

Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electronic Engineering (38^{th}cycle)

Study and Implementation of CGRA architectures for edge applications

By

Alessio Naclerio

******

Supervisor(s):
Prof.ssa Mariagrazia Graziano, Supervisor
Prof. Fabrizio Riente, Co-Supervisor
Prof. Maurizio Zamboni, Co-Supervisor

Doctoral Examination Committee:
Prof. Andrés Otero, Referee, Universidad Politécnica de Madrid (UPM)
Prof. Marco Ottavi, Referee, University of Twente

Politecnico di Torino
2026

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Alessio Naclerio
2026

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Contents

List of Figures vi

List of Tables xii

Nomenclature xv

1 Introduction 1

I Background & Motivations 6

2 Background 8

2.1 CGRAs in the compute landscape 8

2.2 Architectural Features of CGRAs 11

2.2.1 Processing Element Array 13

2.2.2 Configuration Storage Subsystem 22

2.2.3 Data Storage Subsystem 24

2.2.4 DAE Paradigm & Address Generation Unit 25

2.2.5 System Integration 28

2.2.6 Computation Model 30

2.2.7 Code Coverage & Target Applications 33

2.3 Considerations & Motivations 34

2.4 The X-HEEP Platform 38
2.4.1 Platform Configurability 39
2.4.2 Platform Extensibility 41

II Mage and Rogue CGRAs 43

3 Introduction & Summary 45

4 Mage: A DAE CGRA Tailored for Static Control Applications 47
4.1 Motivations and Related Works 47
4.2 Mage architecture 50
4.2.1 Multi-Bank ScratchPad Memory Subsystem 51
4.2.2 Address Generation Unit 52
4.2.3 Processing Element Array 57
4.2.4 Crossbars and Access-Execute Channels 65
4.3 Kernel Mapping and Scheduling Methodology 68
4.4 Results & Discussion 74
4.4.1 Mage Implementations 74
4.4.2 Experimental setup 76
4.4.3 Evaluation of Area Occupation and Scaling 78
4.4.4 Benchmarks Evaluation 83
4.4.5 Accelerating Temporal Convolutional Networks with Mage . 110
4.4.6 Comparison with State-of-the-Art CGRAs 116
4.5 Conclusions 121

5 Rogue: A Streaming DAE CGRA for Integer Activation Approximation 124
5.1 Introduction & Motivations 124
5.2 Related Works 126

5.3 System Integration of Streaming Accelerators 129
5.3.1 DMA-Coupled Accelerator Interface in X-HEEP 129
5.3.2 HEEPidermis & Digital Level-Crossing Module 132
5.4 Proposed Instruction Set Architecture 135
5.5 Rogue Architecture 138
5.5.1 Data Stream Management Unit 139
5.5.2 Processing Element Array 145
5.6 Workloads Mapping & Execution Model 152
5.7 Results and Discussion 157
5.7.1 Rogue Implementations 157
5.7.2 Experimental Setup 159
5.7.3 Area Evaluation - *s*, *p* & *g* Variants 163
5.7.4 Performance Evaluation - *s* & *p* Rogue Variants 165
5.7.5 Energy Efficiency Evaluation - *s* & *p* Rogue Variants 169
5.7.6 Performance Evaluation - *g* Rogue Variants 171
5.7.7 Energy & Area Efficiency Evaluation - *g* Rogue Variants . . 174
5.7.8 Comparison with State-of-the-Art Accelerators & CGRAs . 177
5.8 Rogue Integration in the polHEEPo SoC 189
5.9 Conclusions 189

6 Conclusions & Future Works **193**

References **198**

List of Figures

2.1 CGRA in the compute landscape, comparing it with other processing architectures regarding flexibility, performance and energy efficiency. 10

2.2 Overview of the main components found in CGRA designs. 13

2.3 Overview of possible CGRA's Processing Element Arrays (PEA) implementations. On the left, an R×C array of PEs with N2N, diagonal and torus connections. On the right, an R×C array of PEs with Switch Blocks for routing. 14

2.4 Overview of a CGRA's Processing Element (PE) internal structure, along with its internal components, namely Functional Unit (FU), Control Subsystem (CS), Input Multiplexers and I/O Memory Elements, indicated as IB_i (Input Buffers) and OB_i (Output Buffers). . . 15

2.5 Configuration Storage Subsystem (CSS) and Data Storage Subsystem (DSS) and their internal components. 23

2.6 Overview of the Decoupled Access-Execute (DAE) paradigm implementation on a CGRA. The *Execute* unit is implemented by the PEA, the *Access* unit is implemented by the AGU and a Scratchpad Memory is interposed between the two to be accessed by the AGU and provide and retrieve data from the PEA. 26

2.7 Comparison of II across several workloads on ideal standard CGRAs (light brown bars) and ideal DAE CGRA (light yellow bars). Data is taken from [1]. 27

2.8 Address Generation Instructions (AGIs) percentage across various workloads when either a single-bank data memory is used (light brown bars) and multi-bank data memory is employed (light yellow bars). Data is taken from [1]. 28

2.9 Data layout inside a scratchpad memory based on the Generalize Memory Partitioning (GMP), with $N_b = 4$ and $B_s = 2$. 29

2.10 The loop DFG on the left is modulo scheduled with $II = 1$ on the 2×2 PEA on the right. In the considered case $RecMII = 1$ as no loop-carried dependencies are present, and $ResMII = \frac{4}{4} = 1$, hence the kernel can be mapped with $II = max(ResMII, RecMII) = 1$. Therefore, the *kernel* is composed of a set of instructions that span through a single cycle. 31

2.11 The loop DFG on the left is modulo scheduled with $II = 2$ on the 2×2 PEA on the right. In the considered case $RecMII = 1$ as no loop-carried dependencies are present, and $ResMII = \lceil \frac{6}{4} \rceil = 2$, hence the kernel can be mapped with $II = max(ResMII, RecMII) = 2$. Therefore, the *kernel* is composed of a set of instructions that span through two consecutive cycles. 32

2.12 X-HEEP. 39

4.1 Overview of Mage template-based architecture encompassing PEA, AGU, SpM and crossbars. 50

4.2 Mage's AGU and its internal components, namely HLU, HLRF, HLDU and AGEs. Template parameters and configuration registers related to the various module are also reported, accordingly to Figure 4.4 and Figure 4.3. 53

4.3 Template parameters to define the structure of the AGU. 56

4.4 Configuration registers of Mage used by the AGU. 57

4.5 Mage's PEA template-based structure. Template parameters and configuration registers related to the various module are also reported, accordingly to Figure 4.7 and Figure 4.6. 58

4.6 Configuration registers of Mage used by the PEA and PEs. 64

4.7 Template parameters to define the structure of the PEA. 65

4.8 Mage AGU→SpM and SpM↔PEA template-based structure. For better understanding, *n_ch* and *n_ages* have been set to two in the reported illustration. Each AGE and bank has two subscripts, the first indicating the channels it belongs to, and the latter indicating its index within the channel. For the Mage implementation reported in figure, other two channels are present on the right side, but they are not shown for better clarity. Template parameters and configuration registers related to the various module are also reported, accordingly to Figure 4.10 and Figure 4.9. Referring to subsection 4.4.1, the reported Mage variant is *4x4_4x2*, without the two right-side channels. 66

4.9 Configuration registers of Mage used by the SpM ↔ PEA crossbar. 68

4.10 Template parameters to define the structure of the SpM ↔ PEA and AGU → SpM crossbars. 68

4.11 *matmul* DFG with $UF = 2$ along with the related ALAP scheduling and kernel identification. 71

4.12 Mapping of *matmul* DFG with $UF = 2$ on a Mage implementation with 2 non-homogeneous access-execute channels, in the mage variant *3x2_2x2_2x1* presented in subsection 4.4.1. 73

4.13 Area breakdown of Mage **5x2_2x4_2x1_v** throughout scaling of memory banks from 4KiB to 32KiB each. 79

4.14 Area breakdown of Mage **4x2_2x4_s**. 81

4.15 Area breakdown of Mage **4x2_2x4_v**. 82

4.16 Speedup scaling of scalar Mage across input sizes and unroll factors. X-axis labels report the main kernel dimensions: for *atax*, *bicg*, *firstdiff*, and *hydro* the label is $N \times M$, where N and M are the number of iterations for the two loops used to describe the kernel. For *conv1d*, it is $L \times K$, where L is the signal length and k the kernel size. For *conv2d* it is $H \times W \times K$, where H is the input matrix height, W the input matrix width, and K the filter height and width. For *gemm*, it is $N \times K \times M$, where N and M are the number of rows and columns of the output matrix, and K is the number of columns of the input matrix. 86

4.17 MOPS scaling of scalar Mage across input sizes and unroll factors. X-axis labels report the main kernel dimensions, explained in Figure 4.17. 89

4.18 Energy scaling of **scalar** Mage across input sizes and UF. X-axis labels report the main kernel dimensions, explained in Figure 4.16. . 92

4.19 Speedup scaling of scalar and vector Mage across input sizes and unroll factors for *matmul*. X-axis labels report the main kernel dimensions $N \times K \times M$, where N and M are the number of rows and columns of the output matrix, and K is the number of columns of the input matrix. 96

4.20 Speedup scaling of scalar and vector Mage across input sizes and unroll factors for *dconv1d*. X-axis labels report the main kernel dimensions $Ch_{out} \times Ch_{in} \times T \times K$, where Ch_{out} and Ch_{out} are the number input and output channels, respectively, T is the length of the time sequence and K is the 1D filter size. 98

4.21 MOPS scaling of scalar Mage across input sizes and unroll factors for *matmul*. X-axis labels report the main kernel dimensions, explained in Figure 4.19. 99

4.22 MOPS scaling of scalar Mage across input sizes and unroll factors for *dconv1d*. X-axis labels report the main kernel dimensions, explained in Figure 4.20. 100

4.23 Execution and configuration cycles for *matmul* across various UFs in `int8` 101

4.24 Scaling of energy savings on *matmul* for scalar and vector Mage 5x2_2x4_2x1 variants across input sizes and unroll factors. The savings against the *unoptimized* baseline are achieved by the scalar variant, while the savings against the *scalar* and *simd* ones are obtained by the vector implementation. 106

4.25 Scaling of energy savings on *dconv1d* for scalar and vector Mage 5x2_2x4_2x1 variants across input sizes and unroll factors. The savings against the *unoptimized* baseline are achieved by the scalar variant, while the savings against the *scalar* and *simd* ones are obtained by the vector implementation. 107

4.26 Block diagram of the TCN pipeline considered for Mage-based acceleration. 112

5.1 (a) Heepidermis SoC picture on a finger. (b) Heepidermis SoC picture taken using a microscope. (c) Heepidermis SoC layout view in which the portion highlighted in red in the dLC module. (d) dlC area occupation within the Heepidermis layout. 132

