline data movement with computation.

Each channel exposes an identical register set replicated at different base addresses: AXI-

to-OBI (L2→L1) at 0x0000_0200 and OBI-to-AXI (L1→L2) at 0x0000_0400. Table 3.3 summarizes the register map for each channel.

Offset	Register	Access	Function
0x00	CONF	W	Transfer configuration
0x04	STATUS	R	Channel busy status (per stream)
0x44	NEXT_ID	R	Next transfer identifier
0x84	DONE_ID	R	Last completed transfer ID
0xD0	DST_ADDR	W	Destination address
0xD8	SRC_ADDR	W	Source address
0xE0	LENGTH	W	Transfer length (bytes)
0xE8	DST_STRIDE_2	W	2D destination stride
0xF0	SRC_STRIDE_2	W	2D source stride
0xF8	REPS_2	W	2D repetition count
0x100	DST_STRIDE_3	W	3D destination stride
0x108	SRC_STRIDE_3	W	3D source stride
0x110	REPS_3	W	3D repetition count

Table 3.3: iDMA memory-mapped register map (replicated for each channel)

The organization of the register mirrors the iDMA frontend interface, providing fields for the source address, the destination address, and the transfer length as base parameters. Optional stride parameters enable multi-dimensional transfers: SRC_STRIDE_2, DST_STRIDE_2, and REPS_2 configure 2D transfers, while SRC_STRIDE_3, DST_STRIDE_3, and REPS_3 extend to 3D operations. The CONF register encodes transfer options including dimensional enable flags and protocol-specific settings. The STATUS register tracks busy state for multiple concurrent streams, while NEXT_ID and DONE_ID registers implement a transfer tracking mechanism: software reads NEXT_ID when submitting a transfer to obtain its identifier, then polls DONE_ID to determine completion. Each channel generates four interrupt signals: irq_busy_o indicates ongoing transfer activity, irq_start_o signals transfer initiation, irq_done_o marks completion, and irq_error_o reports protocol or address errors. These eight total status signals (four per channel) enable fine-grained software coordination between DMA transfers and accelerator execution.

For software control, the interface (idma_mm_utils.h) provides direction aware access functions that abstract channel selection and register addressing. Helper functions include configuration routines to configure transfers (idma_l2_to_l1(), idma_l1_to_l2()), submission functions that return unique transfer identifiers, and synchronization primitives to track completion. The typical usage pattern involves configuring the appropriate channel with source, destination, length, and optional stride parameters; submitting the transfer to receive a transfer ID (optionally continuing with other work or configuring the opposite channel for bidirectional operation) and waiting for completion through explicit polling (dma_wait(transfer_id)) or barrier synchro-

nization (dma_barrier()) that blocks until all pending transfers complete.

3.1.4 FractalSync Memory-Mapping

Unlike RedMulE and iDMA, which had existing memory-mapped control infrastructure, FractalSync's original implementation relied exclusively on XIF custom instructions for its synchronization primitives. The transition to memory-mapped control required developing a completely new slave interface, implemented specifically for the MAGIA architecture in the obi_slave_fsync module.

The hardware integration directly connects the FractalSync slave interface to the tile's OBI crossbar. A dedicated routing rule directs all memory transactions within the address range 0x0000_0600–0x0000_06FF to FractalSync's OBI slave port. The implementation occupies 256 bytes, reflecting the minimal control requirements of the synchronization protocol. The OBI slave module implements the complete protocol conversion, translating memory-mapped register writes into internal synchronization control signals that propagate through the hierarchical synchronization tree. Table 3.4 summarizes the four-register interface.

Offset	Register	Access	Function
0x00	AGGR_REG	W	Aggregation level
0x04	ID_REG	W	Synchronization identifier
0x08	CONTROL_REG	W	Trigger synchronization
0x0C	STATUS_REG	R	Busy status (bit 2)

Table 3.4: FractalSync memory-mapped register map

FractalSync organizes tiles in a hierarchical tree mapped onto the two-dimensional mesh, enabling logarithmic-complexity barrier synchronization. This significantly reduces synchronization overhead compared to naive approaches; for instance, a 256-tile system requires only 8 tree levels ($\log_2 256$) for global synchronization. The architecture also supports synchronization at different granularities, ranging from neighbor-level barriers to full-system coordination.

The synchronization protocol relies on three memory-mapped registers. The AGGR_REG selects the aggregation level, determining the size of the synchronization group. The ID_REG identifies the barrier instance, allowing multiple independent synchronization contexts. Writing to CONTROL_REG triggers the operation. Completion can be detected by polling STATUS_REG, whose bit 2 indicates whether a synchronization is ongoing. Additionally, hardware signals (done_o, error_o) enable event-driven completion handling.

The software interface adapts the original Victor Isachi's XIF-based API to a memory-mapped model. A low-level layer provides direct register access, while a higher-level API exposes convenient functions such as global, row-wise, column-wise, and neighbor synchronization.

3.2 Event unit Integration

3.2.1 Hardware Architecture

The Event Unit provides centralized interrupt and event management for the MAGIA tile, addressing race condition issues that emerged during the coordination of multiple IPs. In the original architecture, completion signals generated by the accelerators were directly connected to the processor interrupt inputs. During pipelined execution patterns, multiple accelerators, such as RedMulE, iDMA, and FractalSync, could generate completion events nearly simultaneously. If several events arrived while the processor was already servicing an interrupt or executing unrelated code, some of these signals could be missed or incorrectly handled. This direct interrupt-based signaling mechanism therefore exposed the system to race conditions, potentially leading to synchronization failures and incorrect computation results. The Event Unit solves this problem by implementing a hardware event buffering and masking architecture based on the PULP event_unit_flex IP [13][14]. Unlike traditional interrupt controllers that route signals directly to processor interrupt lines without state retention, the Event Unit captures all incoming event signals in a 32-bit hardware buffer register. Each bit position corresponds to a specific event source, and the buffer accumulates events until explicitly cleared by the software. This accumulation before acknowledgment model ensures that events are never lost: they remain in the buffer until software can process them. As shown in Table 3.5, the Event Unit exposes set of memory-mapped registers.

Offset	Register	Access	Function
0x00	CORE_MASK	R/W	Enable event sources
0x1C	CORE_BUFFER	R	Pending events (latched)
0x20	CORE_BUFFER_MASKED	R	Pending events (masked)
0x28	CORE_BUFFER_CLEAR	W	Clear events (W1C)
0x38	CORE_EVENT_WAIT	R	Wait for event
0x3C	CORE_EVENT_WAIT_CLEAR	R	Wait + clear

Table 3.5: Event Unit: key memory-mapped registers

The Event Unit hardware implements two distinct access interfaces. The primary control interface connects to the tile's OBI crossbar through a standard periph_slave port, occupying 4 KB of address space (0x0000_0700–0x0000_16FF). This interface provides memory-mapped access to all Event Unit configuration and status registers, allowing software to configure event masks, read event buffers, clear acknowledged events, and control interrupt routing. The periph_slave port follows standard OBI protocol semantics with multi-cycle address and data phases. The secondary interface, eu_direct_link, implements a low-latency bypass path specifically optimized for Wait-For-Event (WFE) blocking operations. This direct link provides a dedicated connection between the Event Unit and the processor core's data port, bypassing the

OBI crossbar entirely. A custom demultiplexer module (core_data_demux_eu_direct) sits between the core and the crossbar, intercepting memory accesses that target the Event Unit address range (0x0000_0700–0x0000_16FF). When the core issues a read to an Event Unit register, the demux routes the request through the eu_direct_link instead of the crossbar, eliminating interconnect arbitration delays and protocol conversion overhead. This bypass architecture enables the p.elw (event load word) mechanism, a PULP-specific instruction extension that implements energy-efficient blocking synchronization. When the processor executes a p.elw load targeting an Event Unit wait register and no enabled events are pending, the core does not complete the load immediately as with standard RISC-V loads. Instead, the Event Unit de-asserts its core_clock_en_o control signal, which feeds a clock gating cell upstream of the processor core. The clock gating cell blocks clock distribution to the core, halting instruction execution and significantly reducing power consumption while maintaining architectural state. When an enabled event arrives at the Event Unit, edge-detection logic captures it in the buffer and immediately re-asserts core_clock_en_o. The clock gating cell responds by restoring the core's clock distribution, allowing the p.elw load to complete with the event bitmask as its return value. The eu_direct_link protocol differs from OBI by using a simplified request-response handshake with inverted write-enable polarity (wen instead of we), matching the Event Unit's native interface conventions inherited from PULP IP. For configuration operations through periph_slave, the Event Unit operates as a standard memory-mapped peripheral. Writes to mask registers, buffer clear operations, and interrupt configuration proceed through the OBI crossbar with normal multi-cycle latency.

The Event Unit collects event signals from all accelerators and peripheral components through dedicated input ports. Each component generates specific event types that are mapped to designated bit positions within a 32-bit event buffer. RedMulE provides three events mapped to bits 9–11: a busy indicator (bit 9), a completion signal (bit 10), and a secondary event (bit 11). The iDMA controller, reflecting its dual-channel architecture, contributes eight events: primary completion signals for the two transfer directions are mapped to bits 2 (AXI-to-OBI, L2→L1) and 3 (OBI-to-AXI, L1→L2), while extended status signals such as error, start, and busy indicators for both channels occupy bits 26–31. FractalSync generates completion (bit 24) and error (bit 25) events, while Spatz CC produces completion (bit 8) and start trigger (bit 23) signals. Once set, a buffer bit remains asserted until software explicitly clears it by writing to the BUFFER_CLEAR register.

Event visibility and interrupt generation are controlled through mask registers. The CORE_MASK register determines which events are visible to software: only events whose corresponding mask bit is enabled appear in masked buffer reads and can trigger wake operations. Events arriving while their mask bit is disabled are still accumulated in the hardware buffer but remain hidden until the mask is enabled. A second mask layer, CORE_IRQ_MASK, controls interrupt generation. When an event occurs and both the CORE_MASK and CORE_IRQ_MASK bits are enabled, the Event Unit asserts an interrupt toward the processor. This two-level filter-

ing mechanism enables hybrid synchronization strategies, allowing interrupts for rare or hazard events while using polling or WFE for frequent accelerator operations.