5.2 Overall template-based architecture of Rogue encompassing PEA and DSMU. 138

5.3 Rogue's DSMU template-based structure, along with template parameters and configuration registers identified with the same "section" titles used in Figure 5.5 and Figure 5.4, respectively. 140

5.4 Configuration registers of Rogue used by the DSMU. 143

5.5 Template parameters to define the internal structure of the DSMU. . 144

5.6 Rogue's PEA template-based structure, along with template parameters and configuration registers identified with the same "section" titles used in Figure 5.8 and Figure 5.7, respectively. 145

5.7 Configuration registers of Rogue used by the PEA and internally to PEs. 151

5.8 Template parameters to define the structure of the PEA. 152

5.9 Rewriting of *GELU* using Rogue's ISA, its DFG and scheduled DFG, along with the relative mapping onto a 4×2-PEA Rogue. 154

5.10 Throughput scaling for *S* and *P* Rogue implementations over growing input sequence length. 165

5.11 Scaling of speedup over execution on the CV32E40P implementing the *XCVPULP* ISA extensions for *S* and *P* Rogue implementations over growing input sequence length. 168

5.12 Scaling of energy savings over execution on the CV32E40P implementing the *XCVPULP* ISA extensions for *S* and *P* Rogue implementations over growing input sequence length. 170

5.13 Scaling of MOPS on *matmul* for g Rogue implementations over growing input matrices. The sizes are $N \times K \times M$, where N and K are rows and columns of the first input matrix, and K and M rows and columns of the second. 172

5.14 Scaling of speedup compared to *CV32E40P* on *matmul* for G Rogue implementations over growing input matrices. 173

5.15 Scaling of energy savings compared to *CV32E40P* on *matmul* for G Rogue implementations over growing input matrices. 175

5.16 Scaling of energy efficiency (MOPS/mW) on *matmul* for G Rogue implementations over growing input matrices. 175

5.17 Scaling of area efficiency (MOPS/mm^2) on *matmul* for G Rogue implementations over growing input matrices. 177

5.18 On the left, polHEEPo's layout is shown, while, on the right, the polHEEPo chip is shown mounted on a temporary board for testing its functionalities, waiting for the custom boards to be shipped. . . . 188

List of Tables

2.1 Kernel classification across CGRA architectures. For each architecture, the number of kernels (**N**) and the corresponding percentage over the total supported workloads (**%**) are reported for each kernel class. Kernel classes are SCoP (Static Control Programs), IA (codes containing Indirect Addressing), CC (codes containing Complex Control) and SC (codes containing Simple Control). 35

4.1 Instructions supported by Mage PEs, along with their opcode and vector mode in the employed ISA and their function. Although not related to the ISA by definition, latencies are also reported for the different PE variants Mage implements. 60

4.2 Benchmarked SCoP kernels used to evaluate Mage. 77

4.3 Mage area breakdown across **scalar** and **vector** variants (unit: $\mu m^2 \times 10^3$). 80

4.4 Achievable UFs on Mage architectures and AGU and PEA utilization percentages. 84

4.5 Maximum speedup obtained by Mage per kernel and unroll factor (UF). Execution cycles are also reported for both Mage and the CV32E40P CPU. 85

4.6 Maximum MOPS obtained by Mage @250MHz per kernel and unroll factor (UF). Mage cycles and the number of operations at the maximum MOPS point are also reported. 88

4.7 Maximum area efficiency (MOPS/mm^2) for each kernels and UF on **scalar** Mage architectures. 90

4.8 Maximum energy savings obtained by **scalar** Mage per kernel and UF compared to the *scalar* CPU baseline. 92

4.9 Peak MOPS/mW achieved by **scalar** Mage architectures for kernels and related UFs. 94

4.10 Achievable UFs on Mage architectures for *matmul* and *dconv1d*, along with AGU and PEA utilization percentages. 95

4.11 Peak MOPS/mm^2 for *matmul* and *dconv1d* on **scalar** Mage architectures. 102

4.12 Peak MOPS/mm^2 for *matmul* and *dconv1d* on **vector** Mage architectures. 103

4.13 Maximum energy saving for **scalar** Mage architectures. Results are reported for *matmul* and *dconv1d*, compared to the *unoptimized* CPU baseline. 104

4.14 Maximum energy saving for **vector** Mage architectures. Results are reported for *matmul* and *dconv1d*, compared to the *scalar* and *simd* baselines for `int32`, `int16`, and `int8`. 105

4.15 Power breakdown of Mage for *dconv1d* on *5x2_2x4_2x1_v* across various UFs for `int32` and at maximum UF for `int16` and *int8*. Absolute power values are reported in mW. 108

4.16 Peak MOPS/mW for **scalar** Mage architectures for *matmul* and *dconv1d*. 109

4.17 Peak MOPS/mW for **vector** Mage architectures for *matmul* and *dconv1d*. 110

4.18 Comparison between Mage and other state-of-the-art DAE CGRAs. 117

4.19 Comparison between Mage and other state-of-the-art CGRAs. . . . 120

5.1 Description of proposed ISA instructions. The latency of Rogue's implementation is also reported. 136

5.2 Workloads used to evaluate Rogue on integer approximations of activations and GeMM kernels. 160

5.3 For each **kernel** and related **UF**, one or multiple **DMA Transfer** may be needed. For each Rogue variant, the number of DMA channels used to implement the related workload on the specified UF in the indicated DMA transfer is indicated (in the related cell within the variant column). As for *matmul*, *N* is the number of rows of the first input matrix. 161

5.4 Area breakdown of *S* and *P* Rogue variants and related X-HEEP systems ($\times 10^3$ μm^2) in 16 nm technology at 330 MHz. 163

5.5 Area breakdown of Rogue *s*, *p* and *g* variants and related X-HEEP systems ($\times 10^3$ μm^2) in 65 nm technology at 250 MHz. 164

5.6 Comparison with State-of-the-Art Accelerators. 178

5.7 Technology, area, and power comparison between Rogue and Mage. 181

5.8 Performance and efficiency comparison between Rogue and Mage. . 182

5.9 Comparison between Rogue and other state-of-the-art DAE CGRAs. 184

5.10 Technology, area, and power comparison between Rogue and other state-of-the-art CGRAs. 185

5.11 Performance and efficiency comparison between Rogue and other state-of-the-art CGRAs. 186

Nomenclature

Acronyms / Abbreviations

AGI Address Generation Instruction

AGU Address Generation Unit

AI Artificial Intelligence

ASIC Application-Specific Integrated Circuit

BU Branch Unit

CGP Cartesian Genetic Programming

CGRA Coarse-Grain Reconfigurable Array

CNN Convolutional Neural Network

CPU Central Processing Unit

CRF Configuration Register File

CS Control Subsystem

CSS Configuration Storage Subsystem

DCoP Dynamic Control Program

DFG Data Flow Graph

DISE Dual-Issue Single-Execution

DMA Direct Memory Access

DOO Delay-Operand Overlay

DRF Data Register File

DSE Design Space Exploration

DSMU Data Stream Management Unit

DSP Digital Signal Processor

DSS Data Storage Subsystem

FIFO First In First Out

FP Full Predication

FP floating-point

GELU Gaussian Error Linear Unit

GeMM General Matrix Multiplication

GMP Generalize Memory Partitioning

GPU Graphics Processing Unit

HPC High Performance Computing

I/O Input/Output

II Initiation Interval

IMs Input Multiplexers

INT integer

IP Intellectual Property

ISA Instruction Set Architecture

IV Induction Variable

LSTM Long-Short Term Memory

LSU Load-Store Unit

LUT Look-Up Table

MAE Mean Average Error

MII Minimum Initiation Interval

ML Machine Learning

N2N Neighbor-to-Neighbor

OBI Open Bus Interface

OpC Output-per-Cycle

PA Polynomial Approximation

PC Program Counter

PE Processing Element

PEA Processing Element Array

PP Partial Predication

PWLA Piece-Wise Linear Approximation

PWPA Piece-Wise Polynomial Approximation

RecMII Recurrence Minimum Initiation Interval

ReLU Rectified Linear Unit

ResMII Resource Minimum Initiation Interval

RMSE Root Mean Squared Error

SB Switch Block

SCoP Static Control Program

SFP State-based Full Predication

SiLU Sigmoid Linear Unit

SIMD Single-Instruction Multiple-Data

SIMT Single-Instruction Multi-Thread

SoC System on Chip

SpM Scratchpad Memory

Tanh Hyperbolic Tangent

ViT Vision Transformer

Chapter 1

Introduction

The growing deployment of data-intensive applications on edge platforms is increasingly requiring hardware designs to deliver high performance along with strong energy efficiency under tight area constraints, while still preserving enough flexibility to adapt to evolving workloads. Although general-purpose processors remain attractive due to their high programmability and mature software ecosystems, their instruction-based execution model as well as their memory management overheads often result in limiting both throughput and energy efficiency on regular, compute-heavy kernels. On the other hand, Application-Specific Integrated Circuits (ASICs) usually achieve high performance and efficiency, but at the cost of specializing their implementation to a narrow workload subset, thus significantly reducing versatility and increasing design cost. In this landscape, **Coarse-Grain Reconfigurable Arrays** (*CGRAs*) represent an attractive middle design point, as they effectively combine spatial parallelism and reconfigurability in order to approach ASIC-like efficiency on structured kernels while also retaining significantly more flexibility.

A CGRA is typically organized as an array of *Processing Elements* (PEs), namely coarse-grain computational units interconnected through a configurable communication fabric. Each PE usually supports a set of arithmetic and logical operations, while the interconnect allows data to be routed across the array according to the requirements of the mapped kernel. Compared to instruction-driven architectures such as CPUs, CGRA's execution model reduces control overheads and favors efficient data movement and reuse. At the same time, unlike ASICs, the array can be reconfigured to support different kernels, thus preserving a practical degree of programmability.

The first approaches that can be assimilated to the CGRA trace back to the late 1990s, and early CGRAs were primarily conceived for structured workloads, usually expressed as single loops, with the main focus placed on data-related computation rather than control. This focus was not incidental, as CGRA architectures and execution models deliver their greatest benefits when parallelism and data movement can be exposed and scheduled statically, so that the array operates in a steady-state, high-throughput regime. Nonetheless, *recent research on CGRAs is increasingly exploring architectures supporting broader code coverage*, especially through support for **Dynamic-Control Programs** (DCoPs) including irregular memory accesses and dynamic data-dependent control flow. While this trend is valuable from an applicability standpoint, it also introduces additional hardware mechanisms for control handling, synchronization, and memory management that can degrade throughput, area efficiency, and energy efficiency on the structured kernels for which CGRAs are most effective. In addition, the analysis carried out in this thesis shows that the majority of workloads accelerated by recent CGRAs still falls within the class of **Static Control Programs** (SCoPs), namely loop-based kernels with affine memory accesses and compile-time predictable control flow. This is not a coincidence, as **SCoPs encompass a large fraction of the computational kernels that are most relevant in many edge-application domains**, including Machine Learning, Signal and Image Processing, and Hardware Security. This suggests that an important opportunity remains underexplored. While CGRA research has progressively moved toward supporting increasingly general program regions, the workloads that most consistently benefit from acceleration are still largely regular and compute-intensive rather than control-intensive. Therefore, the cost of supporting broader dynamic behavior may not always be justified, especially when it degrades the efficiency attainable on the structured kernels that continue to dominate edge-oriented acceleration scenarios. For this reason, explicitly designing CGRAs with SCoPs as the primary target represents a highly relevant direction, as it enables the architecture to unlock its full potential on a large portion of workloads in crucial edge application fields.

In this context, the **Decoupled Access-Execute** (*DAE*) paradigm provides a particularly suitable architectural bridge between the regularity of SCoP workloads and the design direction pursued by the CGRA architectures proposed in this thesis. In the DAE model, memory access and address generation are delegated to dedicated hardware units, while the computational core is reserved for arithmetic and

data-related operations. This separation is especially effective for SCoP kernels, since their memory accesses can be expressed as affine functions of loop induction variables and can therefore be generated efficiently in hardware. In the CGRA context, this means that address generation instructions do not consume computational resources within the array, but are instead offloaded to specialized units dedicated to address computation. As a result, pressure on the compute fabric is significantly reduced, leaving more resources available for arithmetic operations and increasing the opportunity to exploit parallelism. Therefore, combining the DAE model with an explicit focus on SCoP workloads enables not only higher performance, but also improved area efficiency, since both the computational core and the address-generation subsystem can be specialized for their respective tasks. Moreover, when coupled with a specialized scratchpad memory organization and data layout, the DAE model can also allow for high-bandwidth, conflict-free data delivery to the array, further improving both throughput and energy efficiency.

Building on these considerations, this thesis explores two complementary DAE CGRA architectures for edge applications, both integrated within the open-source *X-HEEP platform* [2, 3].

The first proposed architecture is **Mage**, a DAE CGRA explicitly tailored for SCoP workloads. Mage combines a dedicated Address Generation Unit (AGU), capable of computing up to 4D affine access patterns, with an internal multi-bank Scratchpad Memory (SpM), whose data layout is constrained to be compliant with a version of the Generalized Memory Partitioning scheme proposed in [4]. By following this layout, the address generators can access the internal SpM without incurring bank conflicts, thus allowing the PEA to receive data at maximum bandwidth while focusing exclusively on arithmetic operations, without the need to handle either control or address generation. By tailoring its DAE CGRA architecture to SCoP workloads, Mage achieves the highest throughput and area efficiency among the considered state-of-the-art standard and DAE CGRAs on SCoP kernels, while also attaining comparable or higher energy efficiency, reaching up to **7.8 GOPS**, **13.8 GOPS/mm**^{2}, and **345.6 MOPS/mW** on `int8` matrix multiplication. In addition, Mage achieves up to **50**$\times$ speedup and **18**$\times$ energy savings compared to manually optimized `C` code running on the CV32E40P RISC-V CPU with *XCVPULP* extensions, on an `int8` piecewise dilated 1D convolution, a key workload in *Temporal Convolutional Networks* (TCNs). These kernel-level gains also translate to end-to-end applications, as Mage delivers overall speedups of up to **12.9**$\times$ on an `int16`

TCN pipeline and **9.7**× on an `int8` one. Overall, Mage shows that combining the DAE model with a tailored data layout for conflict-free memory accesses enables high throughput together with strong area and energy efficiency on SCoP workloads.

The second proposed architecture is **Rogue**, which explores a lighter, streaming-oriented DAE organization. Rather than integrating a full internal memory subsystem, Rogue relies on a FIFO-based interface that allows it to connect to external Load-Store Units or system DMAs, so that data can be streamed directly from main memory to the array of PEs. In addition, Rogue implements a dedicated Instruction Set Architecture (ISA) tailored to integer approximations of activation functions, while still supporting conventional GeMM-style computation. This makes Rogue particularly suited to Machine Learning pipelines, where both activation functions and General Matrix Multiplication (GeMM) kernels play a central role and still fall within the broader class of SCoP workloads. Compared to Mage, Rogue explores a complementary point in the DAE CGRA design space. While it does not rely on a tailored internal data layout and cannot eliminate memory-access conflicts as Mage, it incurs a much lower system-level area overhead while still delivering substantial throughput and efficiency gains. On integer activation kernels, Rogue achieves up to **98.03**× **speedup** on *Sigmoid*, **92.95**× on *Tanh*, and about **48**× **energy savings** on *GELU* over software execution on the CV32E40P implementing the *XCVPULP* extensions and running compiler-generated code. Compared to activation-specific accelerators, Rogue still delivers competitive throughput while retaining a significantly higher degree of flexibility. Rogue also attains significant results on GeMM workloads, reaching up to **932 MOPS**, **67.8 MOPS/mW**, and **6522.6 MOPS/mm**^{2} on `int32` matrix multiplication, while incurring only limited system-level area overhead, up to just **12.2%**. Overall, Rogue proves highly competitive with state-of-the-art standard and DAE CGRAs in terms of throughput and efficiency, while offering a more compact and integration-friendly streaming organization. Furthermore, Rogue has also recently been integrated into the **polHEEPo** SoC, an X-HEEP-based chip taped out in TSMC 65 nm technology, where it operates as a DMA-coupled streaming accelerator for integer activations. Overall, Mage and Rogue do not represent alternative implementations of the same idea, but rather two complementary specializations of the DAE CGRA paradigm. Mage targets maximum throughput on structured kernels through conflict-free memory accesses, whereas Rogue favors a more compact implementation incurring lower system-level area overheads, while still delivering substantial gains on relevant edge workloads.

The remainder of this thesis is organized as follows. Chapter 2 reviews the CGRA design space, discussing architectural features, computation models, integration strategies, and workload classes, and explains the motivations behind the work carried out in this thesis. Chapter 3 introduces Mage and Rogue and summarizes their positioning in the DAE CGRA design space. Chapter 4 presents Mage, including its architecture, mapping methodology, design-space exploration, and comparison with state-of-the-art CGRAs. Chapter 5 presents Rogue, including its DMA-coupled integration model, dedicated ISA, architectural organization, evaluation on activation and GeMM kernels, and comparison with both activation accelerators and other CGRAs. Finally, chapter 6 concludes the thesis and discusses possible future work.

Part I

Background & Motivations

Chapter 2

Background

2.1 CGRAs in the compute landscape

In recent years, increasingly tighter power budgets in popular and emerging edge applications have driven the demand for integrated circuits that can deliver high performance along with substantial energy efficiency, while retaining as much programming flexibility as possible.

Modern computing platforms span a broad design space defined by *flexibility*, *performance*, and *energy efficiency*. At one end, Central Processing Units (CPUs) provide the greatest programmability thanks to mature software ecosystems and high-level languages. However, their general-purpose micro-architectures introduce control and data-movement overheads that constrain both performance and energy efficiency. This is particularly true for nowadays pervasive data-intensive applications, in which CPUs largely struggle to fully exploit parallelism and data locality. At the opposite end, Application-Specific Integrated Circuits (ASICs) can achieve the highest performance and energy efficiency by entirely sacrificing flexibility. Nonetheless, since they are tailored to a single application, they cannot adapt to evolving workloads, which shortens their effective lifetime and increases non-recurring engineering costs, a drawback that is amplified when considering expensive advanced technology nodes.

Between these extremes, several widely used alternatives are available. Graphics Processing Units (GPUs) combine high-level programmability with high throughput by leveraging *Single-Instruction Multi-Thread (SIMT)* execution and high-bandwidth

memory systems, which makes them effective for regular, data-parallel kernels. However, this capability, comes with notable area and energy overheads due to complex hardware mechanisms, which may also turn out to be unnecessary when workloads have simple control flow and predictable memory access patterns. Digital Signal Processors (DSPs) similarly offer substantial flexibility and strong performance within their intended domains, but they often fall short in energy efficiency. Reconfigurable architectures occupy a middle ground in this landscape. *Field-Programmable Gate Arrays (FPGAs)* provide extensive post-fabrication flexibility through fine-grained configurability, enabling highly customized datapaths that can match or exceed the performance of GPUs and DSPs while operating within tighter power budgets. However, bit-level programmability comes with substantial costs: extensive routing overheads increase area and complicate timing closure, which can reduce achievable clock frequency, and compilation and reconfiguration times remain significant.

In this scenario **Coarse Grain Reconfigurable Arrays (CGRAs)** aim for a more targeted compromise, as illustrated in Figure 2.1. CGRAs provide enough programmability to keep up with evolving kernels, but at a coarser granularity than FPGAs, typically at the operator level. This choice reduces the overheads of fine-grained configuration and routing, improves performance and area efficiency compared to FPGAs, and enables ASIC-like efficiency on structured kernels while preserving a practical degree of programmability. To this end, CGRAs are commonly organized as an array of Processing Elements (PEs) equipped with coarse-grain operators, such as adders, multipliers, and shifters, interconnected by a reconfigurable routing overlay. Each PE executes a local operation on data streams arriving from memory or from neighboring PEs, and the array can be reconfigured at run time, often up to a cycle-by-cycle basis. This organization enables mapping computation across both the *spatial* and the *temporal* dimension. Hence, each operation is assigned a spatial location (the PE that performs it) and, when resources are time-multiplexed, a temporal slot (the cycle or configuration context in which it is executed). As a result, a CGRA can sustain a steady-state, pipelined execution over many iterations of a kernel, particularly when the computation is fed by continuous input/output streams. Compared to instruction-based circuits like CPUs, GPUs and DSPs, this execution style reduces instruction-related overheads, exposes parallelism naturally across the array, and promotes efficient data movement and reuse, eventually approaching ASIC-like energy efficiency on the targeted class of structured workloads.

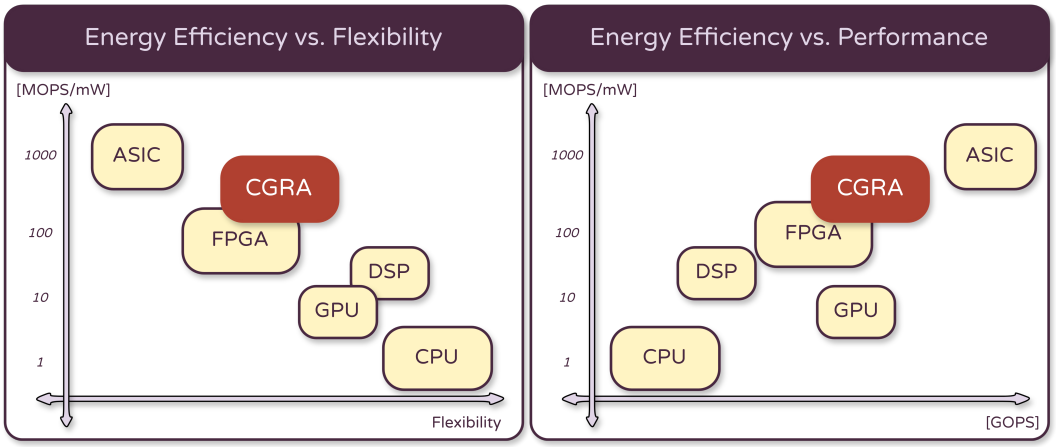


Fig. 2.1 CGRA in the compute landscape, comparing it with other processing architectures regarding flexibility, performance and energy efficiency.