3.2.2 Software Interface

The software interface (event_unit_utils.h) provides two primary synchronization strategies: polling and Wait-For-Event (WFE). Polling mode implements active waiting through repeated reads of the Event Unit buffer registers. Software can poll either CORE_BUFFER (returning all accumulated events regardless of mask configuration) or CORE_BUFFER_MASKED (returning only events enabled in CORE_MASK). A typical polling loop repeatedly reads CORE _BUFFER_MASKED until the desired event bits appear. For example, waiting for RedMulE completion using polling:

```
/* Enable RedMulE completion event in mask */
eu_redmule_init();
redmule_start(...);
while (!(eu_read_buffer_masked() & EU_REDMULE_DONE_MASK));
```

Polling allows the core to continue to execute other tasks while periodically checking whether the operation has been completed. This approach provides flexibility, as the software can query the status register only when needed and interleave useful computation during the waiting phase. However, polling requires the core to remain active and continuously read the status register, increasing power consumption.

Alternatively, the WFE (Wait-For-Event) mode uses the p.elw instruction to place the processor in a low-power sleep state until an event occurs. If useful work cannot be performed while waiting, WFE is generally preferable due to its higher energy efficiency. Conversely, if the core has other tasks to execute, polling may be advantageous as it avoids stalling the processor and allows for continued forward progress. A typical WFE workflow is shown below.

```
/* Enable iDMA channel completion event */
eu_idma_init();
/* Configure and trigger DMA transfer */
idma_start_transfer(...);
/* Sleep until completion */
uint32_t events = eu_wait_events_wfe(EU_IDMA_A2O_DONE_MASK);
/* Process transferred data */
```

3.3 Replacement of CV32E40X with CV32E40P

The CV32E40P core was instantiated as module i_cv32e40p_core within magia_tile.sv. The core interfaces include instruction and data memory ports, interrupt signals connected to the Event Unit, and the debug interface.

The same 16KB pulp_icache_wrap instruction cache module was maintained from the previous core implementation, requiring no modifications. The cache connects the core's fetch interface to L2 memory through AXI.

A core data interface demultiplexer module (core_data_demux_eu_direct) was implemented to route data memory requests based on the address range. Requests to the Event Unit address range (0x10404000–0x104040FF) are routed through a dedicated direct path to the Event Unit, while all other requests go through the OBI crossbar to access L1 SPM, L2 memory, and peripherals. The direct path was added to reduce p.elw instruction latency.

The ISA string was updated from XTEN = imafc to XTEN = imfcxpulpv2 in the Makefile, resulting in the complete specification rv32imfcxpulpv2, which adds support for XpulpV2.

3.4 Spatz Core Complex Integration

In the following, it is explained how the Spatz Core Complex was integrated into the MAGIA tile. The Spatz Core Complex (Spatz CC) is a RISC-V vector core that implements the Zve32d extension, as described in the Background chapter. The CV32E40P host core controls the Spatz CC through memory-mapped registers and interrupt-driven task dispatching, following an accelerator-style programming model.

3.4.1 Hardware Integration

Spatz CC was integrated into the MAGIA tile through the spatz_cc_wrapper module. The wrapper encapsulates the Spatz CC core and provides interface adaptation between Spatz's native TCDM protocol and the tile's HCI interconnect. The Spatz CC implements configurable vector length (VLEN: 128-512 bits), configurable number of FPU lanes (1-8), and optional double-precision support (RVD).

The Spatz CC configuration parameters are set through Makefile defines and propagated to magia_tile_pkg.sv. The main parameters include: SPATZ_RVD (0 for 32-bit TCDM/ELEN = 32, 1 for 64-bit TCDM/ELEN = 64), SPATZ_VLEN (vector register length in bits, default 256), SPATZ_N_FPU (number of FPU lanes, default 4), SPATZ_N_IPU (number of integer processing units, default 1), SPATZ_RVF (single-precision FP support, default 1), SPATZ_RVV (vector extension enable, default 1), and SPATZ_XDIVSQRT (FP division/square root enable, default 0). The number of TCDM ports is computed as SPATZ_NUM_FU + 1, where SPATZ_NUM_FU is the maximum of N_FPU and N_IPU. The number of HCI ports is doubled when RVD = 1 to support 64-bit transfers using two 32-bit HCI ports per TCDM port.

The Spatz CC connects to the tile through three primary interfaces: an OBI slave interface for control registers, HCI master ports for L1 SPM access, and an OBI master port for general memory operations. The OBI slave interface exposes seven memory-mapped control registers at base address 0x00001700. Table 3.6 lists the control registers with their offsets and descriptions.

Offset	Register	Access	Function
0x00	SPATZ_CLK_EN	W	Controls power gating: 1 enables clock, 0 disables clock
0x04	SPATZ_READY	W/R	Initialization flag: accelerator sets to 1 when ready, cleared after first task
0x08	SPATZ_START	W/R	Interrupt trigger: host sets to 1, accelerator clears to 0 after acknowledgment
0x0C	SPATZ_TASKBIN	W/R	Entry point register: holds task function address
0x10	SPATZ_DATA	W/R	Parameter exchange: pointer to task input parameters
0x14	SPATZ_RETURN	W/R	Task result: 0 indicates success, non-zero indicates error or exception
0x18	SPATZ_DONE	W	Completion pulse: generates 1-cycle event signal, auto-clears

Table 3.6: Spatz CC Control Registers

The first six registers are preserved registers that maintain their values until explicitly overwritten. The SPATZ_DONE register is a pulse register that auto-clears after one cycle. The operation protocol follows this sequence: the host writes the binary start address of the accelerator to SPATZ_TASKBIN and activates the clock via SPATZ_CLK_EN. The host polls SPATZ_READY until it becomes 1, indicating that initialization is complete. The host writes the task entry point to SPATZ_TASKBIN, optionally writes a parameter pointer to SPATZ _DATA and triggers the execution by setting SPATZ_START to 1, which generates an external interrupt. The accelerator wakes up from the wait-for-interrupt state, reads the task address, and removes SPATZ_START to 0 as acknowledgment. The task executes, reads the parameters from the address in SPATZ_DATA, and after completion writes the result to SPATZ_RETURN and sets SPATZ_DONE to 1. The SPATZ_DONE signal generates a 1-cycle pulse to the Event Unit. The host reads SPATZ_RETURN to verify task completion: value 0 indicates success, non-zero indicates error.

The HCI master interface connects Spatz CC to the L1 SPM through the HCI interconnect with multiple ports. The number of HCI ports depends on the vector configuration: with RVD = 0 and N_FPU = 4, Spatz has 5 TCDM ports (4 vector FPU ports + 1 Snitch scalar port) which results in 5 master HCI ports. With RVD = 1, each TCDM port requires two 32-bit HCI ports for 64-bit data transfers, resulting in 10 HCI master ports. The HCI interconnect routes these requests to the L1 SPM's interleaved memory banks.

The OBI master interface connects the Snitch scalar core to the OBI crossbar as the third master port. This interface is used for memory accesses outside the L1/TCDM address space,

including peripherals, L2 memory, and cross-tile communication. The OBI crossbar routes Spatz requests to tile resources: L2 memory, L1 SPM, accelerator control registers (RedMulE, iDMA, FSync), Event Unit registers, and Spatz control registers.

Clock gating is implemented for the Spatz CC and is controlled through the SPATZ_CLK_EN register. When SPATZ_CLK_EN = 1, the clock is active; when SPATZ_CLK_EN = 0, the clock is gated to reduce power consumption. The CV32 writes 1 to SPATZ_CLK_EN before initializing Spatz and can write 0 after task completion for power saving.

The Spatz CC is triggered via an external interrupt. The SPATZ_START register output is connected to the machine external interrupt signal. When the host writes 1 to SPATZ_START, the Snitch core receives an external interrupt, wakes from WFI (wait-for-interrupt), reads the task address from SPATZ_TASKBIN, clears SPATZ_START to 0, and jumps to the task. Upon completion of the task, Snitch writes the exit code to SPATZ_RETURN and sets SPATZ_DONE to 1, which generates a pulse of 1-cycle routed to the Event Unit.

The Spatz CC has a dedicated instruction cache with the following configuration: 1 fetch port, 8 L0 cache lines, 32 L1 cache lines of 256 bits each, 2-way set associative organization, and 2 outstanding AXI transactions. The instruction cache connects to the core's fetch interface and to the AXI crossbar. The AXI crossbar routes instruction fetch requests to L2 memory through the NoC.

The Spatz CC boot address is set to 0x10000000, which maps to a minimal bootrom. The bootrom contains only 3 instructions: load the address from SPATZ_TASKBIN register, read the value from that address into a temporary register, and jump to that address. CV32 must write the Spatz runtime entry point to SPATZ_TASKBIN before enabling the Spatz clock for the first time. This minimal bootrom design reduces hardware area and ROM overhead. Figure 3.1 shows the updated tile architecture presented in this chapter.

3.4.2 Software Integration

The software integration consists of runtime initialization code, a task dispatcher, a trap handler, and C API functions. The runtime code is spatz_crt0.S and provides complete initialization transparent to the programmer.

The runtime initialization sequence performs the following steps: set stack pointer to 0x 0001FFF8 (128KB - 8 bytes, stack grows down from top of Snitch local memory), clear all integer registers x1-x31 except stack pointer, enable vector extension by setting mstatus.VS = 0x200 (Initial state), clear BSS section by zeroing memory between __bss_start and __bss_end, install trap handler at mtvec, enable external interrupts by setting mie.MEIE = 0x800, enable global interrupts by setting mstatus.MIE = 0x8, write 1 to SPATZ_READY register to signal CV32 that initialization is complete, execute first WFI (wait for interrupt), and clear SPATZ_READY to 0 after first wakeup. After the first task completes, the execution enters a dispatcher loop consisting of WFI followed by an unconditional jump back to WFI.

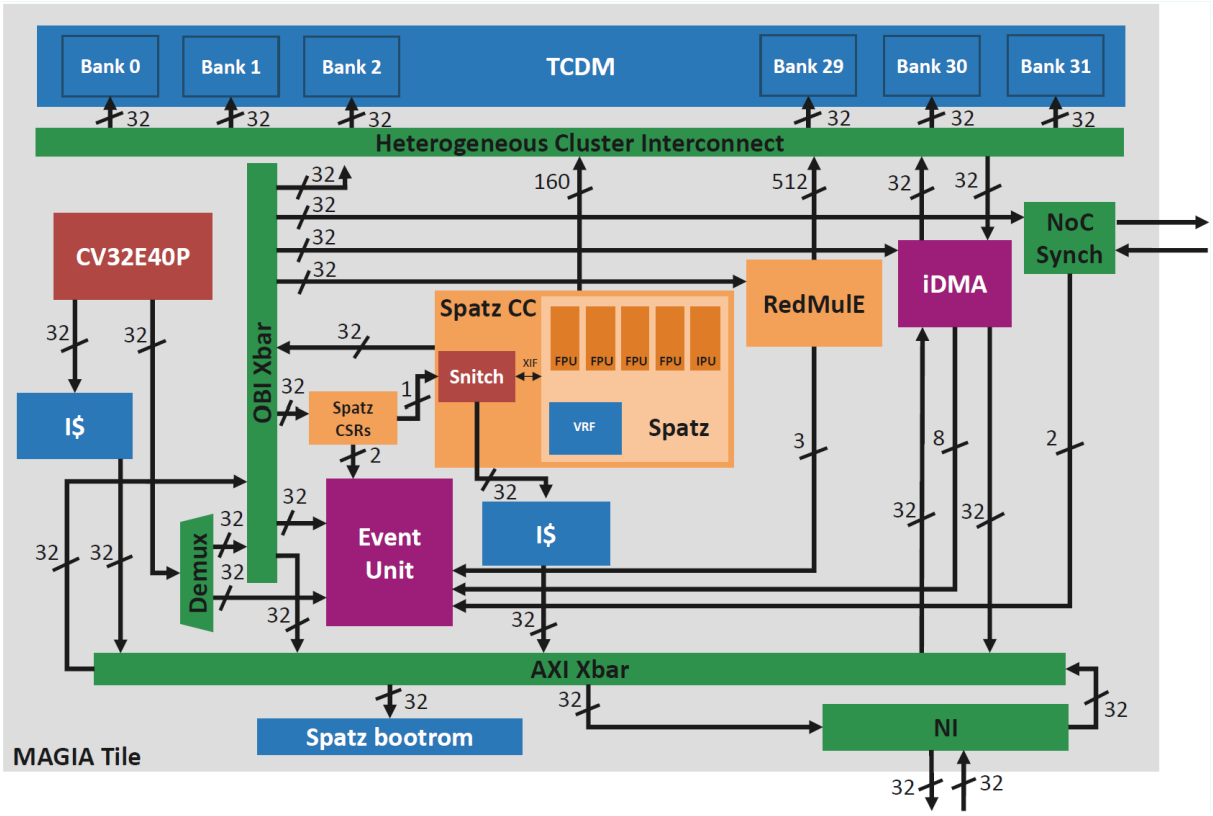

Figure 3.1: Updated MAGIA tile Block Diagram

The SPATZ_READY register provides synchronization between host and accelerator. After initialization, the accelerator sets SPATZ_READY to 1 to indicate that it has entered the WFI state and is ready to receive interrupts. The host polls SPATZ_READY until it becomes 1 before triggering the first task. After the first interrupt wakes the accelerator, it clears SPATZ_READY to 0 (only once) to indicate that normal operation has begun. This mechanism prevents race conditions where the host might trigger an interrupt before the accelerator enters WFI.

The trap handler distinguishes between interrupts and exceptions by checking the trap cause register. For interrupts, the handler verifies that it is an external interrupt, reads the task address from SPATZ_TASKBIN, clears SPATZ_START to 0 to deassert the interrupt signal, jumps to the task address, waits for the task to return, writes exit code 0 (success) to SPATZ_RETURN, sets SPATZ_DONE to 1 to generate the pulse of the Event Unit, and returns from the trap. For exceptions, the handler reads the exception code, sets bit 8 to distinguish exceptions from task errors, writes the error code to SPATZ_RETURN, sets SPATZ_DONE to 1, and halts in an infinite wait loop.

The exit code convention uses value 0 for successful task completion, values in the range 0x100-0x1FF for hardware exceptions encoding the trap cause, and values 1-255 for task-specific errors.

The C API provides functions for controlling Spatz CC through memory-mapped register accesses. The clock control functions write to SPATZ_CLK_EN to enable or disable the accelerator clock. Initialization writes the runtime entry point to SPATZ_TASKBIN, enables the

clock, and polls SPATZ_READY until initialization completes. Task execution writes the task address to SPATZ_TASKBIN, triggers the interrupt by setting SPATZ_START to 1, writes a pointer to SPATZ_DATA to pass data and waits for SPATZ_START to clear to 0. Combined functions handle both parameter passing and task triggering. The result retrieval reads the value of SPATZ_RETURN.

Spatz tasks are written as C functions following a naming convention where task source files end with a specific suffix and the task function name matches the filename. Tasks retrieve their parameters by reading SPATZ_DATA, which provides a pointer to a parameter structure stored in L1 memory. Each task returns an integer exit code to the host, where a value of 0 indicates successful completion, while non-zero values signal an error.

The typical execution flow involves a host program running on the CV32E40P core that prepares the task parameters, starts the execution on Spatz, and waits for completion through an Event Unit notification. An example host-side program that launches a Spatz task and waits for its completion is shown below.

```
#include "magia_spatz_utils.h"
#include "event_unit_utils.h"
#include "my_test_task_bin.h"

typedef struct {
    uint32_t addr;
    uint32_t size;
} params_t;

int main(void)
{
    eu_init();
    eu_enable_events(EU_SPATZ_DONE_MASK);
    spatz_init(SPATZ_BINARY_START);

    params_t params = {
        .addr = DATA_BASE,
        .size = 1024
    };

    spatz_pass_params((uint32_t)&params);
    spatz_run_task(MY_VECTOR_TASK);

    eu_wait_spatz_wfe(EU_SPATZ_DONE_MASK);

    if (spatz_get_exit_code() != 0)
        return 1;

    spatz_clk_dis();
    return 0;
```

```
}
```

The following code shows the corresponding implementation of a Spatz task. The task retrieves the parameter pointer from SPATZ_DATA, accesses the parameter structure stored in L1 memory, configures the vector unit, and performs the desired computation before returning an exit code to the host.

```
#include "magia_spatz_utils.h"

typedef struct {
    uint32_t data_addr;
    uint32_t size;
} task_params_t;

int my_vector_task(void)
{
    uint32_t params_ptr = mmio32(SPATZ_DATA);

    volatile task_params_t *params =
        (volatile task_params_t *)params_ptr;

    uint32_t data_addr = params->data_addr;
    uint32_t size      = params->size;

    asm volatile(
        "vsetvli zero, %0, e32, m4, ta, ma"
        :
        : "r"(32)
    );

    /* Vector operations ... */

    return 0;
}
```

The build flow automatically detects Spatz tasks and generates binary headers. The top-level Makefile scans the host test source files for task symbols. When tasks are detected, the Makefile invokes the Spatz build system passing configuration parameters. The Spatz Makefile compiles all referenced tasks, links them with the runtime and bootrom into a single ELF binary, converts the binary to a C array, and generates a header file containing the binary array and task symbol addresses as offsets. The generated header defines a binary start address symbol and each task symbol as an offset from this base. The host test must include the generated header to access these symbols. A typical host test using Spatz follows this pattern: include the auto-generated header containing task symbols, initialize the Event Unit and enable accelerator completion events, initialize Spatz CC with the binary start address, optionally prepare a parameter structure in L1 memory and pass its pointer, trigger task execution with the task

symbol from the header, wait for completion via Event Unit event, check SPATZ_RETURN for the exit code, and optionally disable the clock via SPATZ_CLK_EN for power saving. The build process is identical whether the test uses Spatz or not, as task detection and compilation occur automatically.

Chapter 4

Physical Implementation

The physical implementation of digital integrated circuits is a critical phase that bridges the gap between the logical design and the actual silicon fabrication. This chapter presents the synthesis and place and route flows employed for the design under study, describing the methodologies, tools, and results obtained across different design configurations.

4.1 Introduction to Physical Design Flow

In digital design, the physical implementation flow transforms a functionally verified Register Transfer Level (RTL) description into a manufacturable layout. This process involves several stages, from logic synthesis to physical placement and routing, each optimizing different aspects of the final chip, such as area, timing, and power consumption.

The physical design flow begins with logic synthesis, where the high-level behavioral description is mapped to a gate-level netlist using standard cells from a technology library. Subsequently, the place and route stage arranges these cells spatially on the chip and creates the interconnections, while meeting timing constraints, power targets, and design rules dictated by the technology node.

For this work, the physical implementation targets the GlobalFoundries 12 nm technology node with multiple threshold voltage (Vt) variants. The available standard cell libraries include Regular Threshold Voltage (RVT), Low Threshold Voltage (LVT), and High Threshold Voltage (HVT) cells, enabling tools to explore different trade-offs between performance and leakage power during synthesis and optimization. In addition, on-chip memory macros were generated using the GlobalFoundries memory compiler, providing area and power efficient SRAM instances tailored to the design requirements.

To properly evaluate the architectural impact of the proposed extensions, the complete physical design flow was implemented for two tile configurations. The first configuration corresponds to a baseline tile based on the MAGIA architecture without the integration of Spatz CC, which serves as a reference implementation. The second configuration extends the tile by integrating

Spatz CC, enabling vector processing capabilities in addition to RedMulE.

Both configurations were synthesized, placed, and routed using the same technology libraries, design constraints, and operating conditions. This approach enables the evaluation of the impact introduced by the integration of Spatz CC with respect to the baseline MAGIA tile implementation.

4.2 Synthesis

4.2.1 Synthesis Flow

The logic synthesis flow was originally developed by Riccardo Fiorani Gallotta and subsequently adapted to the current design requirements. The synthesis process employs Synopsys Design Compiler as the Electronic Design Automation (EDA) tool to transform the RTL code into an optimized gate-level netlist targeting the 12 nm GlobalFoundries process.