Moreover, and as an anticipation of the key motivations for the architectural choices explored in this thesis, **the benefits of the CGRA computational model are most evident for a well-defined class of workloads**. These workloads are typically expressed as loop nests with limited data-dependent control flow, and are dominated by computations over data streams with regular, predictable memory access patterns. In this scenario, predictable memory access and reuse patterns allow organizing data movement explicitly so that the CGRA primarily spends execution time in a high-throughput, steady-state regime, thus maximizing both performance and energy efficiency. On the other hand, workloads with irregular, data-dependent control flow or memory accesses usually tend to limit these advantages. In addition, supporting such behavior often requires additional runtime flexibility or micro-architectural mechanisms within the array that result in area and energy overheads. At the same time, support for these workloads usually lead to complicate kernel mapping and scheduling solution, thus causing poor array utilization and occasional stalls. As a result, both performance and energy efficiency can degrade, undermining the very motivation for offloading to a CGRA.

Finally, it is worth highlighting that CGRAs have been extensively investigated in both the High-Performance Computing (HPC) and the embedded-systems domains. Nevertheless, their implementation choices often differ significantly, as the dominant constraints are not the same. In HPC, CGRA designs commonly scale toward large PE arrays, adopt scalable interconnects, and rely on high-bandwidth memory

solutions, with the goal of maximizing peak parallel throughput and sustaining continuous data streaming. In embedded and edge platforms, instead, the primary constraints are power and area budgets, together with the complexity of integrating accelerators within a heterogeneous SoC. As a consequence, the most attractive CGRA instances are typically more compact (often small arrays, e.g. 4×4) and are designed to maximize performance-per-area and energy efficiency while keeping the hardware/software interface and the software stack manageable. As regards this thesis, the proposed CGRA designs target the embedded design point, and therefore adopt architectural choices are taken in this direction.

Overview and Organization

CGRAs span a broad design space, and many different implementations have been proposed in literature. Therefore, a clear classification is essential to enable comparison across prior works and, more importantly, to provide motivations for the design decisions adopted in this thesis. In the following sections, it is tried to organize the CGRA landscape three complementary axes. First, the main ***architectural features*** *are examined in section 2.2, along with their impact on achievable performance, area efficiency, and energy efficiency. Second, the* ***computation models*** *commonly adopted in CGRAs are introduced in subsection 2.2.6, focusing on how workloads are mapped onto the PE array across spatial and temporal dimensions and how execution is carried out. Third, trends in* ***target applications and code coverage*** *are outlined in subsection 2.2.7, since the supported workload classes strongly influence architectural evolution and provide insight into the most promising directions for future CGRA designs.*

2.2 Architectural Features of CGRAs

This section introduces the architectural features that commonly characterize CGRAs and summarizes the motivations behind the main implementation choices found in the literature. Despite the diversity of existing proposals, as shown in Figure 2.2 most CGRAs can be described in terms of four fundamental building blocks:

- a **Processing Element Array** (PEA), which constitutes the computational fabric of the architecture, and it mainly relies on an array of reconfigurable **Processing Elements** (PEs). It performs all arithmetic and logical operations on data that define the Instruction Set Architecture (ISA) supported by the CGRA. Moreover, depending on the architecture, the PEA may also support memory load/store operations and control-flow handling functionalities.

- A **Configuration Storage Subsystem** (CSS), which holds the configuration words that define PE functionality and interconnect behavior. This storage subsystem can be implemented using SRAM macros (either within the CGRA or external) or through distributed local register files embedded within each PE.

- A **Data Storage Subsystem** (DSS), which provides the operands consumed by the PEA and collects produced results. This subsystem may be realized as dedicated internal scratchpad SRAMs, or may rely on shared memories that are external to the CGRA and managed at the system level, depending on the integration model. Moreover, in the proposed classification, this subsystem also includes possible register files that are local to PEs and are used to store several intermediate results for reuse.

- Although not present in most "standard" CGRA designs, the **Address Generation Unit (AGU)** can be considered a fourth building block, as its adoption is becoming increasingly common. This hardware unit is specialized in address generation and facilitates access to the DSS. The AGU is particularly employed in designs that follow the **Decoupled Access-Execute** computational paradigm, where memory access and data-related computations are handled by separate hardware units.

Furthermore, in this section, another important classification characteristic is considered. Although it is not directly included in the architectural features, **system integration**, namely the way the CGRA is integrated into the host system and how it interacts with other components, significantly influences CGRA design.

In the following subsections, a detailed discussion of the mentioned CGRAs building blocks and system integration solutions is provided, along with the design trade-offs they introduce.

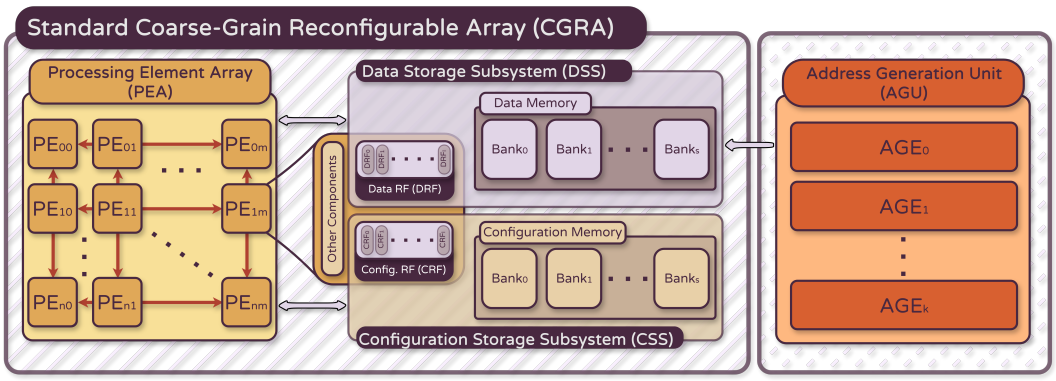


Fig. 2.2 Overview of the main components found in CGRA designs.

2.2.1 Processing Element Array

The PEA is the computational core of a CGRA. Over the years, the literature has proposed many PEA implementations, covering a broad range of hardware complexity. As a result, it is not easy to classify more than 20 years of implementations. For this reason, the following discussion does not aim to be exhaustive, but it focuses on highlighting the main trends that have emerged in PEA implementations through the years.

As the name suggests, the PEA is an array of PEs, which are the basic building blocks and the main sources of computation. Each PE can support multiple operations, and its functionality can be configured at kernel level (i.e., once at the beginning of kernel execution) or cycle by cycle, depending on the implementation. The interconnect between PEs can also be reconfigured, enabling different dataflow patterns. Possible structures of PEA are illustrated in Figure 2.3 The microarchitecture of the PE strongly affects both the set of applications that can be mapped and key hardware metrics such as area, power, and performance. Although many variations exist, a PE is typically organized around four main components, which are described in the following and shown in Figure 2.4.

Functional Unit *(FU)* The FU includes the hardware operators required to execute the operations supported by the CGRA. Over time, FU complexity has progressively increased. Although CGRAs were introduced to move away from the fine-grain reconfigurability of FPGAs, the earliest designs that are now considered CGRAs still adopted a relatively fine configuration granularity. For example, Garp [5] supported

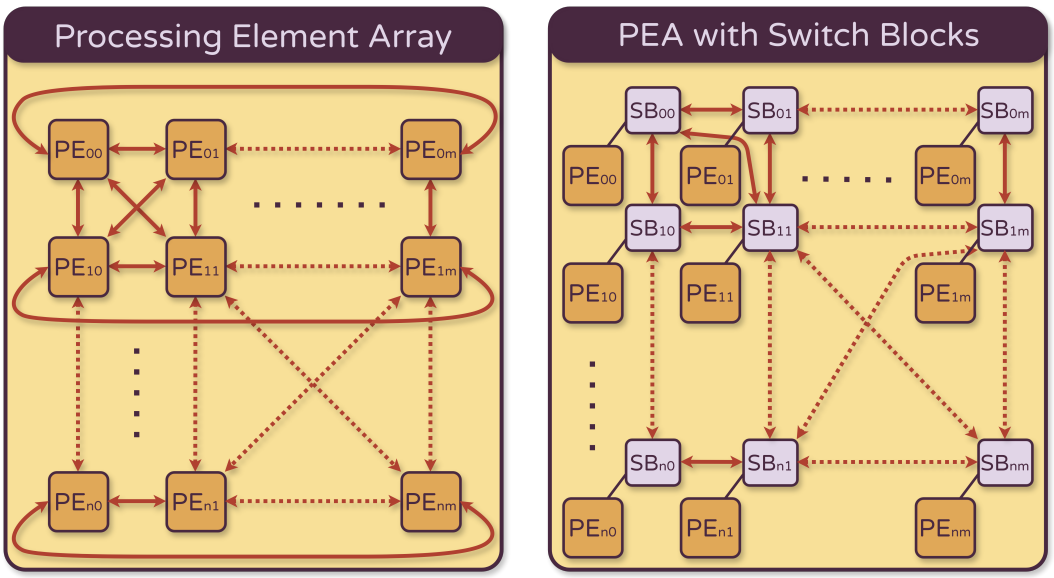


Fig. 2.3 Overview of possible CGRA's Processing Element Arrays (PEA) implementations. On the left, an R×C array of PEs with N2N, diagonal and torus connections. On the right, an R×C array of PEs with Switch Blocks for routing.

reconfiguration at a 2-bit granularity. Its FUs were built around look-up tables, as in FPGAs, and could be connected along the horizontal axis to implement arithmetic operations of different bit-widths. Similarly, CHESS [6] used a 4-bit reconfiguration width. Unlike Garp, it relied on ALU-like structures with fixed functionalities.

Then, the field moved toward progressively coarser and more capable FUs, typically supporting 8-, 16-, and later 32-bit integer operations, and in fewer and mostly later designs also floating-point. These FUs commonly include logic and arithmetic operators such as adders and subtractors, barrel shifters and multipliers. This trend is already visible in the early 2000s across several designs [7–10] that align with the CGRA design approach. In this period, one of the most influential architectures for later CGRAs is Morphosys [11]. Its FUs supported 16-bit arithmetic operations, including adders and subtractors, shifters, and multipliers, as well as bitwise operators. They could also execute additional functions, such as the absolute value of the difference between two operands, and a single-cycle multiply-accumulate (MAC) operation. Most later CGRA FUs, including many recent ones, follow a similar structure: *a coarse-grain FU that mainly executes two-operand instructions, with operators that can sometimes be combined to enable slightly more complex functions and a wider ISA*.

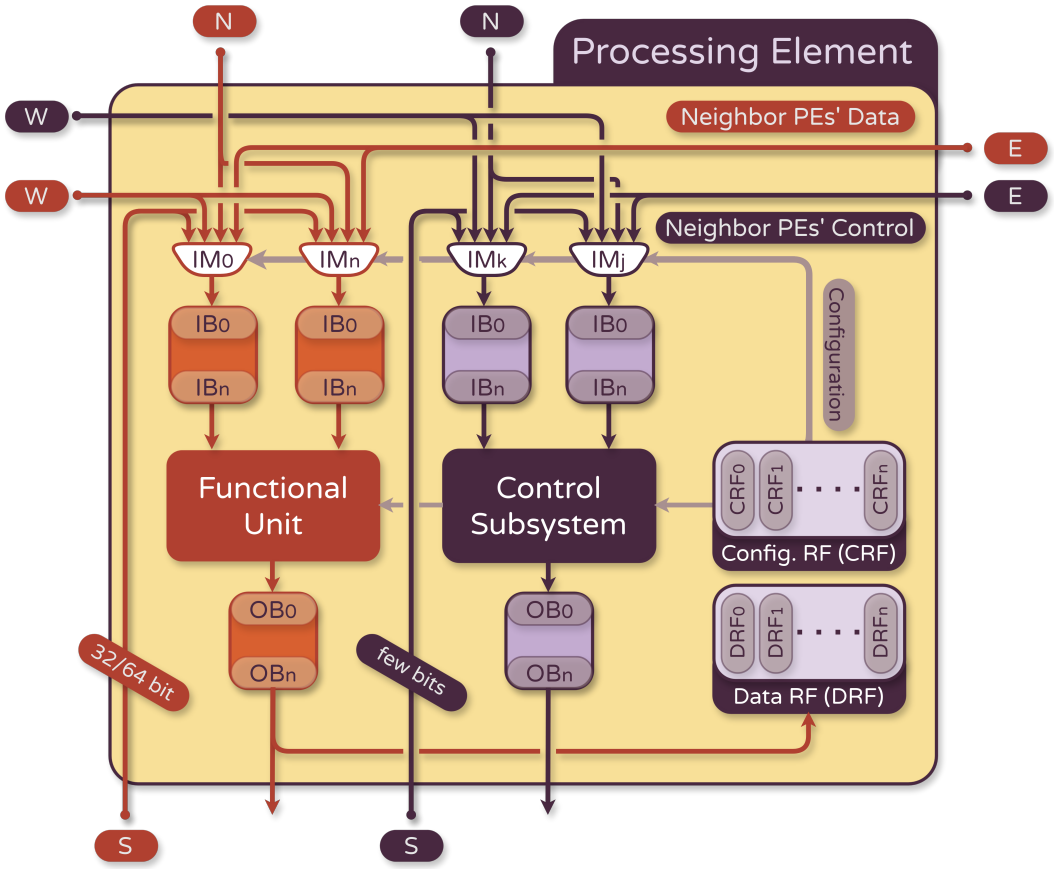


Fig. 2.4 Overview of a CGRA's Processing Element (PE) internal structure, along with its internal components, namely Functional Unit (FU), Control Subsystem (CS), Input Multiplexers and I/O Memory Elements, indicated as IB_i (Input Buffers) and OB_i (Output Buffers).

While this structure represents a common baseline, some designs modify the set of operators or their organization to better match specific computational paradigms. For instance, VecPAC [12] supports vectorization by integrating four 16-bit lanes, which can execute four 16-bit fixed-point operations in parallel or be combined to implement operations up to 64 bits wide. Other CGRAs, such as X-CGRA [13], GP-CGRA [14], and BioCare [15], include 16-bit approximate fixed-point operators in the FU. In BioCare, approximate computation is used to improve energy efficiency for bio-signal processing at the edge. The target application domain can also motivate less common operator choices. For example, PICACHU [16] and NX-CGRA [17] incorporate support for non-linear functions that appear in Transformer models.

Their FUs include floating-point and fixed-point dividers, respectively, which are operators that are usually avoided in many CGRAs due to their cost.

Input Multiplexers *(IMs)* All PEs feature two IMs that select the operands of the FU from the set of inputs available for the PE. These inputs can typically come from neighboring PEs or from the DSS, including both a data memory internal or external to the CGRA, or a register file local to the PE. Naturally, the complexity of the IMs scales with the number of available input sources. In CGRAs that support elastic computation, additional IMs are also required for the valid/ready signals that propagate through the PEA together with the corresponding data. Although two data-related IMs are common in most CGRAs, there is no strict requirement to limit their number. For example, some CGRAs [18] include extra multiplexers for a *bypass operand*, which allows an operand to be forwarded to neighboring PEs without passing through the FU. However, increasing the number of IMs raises PE area and, more importantly, increases routing complexity in the physical implementation. As a result, this choice must be carefully evaluated.

Input/Output Memory Elements These are flip-flop-based storage elements used to hold input and/or output operands of the FU within a PE. Several designs [11, 19, 20, 17] only include a single register to store the FU output before forwarding it to the DSS or neighboring elements. Many other CGRAs [21, 22, 18] also add input buffers. Output registers are more common than input ones, but input storage cannot always be avoided. For instance, in CGRAs that support elastic computation, input buffers are often used for synchronization, either as single registers or as First-In-First-Out (FIFO) structures [18]. In this setting, an explicit output register may also be unnecessary in some designs, as shown by UE-CGRA [18].

Control Subsystem *(CS)* The CS encompasses all the logic devoted to handling the access to CSS discussed in subsection 2.2.2, as well as the handling of control flow operations in general. To be fair, the "standard" approach to CGRAs employed by Morphosys [11] and others [10, 7] and do not include an explicit CS within PEs. The main reason is that Morphosys-like CGRAs are strictly dataflow-centric, mainly targeting applications with few control-flow burden, as discussed in subsection 2.2.7. Hence, control-flow handling is not essential in these CGRAs. An additional moti-

vation is that these architectures are often statically configured, with the CSS often coinciding with a single configuration register within the PE. As a result, handling the access to the CSS is not needed in these cases. However, as in recent years the trend is to support more and more complex code constructs on CGRA (as discussed in subsection 2.2.7), components that can be identified as CS can be commonly found in recent PEs. In this context, several solutions have been proposed to enlarge CGRA code coverage towards complex control flow handling.

One of the most widely adopted techniques to support conditional execution on CGRAs is **Predication**. Predication is a control-flow handling mechanism in which operations are augmented with a boolean guard, called *predicate*, so that the operation commits its effects (i.e., updates its destination register or forwards its result) only if the predicate is asserted. Predication is usually applied both in static CGRAs, such as HyCUBE [23], and elastic ones, like Snafu [21] that employs static or modulo-scheduled execution and, architecturally, it usually requires extending the PE with a predicate register file and predicate-evaluation and gating logic.

Different predication schemes have been proposed. A common approach is **Partial Predication** (PP), where the *Then* and *Else* regions are mapped onto distinct spatial paths and their results are reconverged to a specific PE that selects the valid result based on the predicate. PP is straightforward to map, but it always allocates PEs to computations that are later discarded, thus wasting power and hardware resources. In contrast, **Full Predication** (FP) maps mutually exclusive instructions operating on the same value onto the *same* PE, but at different time slots. Hence, the value has a single "home" location, and only the operation whose predicate is true updates it at runtime. This reduces the need for merge logic and can improve energy efficiency, but it increases latency since the final value becomes available only after the scheduled slots for both paths. Several variations of FP target further power minimization. While FP still requires fetching/decoding operations from both paths, **State-based FP** (SPF) [24] adds an instruction-skip mechanism, typically implemented with small counters inside the PE, to bypass decoding of operations belonging to the non-taken path. SPF mainly reduces wasted decode energy, but does not remove FP's latency behavior. Another scheme is **Dual-Issue Single-Execution** (DISE) [25], which improves latency over FP by pairing one operation from the *Then* and one from the *Else* path on the same PE in the same cycle, with the predicate selecting which one actually executes/commits. Overall, as regards the control subsystem within the PE, predication usually mainly requires additional

registers to store predicates and few related control logic. As a result, it is quite easy to implement with a low area overhead, but it allows handling rather simple *ITE* constructs.

A second family of solutions to control flow handling in CGRAs stays close to a **PC-centric** execution model, where control flow is explicitly managed through a PC and branch instructions. In these implementations, such as IPA [22], R-BLOCKS [19] or NX-CGRA [17], the PE-local control subsystem mainly resembles a lightweight CPU front-end interface rather than predicate-based masking, thus encompassing PC calculation, PC-based access to configuration and branching support. In the specific case of R-BLOCKS, PEs fully dedicated to control-flow handling are used. These PEs, called *Branch Units* (BUs), support control-flow operations and generates/broadcasts the *same* PC to one or more instruction fetch/decode units over the reconfigurable network. This PC-based mechanism is quite flexible, allowing for wider code coverage than predication, but it usually requires increased PE area overhead compared to predication. As a matter of fact, it required additional registers to manage jump conditions and locations, as well as a local controller to generate the PC, access the CSS and manage branch instructions.

Another family of solutions, usually employed by elastic CGRAs like STRELA [26], UE-CGRA [18], but also Riptide [20], considers control-flow constructs as part of the data-flow graph, but differently from predication, dedicated directives/operators can be employed by PEs, such as *merge* and *join*. In this class of designs, conditionals are typically implemented by branch-like operators that forward tokens (ready/valid signals) only to the selected successor, and by *merge/join* operators that reconverge tokens from mutually exclusive paths. Therefore, the control subsystem is effectively embodied by the PE's support for these control-oriented operators plus the associated elastic control logic.

In literature, other more specific solutions that do not fit the mentioned common approaches are also applied to manage control flow. A notable mention is Marionette [27], which decouples control-related operations within a PE through a dedicated **control plane** separated from data-related logic. In Marionette, control flow is represented by instruction addresses, and PEs generate and send instruction addresses to other PEs to coordinate execution in an autonomous, peer-to-peer fashion. Architecturally, this requires extending each PE with additional control-related blocks for generating, scheduling and sending instruction addresses to other PEs,

as well as local storage/state for this control information. Other approaches to CS design could be found in literature for specific PE designs, but they will not be discussed here since the motivations behind their usage are usually too specifically related to the specific CGRA implementation.