The synthesis flow operates under specific Process-Voltage-Temperature (PVT) corners to ensure robust operation across manufacturing variations and environmental conditions. For this implementation, the worst-case synthesis corner was selected to guarantee timing closure under the most challenging operating conditions. The corner specifications include a Slow-Slow Process-worst Geometry (SSPG) corner to account for worst-case device characteristics, a supply voltage of 0.72V to model low-voltage operating conditions, cold temperature conditions of -40°C representing the worst-case scenario for certain timing paths, and a target clock period of 1.25 ns corresponding to an 800 MHz operating frequency.

The synthesis tool utilizes multi-voltage optimization techniques, selecting appropriate threshold voltage cells (RVT, LVT, and HVT) to balance performance and power consumption. The clock gating structures are automatically inserted to reduce dynamic power, and hierarchical synthesis strategies are used to manage design complexity.

Memory elements in the design include both register files and SRAM macros. These macros were pre-characterized for the 12 nm GlobalFoundries technology and integrated into the Synopsys synthesis flow through timing libraries (.lib) and physical abstractions (LEF files). The GlobalFoundries memory compiler generates these macros with specific configurations optimized for access time, area, and power efficiency.

The synthesis flow follows these main steps:

1. **RTL elaboration and linking**: The design hierarchy is parsed and all modules are linked with their dependencies

2. **Constraint definition**: Timing constraints, including clock definitions, input/output delays, and false paths, are applied

3. **Logic optimization**: The tool performs technology-independent optimization followed by technology mapping using the standard cell library

4. **Design for Test (DFT) insertion**: Optional scan chain insertion for manufacturing test (when enabled)

5. **Incremental optimization**: Iterative refinement to meet timing, area, and congestion targets

6. **Report generation**: Comprehensive reports on area, timing, power, and quality of results (QoR)

The output of the synthesis flow includes a gate-level Verilog netlist, timing constraints in Synopsys Design Constraints (SDC) format, and various reports documenting the achieved results.

To guide the synthesis and physical implementation tools toward achieving the desired performance, area, and power targets, design constraints are defined using the Synopsys Design Constraints (SDC) format. The fundamental timing constraint is the clock definition with a period of 1.25 ns (800 MHz), with clock uncertainty margins of 100-125 ps (8-10% of the clock period) specified to account for clock jitter, skew, and on-chip variation effects. Input and output delay constraints model interface timing with external systems, typically ranging from 20-40% of the clock period. False path and multicycle path constraints handle asynchronous signals and multicycle operations. Maximum transition, capacitance, and fanout constraints ensure signal integrity and manageable load distribution. These constraints are refined iteratively during synthesis, and place and route are refined to achieve an optimal trade-off between performance, area and power.

4.2.2 Post Synthesis Results

The synthesis results were analyzed for two design configurations: the baseline MAGIA tile architecture without the Spatz CC vector processor and the complete MAGIA tile including the integrated Spatz CC. Table 4.1 presents a comprehensive comparison of the area metrics obtained from both configurations.

Metric	**Without Spatz CC**	**With Spatz CC**
Total Cell Area [μm^2]	1,219,957.53	1,290,336.71
Combinational Area [μm^2]	161,851.05	201,743.42
Non-combinational Area [μm^2]	110,414.95	132,376.44
Macro/Black Box Area [μm^2]	947,691.53	956,216.84
Buffer/Inverter Area [μm^2]	14,931.79	18,196.34
Number of Cells	1,068,355	1,277,408
Combinational Cells	895,815	1,069,012
Sequential Cells	164,533	197,056
Macros	36	40

Table 4.1: Post-synthesis area comparison between design configurations

A notable characteristic of this design configuration is the dominance of macro area, which accounts for approximately 80% of the total cell area in the baseline configuration and 74% in the Spatz CC-integrated version. This area is predominantly occupied by the L1 scratchpad memory (SPM) and instruction cache (iCache), a design choice that reflects the architecture's focus on accelerator-centric computing. The L1 memory serves as the primary data storage for computational accelerators, storing the working datasets that must be processed with minimal access latency. This memory-centric approach enables high data reuse and bandwidth, critical requirements to achieve peak performance in vector and matrix operations.

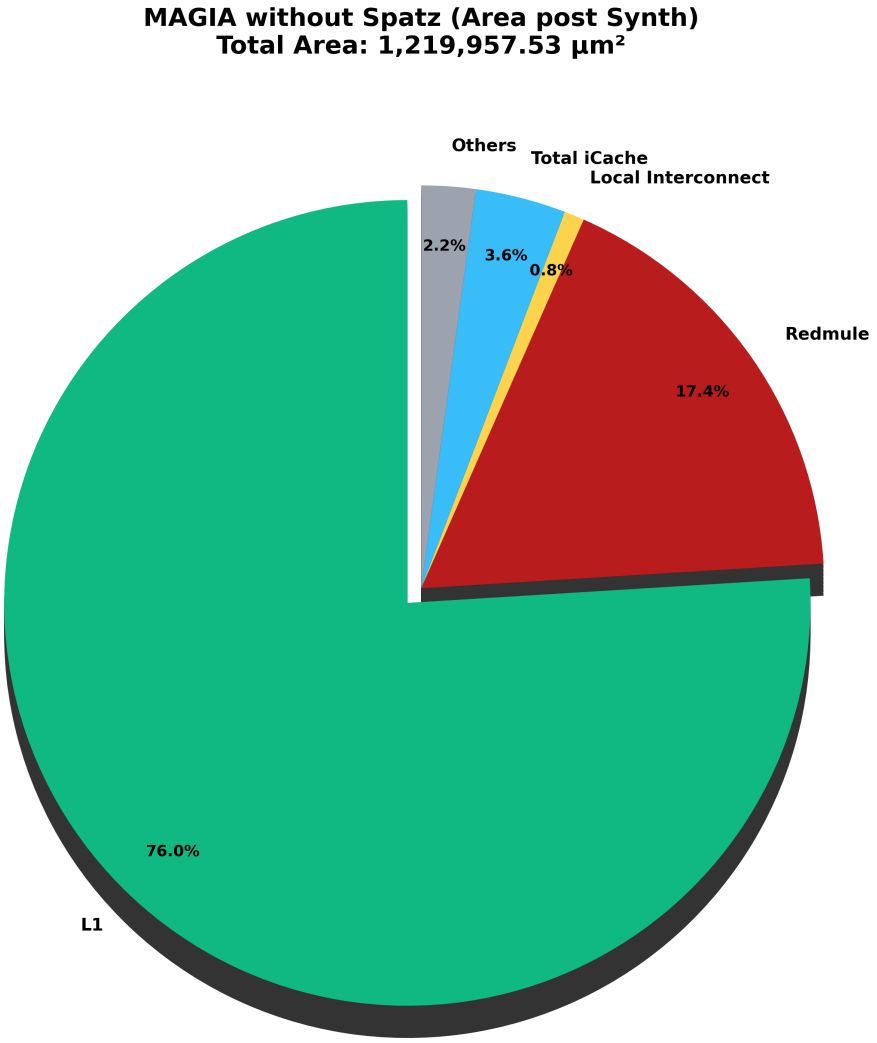

Figure 4.1: Percentage area breakdown of the baseline MAGIA tile without Spatz CC

Integration of the Spatz CC vector processor introduces an area overhead of approximately 70,379 μm^2, representing a 5.8% increase in total cell area. This overhead is distributed across different components. The combinational area increases by 39,892 μm^2 (24.6% increase), reflecting the additional datapath logic required for vector processing operations. The Non-combinational area grows by 21,961 μm^2 (19.9% increase), corresponding to the state registers

and pipeline stages in the Spatz CC vector processor. Four additional memory macros are instantiated for Spatz CC, contributing 8,525 μm^2 (0.9% increase) to the total macro area. The cell count increases from approximately 1.07 million to 1.28 million cells, with sequential elements increasing from 164,533 to 197,056 registers, reflecting the introduced Spatz CC vector processor.

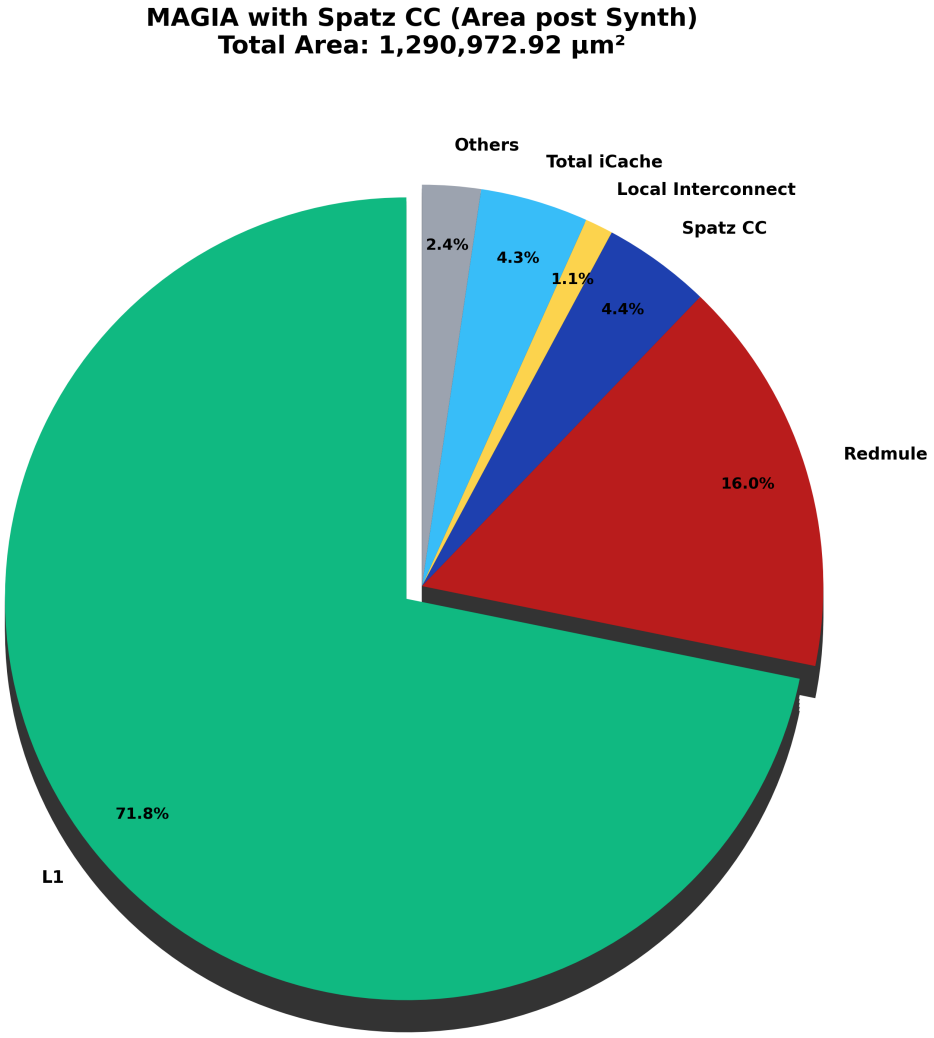

Figure 4.2: Percentage area breakdown of the MAGIA tile with Spatz CC

From a timing perspective, both configurations achieve timing closure at the target clock period of 1.25 ns under the specified corner conditions.

4.3 Place and Route

4.3.1 Place and Route Flow

The place and route (PnR) implementation was also originally developed by Riccardo Fiorani Gallotta and was adapted for the current design iterations. This physical design stage takes the gate-level netlist from Design Compiler synthesis and produces a physical layout that includes standard cell placement, clock tree synthesis, and detailed routing using Cadence Innovus. Cadence Innovus employs advanced placement and routing algorithms to optimize timing closure, power consumption, routability, and design rule compliance. The tool operates within a Multi-Mode Multi-Corner (MMMC) analysis framework, evaluating the design across various operating scenarios.

The implementation flow begins with design import, where the synthesized netlist and 12 nm GlobalFoundries technology files are loaded, and timing constraints are configured for MMMC analysis. During floorplanning, the chip dimensions are estimated based on the post-synthesis standard-cell area. To account for additional physical design overheads, such as routing resources, clock tree insertion, and placement constraints, a utilization margin is applied. Specifically, the die area is estimated by multiplying the synthesized cell area by a factor of 1.3. This margin provides sufficient space to accommodate interconnect routing and physical design optimizations, ensuring that the design can be successfully placed and routed without critical congestion. The baseline MAGIA tile without Spatz CC, with a synthesis cell area of 1,219,957 μm^2, is implemented in a 1200 μm × 1200 μm floorplan (1.44 mm^2), while the MAGIA tile with Spatz CC requires 1300 μm × 1300 μm (1.69 mm^2), representing a 17% increase in physical die area (Fig. 4.3). Hard macros are placed strategically, and the power grid is synthesized across all 13 metal layers provided by the 12 nm GlobalFoundries technology.

Following power planning, global and detailed placement optimize cell locations for timing and routability. Clock Tree Synthesis (CTS) generates balanced clock distributions with minimal skew, followed by hold violation fixing. Routing proceeds through global and detailed stages, with post-route optimization addressing timing violations. The finishing stage inserts filler cells, performs final checks for DRC compliance, and completes the layout with metal fill to meet density requirements.

4.3.2 Post Place and Route Results

Running the complete physical design flow on both implementations allowed us to assess whether the integration of the Spatz CC introduced significant differences in timing closure or physical design constraints. The results show that no major differences emerged between the two configurations, except for the expected increase in area and a different macro placement due to the additional Spatz CC icache. Overall, the integration of Spatz CC does not introduce critical challenges from a physical design perspective.

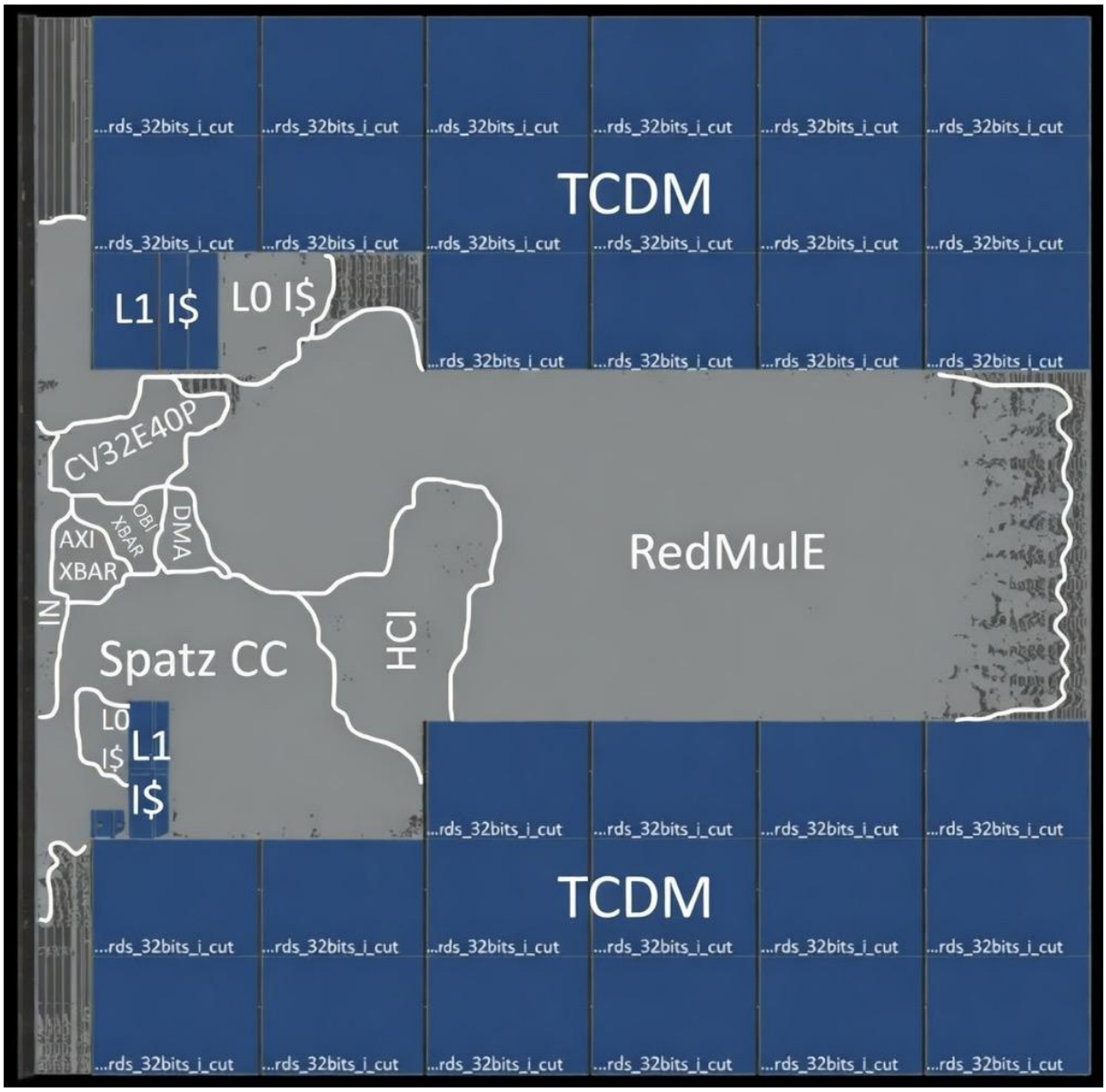

Figure 4.3: Tile layout [1300 µm x 1300 µm]

For power estimation purposes, parasitic resistance and capacitance (RC) values were extracted from the post-layout netlist of the MAGIA tile with the integrated Spatz CC. The extracted interconnect parasitics account for routing effects across the technology metal stack and provide the necessary information for accurate dynamic and leakage power analysis.

Then a static timing analysis was performed on the post-layout netlist. In the worst-case functional corner, the design approaches the target operating frequency of 800 MHz, exhibiting a Worst Negative Slack (WNS) of -0.050 ns, which corresponds to approximately 4% of the clock period (1.25 ns). Hold time constraints are satisfied across all analysis views.

Chapter 5

Performance Comparison

The MAGIA architecture integrates two accelerators with different optimization targets: Spatz is a vector processor implementing RISC-V Vector Extension instructions suitable for vectorial operations and flexible data access patterns, while RedMule is a matrix multiplication accelerator with dedicated hardware for GEMM operations. This chapter compares their performance and power consumption in four fundamental linear algebra benchmarks. The evaluation measures execution cycles and energy consumption through cycle-accurate simulation with hardware performance counters for timing analysis and post place and route netlist simulation with VCD waveform capture for gate-level power estimation using Synopsys PrimeTime.

Each benchmark adheres to a defined measurement methodology. First, data is initialized in the L1 scratchpad memory. A preliminary phase is then executed to warm-up the accelerator before collecting the measurements. The actual computation is then profiled by sampling the CV32E40P cycle counter immediately before and after accelerator execution.

For power characterization, the same test suite is executed with Value Change Dump (VCD) generation enabled via testbench control registers, which are mapped in the L2 memory space from the tile's perspective. Signal activity is recorded exclusively during the computation interval, thereby constraining VCD size to a manageable level, even for the post-layout netlist comprising several hundred thousand logic gates.

5.1 Benchmarks

The benchmark suite comprises four operations implemented for both accelerators in FP16 precision. Table 5.1 shows the sizes of the tested problems.

The dot product benchmark computes the inner product of two N-element vectors, performing N multiply-accumulate operations to produce a scalar result. The matrix-vector multiplication benchmark extends this pattern to compute the product of an $M \times N$ matrix with an N-element vector, producing an M-element result through M independent dot products with $M \times N$ multiply-accumulate operations.