Interconnections In order to form the PEA, interconnections among PEs are necessary. The PEA interconnection scheme is decided at design time, thus defining, for each PE, the set of elements to which it can communicate. These PEs are usually referred to as *neighbor PEs*. While the interconnection scheme is fixed, connectivity among neighbor elements can be reconfigured on a kernel or cycle basis, depending on the execution model of the CGRA. Interconnections are a key aspect in CGRAs, as they highly influence both the physical implementation of the design and its flexibility in terms of workloads mapping. In the last twenty years, especially for compact CGRAs in the embedded domain, the most used interconnection scheme is the so-called *Nearest-Neighbor* or *Neighbor-to-Neighbor* (N2N) scheme, which requires each PE to communicate with its nearest neighbors along the vertical and and horizontal axes. Some small variations to this scheme can also be found commonly in literature, mainly involving diagonal connections and the *torus* scheme, as shown in Figure 2.3 which is essentially equal to the nearest-neighbor except for boundary PEs along rows and/or columns: the two PEs at the extremes of a row and/or column are connected to one another, thus creating a *toroidal* interconnection pattern. The nearest-neighbor scheme is the most employed as it allows for simple interconnections with contained area occupation, while also allowing for discrete flexibility in terms of workload mapping.

However, N2N fabrics may represent an issue if "long" data dependencies must be mapped onto the PEA. If dependent operations are assigned to PEs that are far apart in the array, values must traverse multiple intermediate elements, often referred to as *hops*, that act as routers rather than be assigned to computation, thus wasting computational resources on routing. To mitigate this limitation, some CGRAs [23, 20, 28] adopt Network-on-Chip (NoC)-based or multi-hop interconnects, where the network can be configured so that data travels across multiple elements within up to a single cycle, effectively enlarging neighborhood. For instance, HyCUBE introduces a reconfigurable single-cycle multi-hop interconnect to enable same-cycle communication between distant PEs, reducing the need to spend intermediate PEs and extra cycles purely for forwarding. This solution highly increase workloads

mapping flexibility, but incurs mainly area and timing issues. As a result, N2N remains the mostly adopted interconnection scheme, especially for CGRAs in the embedded systems domain.

A different approach is employed in HPC-oriented CGRAs, where area constraints are not so stringent. In this context, routability can be increased by decoupling routing from computation through dedicated *Switch Blocks* (SBs), as illustrated in Figure 2.3, acting as discrete routers, as happens in RIKEN CGRA[29]. Since throughput is a key figure for HPC CGRAs, wasting PEs as hops is not ideal. Hence, RIKEN equips each PE with a SB router tile, thus allowing computations to be mapped on the PE and delegate interconnections and possible hops to the SB. In RIKEN, SBs form a king-style mesh with vertical, horizontal and diagonal connections. This significantly improves routability and PE usage, but comes with a notable hardware cost, as this choice is reported to have substantially higher, up to a 6.3×, area occupation compared to a N2N scheme with PEs acting as routers [30]. Therefore, this SB-based organization has naturally motivated Design Space Exploration (DSE) on the interconnect itself [30, 31], to evaluate how reduced reduced-connectivity SB topologies may influence routability, performance, and resource usage. As a result, it has been assessed that the horizontal-diagonal (no vertical connections) connection topology saves about 30% of the resource usage while maintaining virtually the same routing flexibility as the full connectivity topology.

Furthermore, a notable approach to interconnects is the one of Riptide, which exploits them not only for data transport but also for control-flow handling. It uses a statically configured, multi-hop, bufferless NoC and explicitly implements several control-flow operators inside the NoC switches by manipulating routing selection and ready/valid signals. The key motivation is that control-flow operators can be implemented through crossbars that are already present in routing elements, rather than wasting PEs.

Homogeneous vs. Heterogeneous PEAs Historically, most CGRAs have adopted a homogeneous PEA, meaning that all PEs share the same internal microarchitecture. This choice maximizes flexibility because all PEs can execute any instruction defined by the CGRA ISA. In recent years, however, several DSE-oriented works [32, 31, 33] have explicitly explored heterogeneous PEAs and shown benefits in area, power consumption, and energy efficiency. A heterogeneous PEA includes different types

of PEs, each tailored to a subset of the instruction set. This specialization can improve workload distribution across the array, while also avoiding the additional area and power costs of implementing the full ISA in every PE. Several CGRAs with heterogeneous PEAs have been proposed in the literature, and some representative examples are discussed next.

An example is VecPAC [12], which includes vector-enabled PEs. These PEs can operate on two 64-bit fixed-point operands, or partition the same inputs to execute two 32-bit operations or four 16-bit operations in parallel. Since these PEs are more area-intensive than scalar ones, VecPAC adopts a heterogeneous PEA that mixes vector and scalar PEs. This organization captures the benefits of vector computation without paying the area cost of a fully vector-enabled array. PICACHU [16] follows a different approach, featuring hardware/software co-design techniques. PICACHU targets fixed-point and floating-point polynomial approximations of non-linear functions through a tailored ISA. The design partitions the ISA into subsets and maps them to different PE types, which improves workload distribution across the array. Riptide [20] also exploits heterogeneity to maximize energy efficiency. It includes *memory PEs* for loads and stores, *arithmetic PEs* for basic arithmetic and logic operations such as compare, bitwise operations, add, subtract, and shift, *multiplier PEs* for multiplication-related operations, *control-flow PEs* for control-flow handling, although many control functions are implemented in Riptide's NoC, and *stream PEs* that generate affine iterators, typically consumed by memory PEs to support affine memory accesses. Similarly, R-BLOCKS [19] uses dedicated Load-Store Units (LSUs) and Branch Units (BUs), and also distinguishes between PEs for basic arithmetic and logic and PEs specialized for multiplication-related operations.

A main drawback of heterogeneous PEAs is reduced flexibility compared to homogeneous ones. As a matter of fact, heterogeneity makes mapping more constrained, since placement of operations and routing among PEs must respect which operations each PE type can execute. However, [31] shows that, with appropriate DSE and mapping tools, heterogeneous PEAs can still support a wide range of workloads while providing the expected area, power, and energy benefits. Furthermore, compiler support becomes more important, and several compilation approaches have been proposed to handle heterogeneous arrays.

2.2.2 Configuration Storage Subsystem

Within the proposed classification, the Configuration Storage Subsystem (CSS) includes all memory elements used by the CGRA to store the configuration of PEs and their interconnections. The CSS rarely corresponds to a single standalone block. Instead, it is usually defined by the set of SRAM macros and/or registers that may be placed internally to the PEs and/or outside the PEA, as shown in Figure 2.5. The following discusses the most common CSS implementations and their main trade-offs.

In most CGRAs, the CSS includes at least a **Configuration Register File** (CRF) placed locally in each PE. The CRF size largely varies across designs and is closely tied to the supported execution model. For CGRAs that rely on static kernel mapping, the CRF can be as small as a single entry [26, 18] or contain only a few entries [20, 21]. In the single-entry case, the register stores the configuration that remains active throughout the entire execution of the kernel. On the other hand, with a small multi-entry CRF, each entry typically stores the configuration of a different kernel, and configurations are still applied statically. Compared to a single-entry design, multiple entries allow faster switching between kernels because several configurations can be preloaded. This choice is usually lightweight in terms of hardware cost.

CRF size generally increases in CGRAs that adopt a PC-centric execution model [22, 17] or modulo scheduling [23, 12, 16, 1, 34]. In both cases, the CRF size directly affects which workloads can be mapped onto the array. This is especially relevant for modulo-scheduled CGRAs because the CRF bounds the maximum Initiation Interval (II) that can be supported. If the CRF is small, each PE can store only a limited number of instructions, which restricts the set of kernels that can be placed and routed onto the array. As the required II grows, more configuration entries are needed per PE, and workloads that exceed the available CRF capacity cannot be mapped. For modulo-scheduled designs, CRFs are implemented either as flip-flop-based register files [23, 12] or using SRAM macros [1]. Nonetheless, supporting large IIs can make the CRF a major area contributor. For example, in [34] each PE includes a 32-entry, 32-bit CRF implemented with registers, and CRFs account for 39% of the total CGRA area. The overhead can be even larger when SRAM macros are used locally within the PE, as in CASCADE [1], where CRFs occupy more than half of the overall CGRA area when considering both PEs and Stream Units.

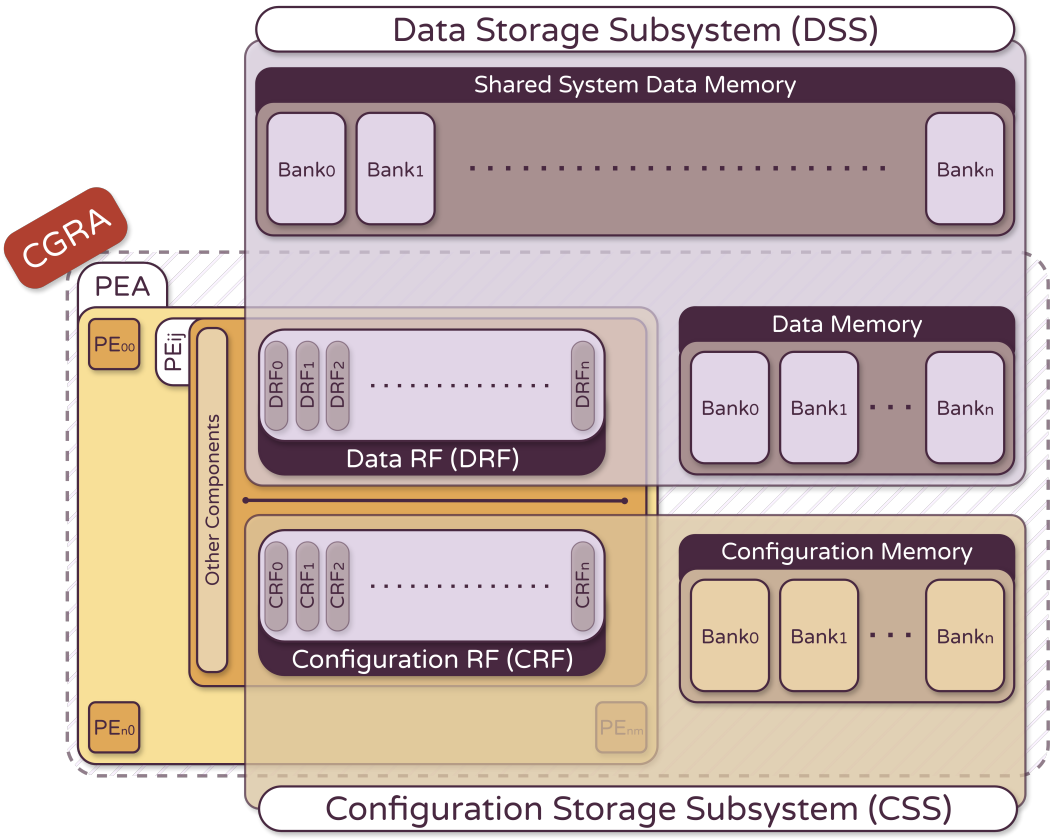


Fig. 2.5 Configuration Storage Subsystem (CSS) and Data Storage Subsystem (DSS) and their internal components.