Benchmark	Problem Size
Matrix-Matrix Multiply	$M \times N \times K$, with $N = 96$ and $M = K = 1, 2, 4, 8, 16, 32, 64, 96$
Matrix-Vector Multiply	$N \times N$ matrix with $N = 64, 96, 128, 256$
Dot Product	vector length $N = 16, 32, 64, 128, 256, 512, 1024, 2048$
Vector Sum	vector length $N = 16, 32, 64, 128, 256, 512$

Table 5.1: Benchmark problem sizes

The matrix multiplication benchmark evaluates general matrix-matrix products, computing $Y = X \times W$ where X has dimensions $M \times N$, W has dimensions $N \times K$, and the result Y has dimensions $M \times K$. The profiling configuration fixes $N = 96$ while varying M and K together across powers of two from 1 to 96, enabling evaluation of both highly rectangular matrices ($M = K = 1$, simulating single-sample inference through a 96-dimensional layer) and square 96×96 matrix products. The vector sum benchmark implements element-wise addition of two N-element vectors, performing N addition operations to combine corresponding elements from the input vectors.

Performance measurements were conducted through systematic compilation and execution of each benchmark across a range of problem sizes. The cycle counting mechanism leverages the CV32E40P's integrated performance counter accessible through custom CSR (Control and Status Register) instructions. Before computation begins, the test programs invoke cv32e40p_ccount_enable() to activate counter 0, which increments on every clock cycle. The measurement region starts immediately before triggering the accelerator and concludes when the accelerator signals completion through the event unit. Reading the cycle counter at both boundaries through cv32e40p_get_cycles() provides the elapsed cycle count for the computation. The following code excerpt from the Spatz dot product performance test demonstrates this measurement approach:

```
/* Configure parameters for N_SIZE elements */
params->n_size = N_SIZE;
spatz_pass_params(PARAM_BASE);

/*Perform Profiling*/
uint32_t start_cycles = cv32e40p_get_cycles();
spatz_run_task(DOTP_TASK);
eu_wait_spatz_polling(EU_SPATZ_DONE_MASK);
uint32_t end_cycles = cv32e40p_get_cycles();
uint32_t cycles = end_cycles - start_cycles;

/* Store results for CSV export */
results[0].M = 1;
results[0].N = N_SIZE;
results[0].macs = N_SIZE;
```

```
results[0].spatz_cycles = cycles;
```

This measurement methodology captures the complete execution time including accelerator dispatch overhead, data access latency, and computation time. The cycle counter operates synchronously with the tile's clock domain, providing exact cycle counts without approximation. By bracketing only the accelerator execution phase these measurements are focusing exclusively on the computational kernel performance. The results are stored in a structured format at a designated L2 memory address where the verification IP (VIP) automatically extracts them and generates CSV files for subsequent analysis.

Power analysis builds on the performance measurement framework by incorporating gate-level simulation with comprehensive signal activity recording. The VCD dumping mechanism operates through memory-mapped control registers monitored by the simulation environment. The test programs include the header file vcd_dump.h which defines macros that write specific values to the VCD trigger address at 0xCCFF1000. When the simulation environment detects the start value (0xABBAABBA), it initiates VCD recording of all netlist signals, and upon detecting the stop value (0xDEADCACA), it terminates recording and closes the VCD file. This software-controlled dumping restricts the VCD output to the exact computation region of interest, targeting the accelerator phase.

The power test structure mirrors the performance tests but inserts VCD control macros around the measured computation. After a warm-up execution, the test initiates VCD dumping immediately before triggering the accelerator, as illustrated in this excerpt from the Spatz dot product power test:

```
/* Warm-up with small problem size */
params->n_size = 16;
spatz_pass_params(PARAM_BASE);
spatz_run_task(DOTP_TASK);
eu_wait_spatz_polling(EU_SPATZ_DONE_MASK);

/* Configure for actual test size */
params->n_size = N_SIZE;
spatz_pass_params(PARAM_BASE);

VCD_DUMP_START_SIMPLE();
spatz_run_task(DOTP_TASK);
eu_wait_spatz_wfe(EU_SPATZ_DONE_MASK);
VCD_DUMP_STOP_SIMPLE();
```

The generated VCD files are subsequently processed by Synopsys PrimeTime, which performs gate-level power analysis by correlating the switching activity recorded in the VCD with the characterized delay and power models from the technology library. This analysis accounts for the dynamic power of signal transitions, the short-circuit power during gate switching, and

the static leakage power.

5.2 Power Analysis Flow

The power analysis methodology processes VCD files generated from post place and route netlist simulations through an automated batch flow. The run_all_power_analysis.csh script iterates through all VCD files for a specified benchmark test, invoking PrimeTime for each file through the run_power_analysis_parameterized.tcl script. This batch processing generates individual power reports for each problem dimension, enabling systematic comparison across the benchmark parameter space.

The analysis operates in the typical process corner using the tt_nominal_max technology libraries at a supply voltage of 0.80V and a temperature of 25°C. Memory macros, including the 8192×32 SRAM banks for L1 storage and register file structures, use characterized models at the tt_nominal corner that match the standard cell operating conditions. This typical corner analysis reflects the nominal silicon process parameters and ambient operating conditions representative of the typical scenario deployment.

The power estimation flow begins by reading the gate-level Verilog netlist produced by Innovus place and route, which contains the complete synthesized design with optimized standard cell mapping and clock tree insertion. After linking the design hierarchy, the script loads the timing constraints from the SDC file to define clock specifications and establish the timing context required for an accurate power calculation. The RC parasitic extraction file in SPEF format provides resistance and capacitance values for all nets extracted from the physical layout. The place and route tool generates three RC corner variants: typical_rc.spef.gz for the nominal resistance and capacitance of the interconnect, best_rc.spef.gz for the minimum RC values corresponding to the best-case routing conditions and worst_rc.spef.gz for the maximum RC values representing the worst-case interconnect delays. Power analysis uses typical RC extraction to match the nominal process corner assumption, loading these parasitics with the -keep option _capacitive _coupling. Switching activity annotation reads the VCD file and correlates signals transitions to nets in the synthesized netlist. After annotating switching activity and loading parasitics, the script executes update_timing -full to propagate clock definitions through the design and compute signal arrival times at all timing points. The analysis generates flat and hierarchical power reports that decompose total power into internal, switching, and leakage components, along with a CSV file containing power breakdown by module for automated data extraction and plotting.

5.3 Performance Analysis: Spatz CC vs RedMulE

From the simulation data, three metrics were derived to characterize accelerator performance: throughput, expressed in giga-floating-point-operations (FP16) per second (GFLOPS); hardware utilization, reported as a percentage; and energy efficiency, expressed in GFLOPS/W.

These metrics provide complementary viewpoints on accelerator effectiveness: throughput captures the absolute computational performance, utilization indicates the degree to which the available hardware resources are effectively exploited, and energy efficiency quantifies the computational performance delivered per unit of power consumption. To establish a reference point for performance assessment, we first compute the theoretical peak throughput as a function of the number of multiply–accumulate (MAC) units and the operating frequency considering that 1 MAC = 2 flops. RedMulE integrates 192 MAC units operating in parallel, while Spatz CC incorporates 8 MAC units, resulting in a 24× difference in raw computational capacity that strongly influences their respective architectural properties. Assuming that each MAC performs two floating-point operations per cycle (one multiplication and one accumulation), the corresponding theoretical peak throughputs are:

$$\text{Peak}_{\text{RedMulE}} = 192 \text{ MACs/cycle} \times 2 \times 0.769 \text{ GHz} = 295.3 \text{ GFLOPS} \tag{5.1}$$

$$\text{Peak}_{\text{Spatz}} = 8 \text{ MACs/cycle} \times 2 \times 0.769 \text{ GHz} = 12.3 \text{ GFLOPS} \tag{5.2}$$

This 24× disparity in computational capacity establishes the necessary context for interpreting the subsequent performance results: a direct comparison of absolute throughput values would be misleading without explicitly accounting for the substantially different hardware resources available to each accelerator. The attained throughput was derived from the RTL simulation results by dividing the total number of floating-point operations by the measured cycle count and multiplying the result by the clock frequency:

$$\text{GFLOPS} = \frac{\text{Total Operations}}{\text{Cycles}} \times f_{\text{clk}} \tag{5.3}$$

where the total operations depend on the kernel type and problem dimensions. For multiply-accumulate kernels (MatMul, MatVec, and DotP), each MAC operation contributes two floating-point operations, while for VecSum only additions are counted:

$$\text{Total Operations}_{\text{MatMul}} = M \times N \times K \times 2 \tag{5.4}$$

$$\text{Total Operations}_{\text{MatVec}} = M \times N \times 2 \tag{5.5}$$

$$\text{Total Operations}_{\text{DotP}} = N \times 2 \tag{5.6}$$

$$\text{Total Operations}_{\text{VecSum}} = N \tag{5.7}$$

Although throughput provides an absolute measure of computational speed, it does not reveal how efficiently the underlying hardware is being exploited. To capture this insight, we define hardware utilization as the ratio of achieved MACs per cycle to the maximum theoretical MACs per cycle:

$$\text{Utilization [\%]} = \frac{\text{MACs}_{\text{achieved}}/\text{Cycles}}{\text{MACs}_{\text{max}}} \times 100 \tag{5.8}$$

where $\text{MACs}_{\text{max}} = 192$ for RedMulE and $\text{MACs}_{\text{max}} = 8$ for Spatz CC. High utilization reflects efficient exploitation of the available computational resources, whereas low utilization highlights potential performance bottlenecks.

To obtain a comprehensive characterization of performance, it is necessary to account not only for the execution speed and computational efficiency of each accelerator but also for the associated energy cost. Because the testbench operated at 200 MHz, whereas the target operating frequency is 769 MHz, the dynamic power components were scaled proportionally to the frequency ratio:

$$P_{\text{total}} = (P_{\text{internal}} + P_{\text{switching}}) \times \frac{f_{\text{target}}}{f_{\text{testbench}}} + P_{\text{leakage}} \tag{5.9}$$

where the frequency scaling factor $f_{\text{target}}/f_{\text{testbench}} = 769/200 = 3.845$ accounts for the difference between the operating points. Combining these power estimates with the measured throughput yields the energy efficiency metric:

$$\text{Efficiency} = \frac{\text{GFLOPS}}{P_{\text{total}}\,[\text{W}]} \tag{5.10}$$

which captures the performance delivered per unit of power, revealing which accelerator achieves better computational efficiency for different workload characteristics.

Experimental Results

Throughput and Hardware Utilization

Table 5.2 reports the measured throughput (GFLOPS) at 769 MHz in all 26 configurations. Figure 5.1 illustrates the corresponding throughput trends on a logarithmic scale, while Figure 5.2 shows the associated hardware utilization for the same workloads.

Kernel	Configuration	GFLOPS	
		RedMulE	Spatz CC
MatMul	[1 × 96] × [96 × 1]	0.25	0.16
	[2 × 96] × [96 × 2]	1.0	0.62
	[4 × 96] × [96 × 4]	4.0	1.6
	[8 × 96] × [96 × 8]	16.2	4.2
	[16 × 96] × [96 × 16]	64.7	10.4
	[32 × 96] × [96 × 32]	148.4	11.2
	[64 × 96] × [96 × 64]	224.0	11.4
	[96 × 96] × [96 × 96]	259.1	11.4
MatVec	[64 × 64] × [64 × 1]	6.2	5.4
	[96 × 96] × [96 × 1]	7.7	7.9
	[128 × 128] × [128 × 1]	7.1	9.2
	[256 × 256] × [256 × 1]	8.1	10.8
DotP	$N = 16$	0.13	0.12
	$N = 32$	0.19	0.25
	$N = 64$	0.27	0.42
	$N = 128$	0.29	0.84
	$N = 256$	0.34	1.4
	$N = 512$	0.36	2.0
	$N = 1024$	0.37	2.7
	$N = 2048$	0.38	3.5
VecSum	$N = 16$	0.04	0.07
	$N = 32$	0.05	0.17
	$N = 64$	0.04	0.34
	$N = 128$	0.03	0.56
	$N = 256$	0.02	0.85
	$N = 512$	0.01	1.0

Table 5.2: Throughput measurements (GFLOPS) across all configurations at 769 MHz.

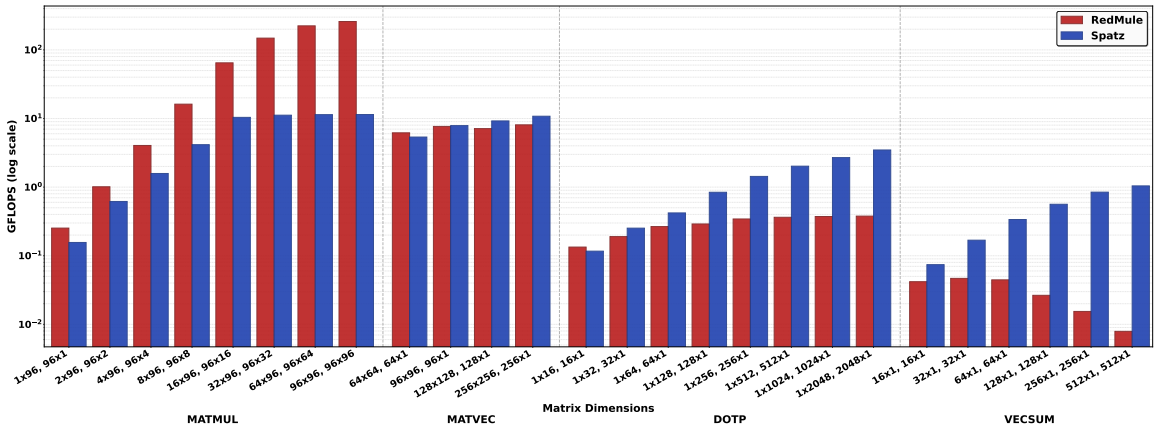

Figure 5.1: Throughput comparison across all 26 tested configurations (logarithmic scale)

The throughput results show clear workload-dependent behavior. RedMulE achieves its highest performance on matrix multiplication, reaching 259.1 GFLOPS for the $96 \times 96 \times 96$ MatMul configuration. Throughput increases consistently with matrix size, reaching 224.0 GFLOPS for $64 \times 96 \times 64$, 148.4 GFLOPS for $32 \times 96 \times 32$, and 64.7 GFLOPS for $16 \times 96 \times 16$. As shown in Figure 5.2, these configurations also correspond to the highest hardware utilization levels for RedMulE.

For lower-intensity workloads, the measured throughput decreases substantially. MatVec kernels reach 6.2–8.1 GFLOPS, while DotP increases slightly from 0.13 GFLOPS at $N = 16$ to 0.38 GFLOPS at $N = 2048$. VecSum throughput decreases from 0.04 GFLOPS to 0.01 GFLOPS across the tested sizes. The utilization trends reported in Figure 5.2 follow a similar pattern, indicating a limited use of the available computing resources.

Spatz CC exhibits a distinct scaling behavior. The MatMul kernel achieves a performance of up to 11.4 GFLOPS for matrix dimensions $\geq 32 \times 96 \times 32$, corresponding to 92.8% of its theoretical peak 12.3 GFLOPS. The MatVec kernels achieve 5.4–10.8 GFLOPS and maintain high computational unit utilization across all evaluated problem sizes, as illustrated in Figure 5.2. The DotP kernel throughput increases from 0.12 GFLOPS at $N = 16$ to 3.5 GFLOPS at $N = 2048$, whereas the VecSum kernel throughput increases from 0.07 GFLOPS to 1.0 GFLOPS over the same range.

Overall, the relative performance between the two accelerators varies significantly between kernels. RedMulE provides substantially higher throughput for large MatMul configurations (up to 22.7×), while the difference is smaller for MatVec. For DotP and VecSum workloads, Spatz CC achieves higher throughput, reaching 9.2× and 100× improvements, respectively, for the largest tested sizes.

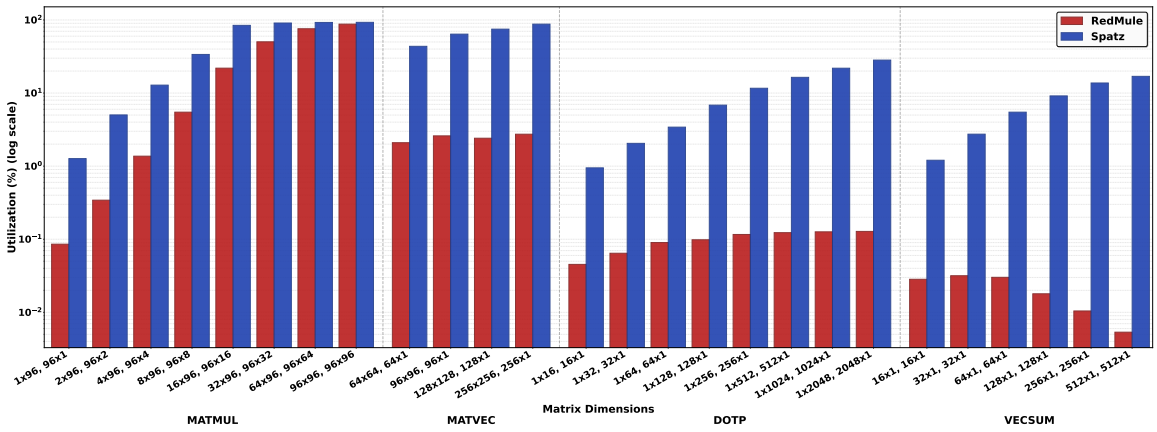

Figure 5.2: Hardware utilization comparison across all 26 tested configurations (logarithmic scale)

Energy Efficiency

Table 5.3 presents energy efficiency measurements (GFLOPS/W) in all configurations, with Figure 5.3 visualizing the performance-per-watt trends on a logarithmic scale.

Kernel	Configuration	Efficiency [GFLOPS/W]	
		RedMulE	Spatz
MatMul	[1 × 96] × [96 × 1]	1.7	5.4
	[2 × 96] × [96 × 2]	6.7	25.4
	[4 × 96] × [96 × 4]	26.4	50.3
	[8 × 96] × [96 × 8]	104.5	118.1
	[16 × 96] × [96 × 16]	407.0	252.4
	[32 × 96] × [96 × 32]	917.6	263.2
	[64 × 96] × [96 × 64]	1331.6	265.5
	[96 × 96] × [96 × 96]	1520.2	265.8
MatVec	[64 × 64] × [64 × 1]	37.2	118.4
	[96 × 96] × [96 × 1]	46.5	147.0
	[128 × 128] × [128 × 1]	42.4	159.2
	[256 × 256] × [256 × 1]	48.1	171.8
DotP	$N = 16$	1.6	4.4
	$N = 32$	2.2	9.5
	$N = 64$	2.9	15.1
	$N = 128$	3.2	28.2
	$N = 256$	3.7	43.7
	$N = 512$	3.9	54.4
	$N = 1024$	4.0	64.6
	$N = 2048$	4.0	75.8
VecSum	$N = 16$	0.29	2.8
	$N = 32$	0.29	6.2
	$N = 64$	0.27	11.3
	$N = 128$	0.16	17.0
	$N = 256$	0.09	22.5
	$N = 512$	0.05	24.6

Table 5.3: Energy efficiency (GFLOPS/W) across all configurations at 769 MHz.