Only a few CGRAs do not include a CRF inside the PEs. One example is R-BLOCKS [19], where the PC-centric model fetches instructions from on-chip **Configuration Memories** (also referred to as instruction memories) addressed by program counters (PCs) generated by Branch Units (BUs). More generally, instruction memories implemented as SRAM macros are also common in other CGRAs [34, 17, 22], but often with a different role. In designs that already use a per-PE CRF, SRAM macros placed outside the PEA are frequently used to store multiple *contexts*, meaning configurations for multiple kernels. This approach is more common than storing many contexts inside each PE, as done in Riptide [20]. Despite relying on SRAM, the area overhead is often modest, around 7% in both [34] and [17], since the context memory is typically only a few KiB. In addition, storing multiple contexts enables more complex computations by allowing several related kernels to execute back to back. For instance, NX-CGRA [17] can store multiple

kernels in a Transformer pipeline and run them in sequence, possibly multiple times, which reduces reconfiguration overhead.

2.2.3 Data Storage Subsystem

In the proposed classification, the Data Storage Subsystem (DSS) includes all memory elements used to store PEA inputs, intermediate values, and outputs, as illustrated in Figure 2.5. This definition excludes the PE input and output buffers, which are discussed in subsection 2.2.1 to keep the presentation consistent with the PE microarchitecture.

At the PE level, the DSS may include a **Data Register File** (DRF). Its main purpose is to store intermediate results for local reuse [22, 11] or to hold constants [17] that are repeatedly consumed by subsequent operations in the same PE. A DRF can be seen as an extension of the PE output buffer. However, its capacity is usually kept rather contained, since replicating it across all PEs can have a non-negligible impact on the overall area. While DRFs are used in several designs, many CGRAs [26, 17, 20, 21] prefer to avoid them mainly for preventing the related area overhead.

The main DSS component is the **Data Memory**, which is primarily used to store inputs and outputs. Intermediate values are less commonly stored in DM because keeping them local and reusing them is often more efficient. The DM is always implemented with one or more SRAM banks, but it is not necessarily integrated within the CGRA. Many implementations rely on the usage of a shared memory managed at the system level. Even though this memory is external to the CGRA, it is included in the classification to capture the full range of design choices observed in the CGRA landscape. For many CGRAs [26, 18, 17, 20, 35], the main motivation for using system memory is area reduction, since integrating a DM inside the CGRA can make the memory dominate the overall area.

At the same time, many CGRAs adopt an internal DM, often referred to as a Scratchpad Memory (SpM). Although this choice increases area, it can reduce access latency, enable a customized interface between the PEA and memory, and provide full control over memory organization and *data layout*. This control is typically not available when relying on shared system memory. With a suitable data layout, bank conflicts can be reduced significantly, which improves both performance and energy

efficiency. For this reason, several works [36–39] propose mapping techniques for CGRAs equipped with multi-bank SpMs that avoid memory access conflicts. However, while many CGRAs [23, 28, 40] integrate SRAM banks as DM, only a few implementations [1] explicitly adopt these conflict-avoidance techniques.

2.2.4 DAE Paradigm & Address Generation Unit

As discussed in subsection 2.2.1, heterogeneous PEAs have become more popular in recent years because they can improve area and energy efficiency compared to homogeneous arrays. One of the increasingly adopted approaches to heterogeneous PEAs involves the adoption of specialized PEs for load/store operations [26, 20, 19, 1, 17, 28]. These are often referred to as **Address Generation Units** (AGUs), although the terminology may vary across different implementation. When AGUs are employed, they exclusively handle address generation and memory accesses, thus effectively implementing the so-called **Decoupled Access Execute** (DAE) paradigm.

Architectures based on the DAE computation principle separate memory access and data-related computation into dedicated units, as illustrated in Figure 2.6. One set of units (AGUs) focuses on generating addresses and performing memory operations, while other units (PEs) operate only on input data to produce results. The DAE principle was first proposed in [41], but it did not become common in general-purpose CPUs because it requires careful synchronization between the two sides, while also highly increasing compiler design complexity. In contrast, DAE has been widely adopted in DSPs, neural-network accelerators [42], and reconfigurable accelerators [43, 44]. In these domains, memory access patterns are often regular, and access and compute stages can be separated more easily, along with a reduced need for complex synchronization. DAE has also been explored in High-Level Synthesis [45] to automatically generate efficient accelerators, and it has appeared in other ASIC and reconfigurable designs even when not explicitly described as DAE.

One of the first works introducing the DAE paradigm to the CGRA world was Chameleon [7]. Its AGUs generate typical DSP memory access patterns, such as incremental and decremental addressing. In more recent years, CGRAs have been proposed exploiting the DAE model both when the DSS is external and when it is internal to the CGRA. In the first category, Riptide [20] uses *Stream PEs* to compute affine access patterns, which are then used by *Memory PEs* to issue load and store

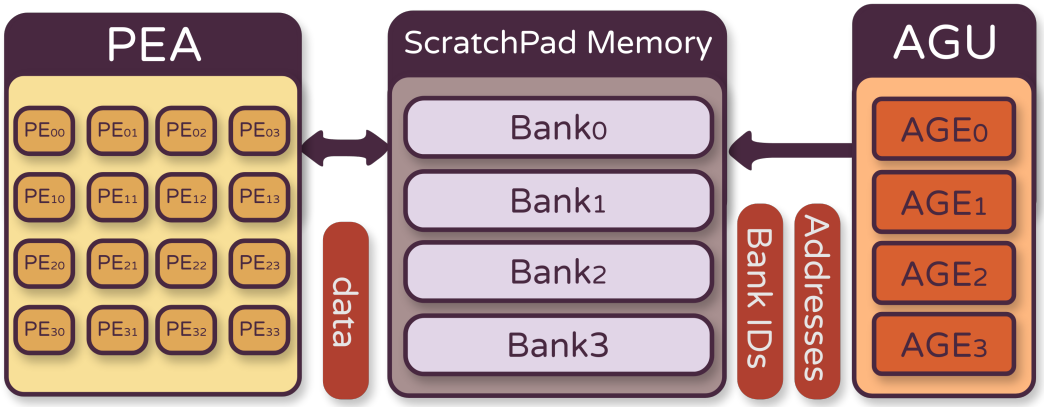


Fig. 2.6 Overview of the Decoupled Access-Execute (DAE) paradigm implementation on a CGRA. The *Execute* unit is implemented by the PEA, the *Access* unit is implemented by the AGU and a Scratchpad Memory is interposed between the two to be accessed by the AGU and provide and retrieve data from the PEA.

operations to system memory. STRELA [26] and NX-CGRA [17] adopt *Input/Output Memory Nodes* (IMNs, OMNs) and *Memory Operation Blocks* (MOBs), respectively, to perform memory accesses, although they do not detail the supported addressing schemes. In the second category, R-BLOCKS [19] uses *LSUs* to access both the internal data memory and the global one. Softbrain [28] relies on *Stream Engines* (SEs) to compute two-dimensional affine accesses of the form $A[C*i+j]$, where the induction variables i and j increment from zero to an upper bound. Softbrain also supports indirect addressing, where the base address to be accessed is read from another array.

Among CGRAs that employ the DAE model, CASCADE [1] stands out because it provides a detailed motivation for this design choice. CASCADE observes that a substantial fraction of PEs can be consumed by **Address Generation Instructions** (AGIs) within a kernel. The authors evaluate the fraction of AGIs across several workloads on a CGRA with single-bank and multi-bank SpM, and report values ranging from 20% up to 80% in the multi-bank case, as reported in Figure 2.8. Multi-bank memories require extra computations to select the bank index and compute offsets, which increases the number of AGIs. If these computations are mapped onto standard PEs, they consume resources that could otherwise execute dataflow operations, and they can also increase the II, thus resulting in performance reduction. CASCADE also compares an ideal DAE CGRA against a conventional one on the

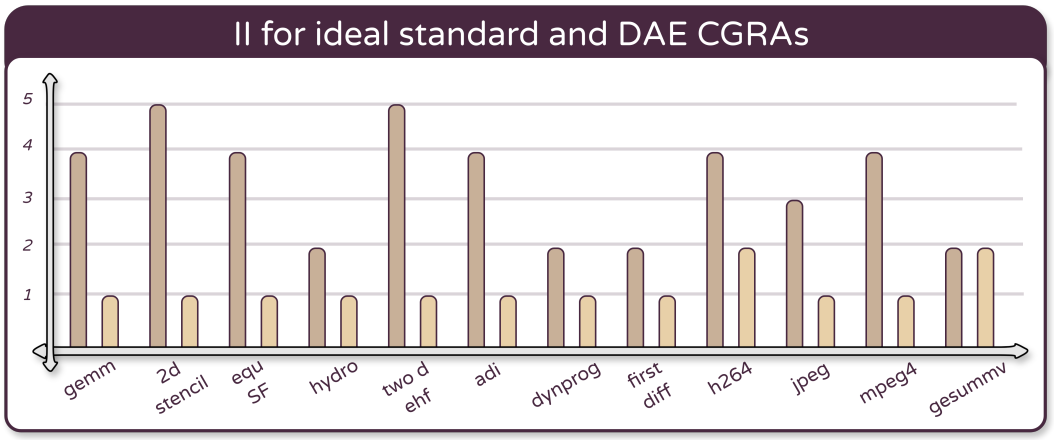


Fig. 2.7 Comparison of II across several workloads on ideal standard CGRAs (light brown bars) and ideal DAE CGRA (light yellow bars). Data is taken from [1].

same kernel set, and reports that the II reduction can be as high as 80%, as shown in Figure 2.7.