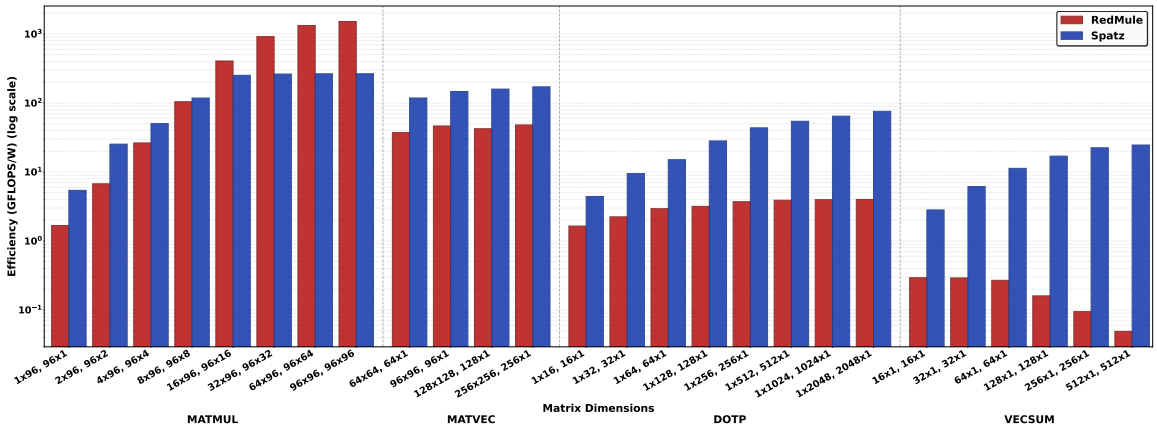

Figure 5.3: Energy efficiency comparison across all 26 tested configurations (logarithmic scale)

Energy-efficiency measurements indicate that the performance per watt closely follows both the computational intensity and hardware utilization characteristics. RedMulE achieves a maximum efficiency of 1520.2 GFLOPS/W for 96×96×96, which constitutes the highest performance per watt value among all 26 evaluated configurations. Large MatMul workloads generally sustain elevated efficiency levels, achieving 1331.6 GFLOPS/W for 64×96×64, 917.6 GFLOPS/W for 32 × 96 × 32, and 407.0 GFLOPS/W for 16 × 96 × 16. This scaling behavior is directly at-

tributable to the utilization of MAC arrays: higher utilization improves amortization of static power consumption over a larger volume of useful computation.

MatVec operations achieve only 37.2–48.1 GFLOPS/W, representing a substantial reduction relative to peak MatMul efficiency. DotP operations exhibit efficiencies between 1.6 GFLOPS/W and 4.0 GFLOPS/W with minimal dependence on vector length, suggesting that the dominant component of power consumption remains approximately constant while throughput saturates. VecSum operations yield the lowest efficiencies, decreasing from 0.29 GFLOPS/W to 0.05 GFLOPS/W, implying that static power dominates overwhelmingly and that the large MAC array provides limited benefit for these computation-light kernels.

In contrast, Spatz CC exhibits more uniform efficiency scaling across the evaluated workload classes. For MatMul, Spatz achieves 265.8 GFLOPS/W for the $96 \times 96 \times 96$ configuration, which is lower than the peak value observed for RedMulE but still indicative of high efficiency for dense matrix operations. However, for highly rectangular matrices, Spatz achieves a higher energy efficiency than RedMulE, even for MatMul workloads. MatVec workloads maintain efficiencies between 118.4 GFLOPS/W and 171.8 GFLOPS/W, substantially exceeding those of RedMulE for the same configurations. The DotP efficiency increases systematically with the length of the vector, from 4.4 GFLOPS/W for $N = 16$ to 75.8 GFLOPS/W for $N = 2048$, consistent with the corresponding performance scaling. VecSum exhibits an analogous behavior, with an efficiency ranging from 2.8 GFLOPS/W to 24.6 GFLOPS/W, and remains significantly more efficient than RedMulE in all vector lengths evaluated.

The efficiency ratio between the two accelerators is strongly dependent on workload characteristics. For the $96 \times 96 \times 96$ MatMul, RedMulE is 5.7× more energy efficient, capitalizing on its larger MAC array for highly parallel matrix computations. This relationship reverses for reduced-density kernels: for MatVec at $N = 256$, Spatz achieves 3.6× higher efficiency; for DotP at $N = 2048$, 18.9× higher; and for VecSum at $N = 512$, 492× higher.

Overall Analysis

The experimental results reveal complementary architectural specializations between the two accelerators, leading to distinct performance regimes as a function of the underlying computational pattern. RedMulE's 192 MAC, 2D systolic-style array is optimized for matrix–matrix multiplication workloads, where high computational density enables continuous activation of the available MAC units. In this regime, RedMulE achieves a peak throughput of 259.1 GFLOPS with MAC utilization of up to 87.7%, closely approaching the theoretical peak, and a maximum energy efficiency of 1520.2 GFLOPS/W. For outer product style computations, performance scales with matrix dimension; for instance, a $96 \times 96 \times 96$ MatMul configuration achieves 22.7× higher throughput and 5.7× higher energy efficiency than Spatz CC.

In MatVec workloads, RedMulE achieves only 2.1–2.7% MAC utilization, with an energy efficiency between 37.2 GFLOPS/W and 48.1 GFLOPS/W. In contrast, Spatz CC maintains substantially higher utilization (43.6–87.7%) and correspondingly higher energy efficiency

(118.4–171.8 GFLOPS/W), due to its vector-oriented architecture.

These differences are further amplified for purely vector workloads. RedMulE cannot efficiently map one-dimensional computational patterns onto its two-dimensional MAC array, resulting in extremely low utilization (below 0.13% for DotP and 0.03% for VecSum) and consequently limited throughput. Spatz CC, in contrast, is inherently well aligned with these kernels and exhibits a monotonically increasing throughput and utilization with vector length, achieving up to 217× higher throughput for DotP and 1690× for VecSum, with corresponding energy-efficiency gains of 18.9× and 492×, respectively.

In general, Spatz CC delivers more uniform performance in the evaluated workloads, with throughput ranging from 0.15 to 11.4 GFLOPS and utilization between 0.95% and 92.8%. Its architecture supports matrix workloads through iterative vector operations, while also efficiently executing native vector kernels.

Chapter 6

Conclusions

This thesis presented the architectural evolution and physical implementation of an enhanced MAGIA compute tile through the integration of a RISC-V vector coprocessor. The work required several architectural modifications, including the transition to a memory-mapped control paradigm and the introduction of an Event Unit providing hardware synchronization mechanisms. The resulting architecture was implemented in GlobalFoundries 12 nm CMOS technology, achieving an operating frequency of 769 MHz with a modest increase in the silicon area of 5.8%.

The experimental evaluation, based on cycle-accurate simulations and post-layout power analysis, highlights the complementary characteristics of the two compute engines. RedMulE achieves a maximum throughput of 259.1 GFLOPS and an energy efficiency of up to 1520 GFLOPS/W for matrix multiplication, making it highly effective for GEMM-dominated workloads. However, its efficiency decreases significantly for vector-oriented kernels, where accelerator utilization drops below 0.13% for purely vector operations.

In contrast, the Spatz Core Complex exhibits more consistent performance across a wider range of computational patterns. For matrix-vector multiplication workloads, Spatz CC achieves higher energy efficiency, ranging from 118 to 172 GFLOPS/W, compared to RedMulE's 37–48 GFLOPS/W For purely vector operations such as dot product and vector summation, Spatz CC provides substantial improvements, reaching up to 19× and 492× higher energy efficiency, respectively. These results demonstrate the suitability of the vector architecture for workloads characterized by one-dimensional data access patterns.

Overall, the results indicate that no single accelerator architecture can efficiently cover the entire spectrum of computational patterns. Instead, different types of accelerator are better suited to specific workload characteristics.

In future developments, tiles may integrate different combinations of acceleration engines depending on the target workload. A tile could implement multiple instances of the Spatz Core Complex, a combination of RedMulE and Spatz CC, or other specialized accelerators tailored to specific computational patterns.

At the system level, heterogeneity can therefore emerge across the mesh, where tiles with

different accelerator configurations coexist within the same interconnect. While the internal compute capabilities of the tiles may vary, the mesh communication infrastructure remains homogeneous, enabling scalable systems that combine specialized computation with a uniform communication substrate.

Bibliography

[1] V. Isachi, A. Nadalini, R. F. Gallotta, A. Garofalo, F. Conti, and D. Rossi, "FractalSync: Lightweight Scalable Global Synchronization of Massive Bulk Synchronous Parallel AI Accelerators," 2025.

[2] Arm Ltd., *AMBA AXI and ACE Protocol Specification*.

[3] F. Conti, G. Paulin, A. Garofalo, D. Rossi, A. Di Mauro, G. Rutishauser, G. Ottavi, M. Eggiman, H. Okuhara, and L. Benini, "Marsellus: A Heterogeneous RISC-V AI-IoT End-Node SoC With 2–8 b DNN Acceleration and 30 percent Boost Adaptive Body Biasing," *IEEE Journal of Solid-State Circuits*, vol. 59, no. 1, pp. 128–142, 2024.

[4] A. Garofalo, Y. Tortorella, M. Perotti, L. Valente, A. Nadalini, L. Benini, D. Rossi, and F. Conti, "DARKSIDE: A Heterogeneous RISC-V Compute Cluster for Extreme-Edge On-Chip DNN Inference and Training," *IEEE Open Journal of the Solid-State Circuits Society*, vol. 2, pp. 231–243, 2022.

[5] OpenHW Group, "CV32E40X User Manual: Introduction." `https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/intro.html`.

[6] OpenHW Group, *OBI Specification v1.6.0*.

[7] OpenHW Group, "CV32E40X User Manual: OpenHW XIF." `https://docs.openhwgroup.org/projects/cv32e40x-user-manual/en/latest/intro.html#openhw-xif`.

[8] Y. Tortorella, L. Bertaccini, D. Rossi, L. Benini, and F. Conti, "RedMulE: A Compact FP16 Matrix-Multiplication Accelerator for Adaptive Deep Learning on RISC-V-Based Ultra-Low-Power SoCs," in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1099–1102, 2022.

[9] M. Cavalcante, D. Wüthrich, M. Perotti, S. Riedel, and L. Benini, "Spatz: A Compact Vector Processing Unit for High-Performance and Energy-Efficient Shared-L1 Clusters," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2022.

[10] M. Perotti, S. Riedel, M. Cavalcante, and L. Benini, "Spatz: Clustering Compact RISC-V-Based Vector Units to Maximize Computing Efficiency," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, pp. 1–1, 01 2025.

[11] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads," *IEEE Transactions on Computers*, vol. 70, no. 11, pp. 1845–1860, 2021.

[12] PULP Platform, "RedMulE Repository." `https://github.com/pulp-platform/redmule`.

[13] PULP Platform, *event_unit_flex Reference Manual*.

[14] F. Glaser, G. Tagliavini, D. Rossi, G. Haugou, Q. Huang, and L. Benini, "Energy-Efficient Hardware-Accelerated Synchronization for Shared-L1-Memory Multiprocessor Clusters," 2020.