In addition, if AGUs are designed together with a well-defined data layout for the SpM, conflict-free memory accesses can be achieved, enabling high-bandwidth streaming between the DSS and the PEA. For example, STRELA takes advantage of the capability of X-HEEP memory subsystem to store data in an interleaved manner across multiple banks to reduce access conflicts. However, among all CGRA designs implementing the DAE model, only CASCADE adopts a data layout model aimed at completely avoiding the occurrence of conflicts in accesses to SpM. CASCADE employs a data layout scheme based on the **Generalize Memory Partitioning** (GMP) scheme, proposed in [4]. In this scheme, the placement of an array in a multi-bank memory is defined by two parameters, and an example is provided in Figure 2.9. The block size B_s specifies how many consecutive elements of the array form a block, while N_b indicates how many banks the blocks are spread across. Once B_s and N_b are fixed, the bank index Bn and the memory address A associated with an element at logical index x are computed as:

$$\begin{aligned} Bn(x) &= \frac{x}{B_s} \% N_b \\ A(x) &= \frac{x}{N_b \cdot B_s} \cdot B_s + x \% B_s \end{aligned} \tag{2.1}$$

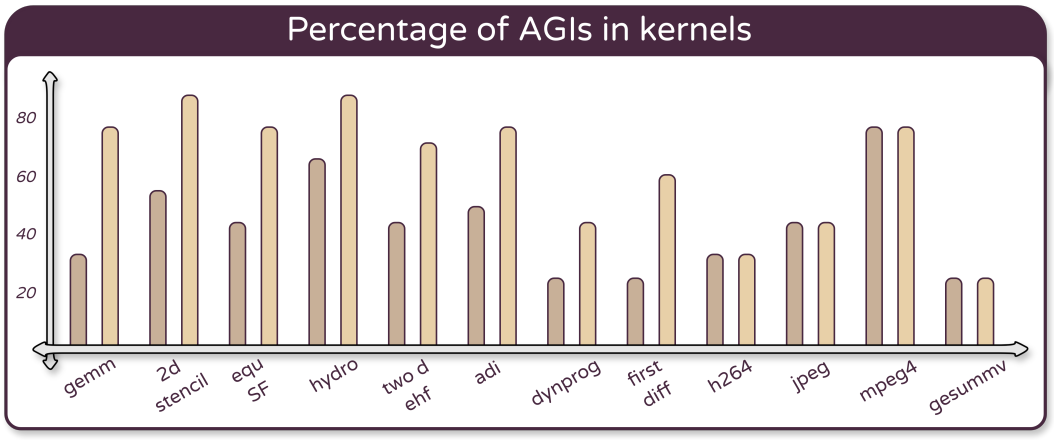


Fig. 2.8 Address Generation Instructions (AGIs) percentage across various workloads when either a single-bank data memory is used (light brown bars) and multi-bank data memory is employed (light yellow bars). Data is taken from [1].

When both parameters are powers of two, these functions can be implemented efficiently in hardware onto the AGU using shift operations. As a result, the AGU can be tailored to this specific scheme and, if scheduling of load/store and data-related operations is carefully performed, memory access conflicts can effectively be avoided. Therefore, in conclusion, the analysis carried out by CASCADE highly motivates the use of the DAE paradigm in CGRAs, delegating AGIs to dedicated units, leaving PEs focused on data-related computation.

Overall, adopting the DAE paradigm in CGRAs is promising, especially when the DSS is an internal multi-bank SpM, since it enables joint tuning of data layout and AGUs to achieve conflict-free accesses. Moreover, the DAE model can also be applied when the CGRA relies on external memories, but it introduces limitations. In particular, control over data layout can be highly reduced, thus making conflict avoidance harder. In addition, supporting multiple master or slave ports can increase system interconnect complexity, which becomes more problematic in advanced technology nodes.

2.2.5 System Integration

Especially for embedded systems, a key factor that drives CGRA design choices is how the CGRA is integrated into the target platform. In literature, two main integra-

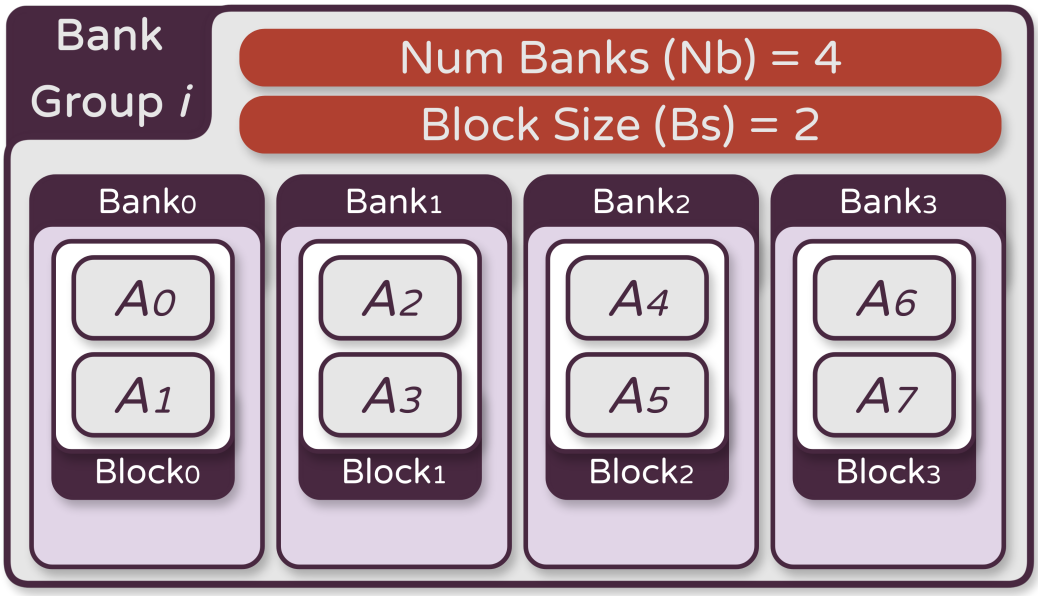


Fig. 2.9 Data layout inside a scratchpad memory based on the Generalize Memory Partitioning (GMP), with $N_b = 4$ and $B_s = 2$.

tion styles can be identified, and they are respectively referred to as loosely-coupled and *tightly-coupled* integration.

Loosely-coupled integration is the most common approach. In this case, considering a minimal system including a CPU and a memory subsystem communicating through a bus, the CGRA is also attached to the bus and equipped with bus-compliant master and or slave interfaces. Several variants exist within this category. CGRAs [1] that include an internal SpM often expose a single slave port, which allows the CPU or a system DMA to move data between main memory and the SpM. In contrast, CGRAs [26, 18] that stream data directly from system memory typically require multiple master ports to sustain the bandwidth needed during execution. While the second approach is more lightweight in terms of area occupation, it increases the complexity of the system bus, an effect that is more evident mainly in implementations using scaled technologies. As a matter of fact, system buses essentially rely on crossbars which are routing-hungry in physical implementation, and these routing issues can become more severe in scaled technology nodes. Moreover, while the usage of an internal SpM allows for fine-tuning of data layout to reduce access conflicts, this cannot be performed when relying on shared memory handled at the system level.

Tightly-coupled integration, on the other hand, has been explored less often as regards CGRA. In this approach, the CGRA is connected more directly to the CPU, typically by inserting it into the CPU pipeline near the execute stage [46, 47]. Operands are provided from the CPU register file through a dedicated interface, and results are forwarded to the write-back and memory stages. This organization enables the system to detect sequences of instructions in the CPU's code that match the CGRA ISA and can benefit from being offloaded.

In conclusion, the vast majority of CGRAs are integrated as loosely-coupled accelerators. The main reason behind this choice is that they can be designed to exploit higher memory bandwidth, as well as more efficient memory access patterns, whether they use internal SpM or stream from external memory. Therefore, this organization allows accelerating entire workloads. On the other hand, tightly-coupled CGRAs typically accelerate only selected code regions, since the CPU remains responsible for their control. However, related works have shown that tightly-coupled designs can still deliver meaningful speedups and energy savings over CPU-only execution.

2.2.6 Computation Model

CGRAs can execute workloads using different computation models. In this classification, two orthogonal aspects are considered: the *scheduling mechanism* and the *execution mechanism*. The **scheduling mechanism** describes how configurations for CGRA components, such as PEs and interconnects, are assigned and applied over time. In other words, it refers to who decides which element adopts a given configuration, and when that configuration becomes active. Scheduling can be **static** or **dynamic**.

In **static scheduling**, similarly to FPGA bitstreams, configuration decisions are made entirely at compile time, or directly by the user if a compiler is not available. In this case, the compiler must solve both a scheduling and a mapping problem. In other words, it must assign each operation to a time step and also to a physical location on the array, while respecting the PEA size and its interconnect topology.

A widely used static-scheduling technique is **modulo scheduling**. It is typically applied to loop-based kernels and can be seen as a form of software pipelining that exploits instruction-level parallelism across different loop iterations. Since a

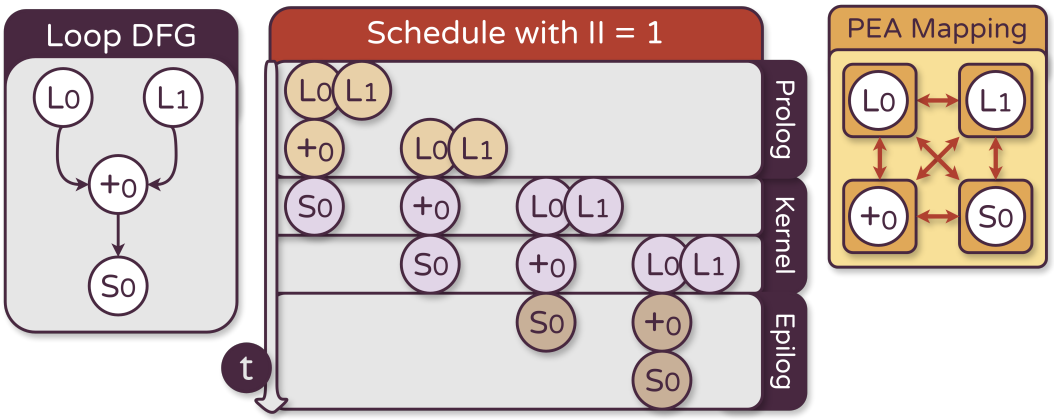


Fig. 2.10 The loop DFG on the left is modulo scheduled with $II = 1$ on the 2×2 PEA on the right. In the considered case $RecMII = 1$ as no loop-carried dependencies are present, and $ResMII = \frac{4}{4} = 1$, hence the kernel can be mapped with $II = max(ResMII, RecMII) = 1$. Therefore, the *kernel* is composed of a set of instructions that span through a single cycle.

single iteration often exposes limited parallelism, overlapping multiple iterations increases concurrency and improves PE utilization. Modulo scheduling starts from the kernel *Data-Flow Graph* (DFG) and a description of the target PEA, including its size and interconnect. It then computes the *Minimum Initiation Interval* (MII), defined as the minimum number of cycles between the start of two consecutive loop iterations. The MII is the maximum between the *resource-constrained* bound and the *recurrence-constrained* bound, respectively denoted as *ResMII* and *RecMII*. ResMII captures limitations due to available resources, such as the number of PEs relative to the number of DFG operations. RecMII captures loop-carried dependencies and it is determined by the workload, while being independent of the CGRA microarchitecture. After computing these bounds, the compiler solves a combined scheduling and mapping problem to select a feasible II, place operations onto PEs, and route values across the available interconnect. The resulting schedule has three main phases. The *prologue* and *epilogue* are respectively executed once at the beginning and end of execution, while the central part, defined as *kernel*, is repeatedly carried out a number of times equal to the amount of iterations of the loop. The kernel has a schedule length equal to II and contains operations that may belong to different loop iterations. This mirrors a software pipeline: the prologue fills the pipeline, the kernel sustains steady-state execution, and the epilogue drains it. During the kernel, each PE cycles through II configurations, typically switching every cycle. For this reason, II is a key performance parameter, especially for loops with many iterations. The ideal

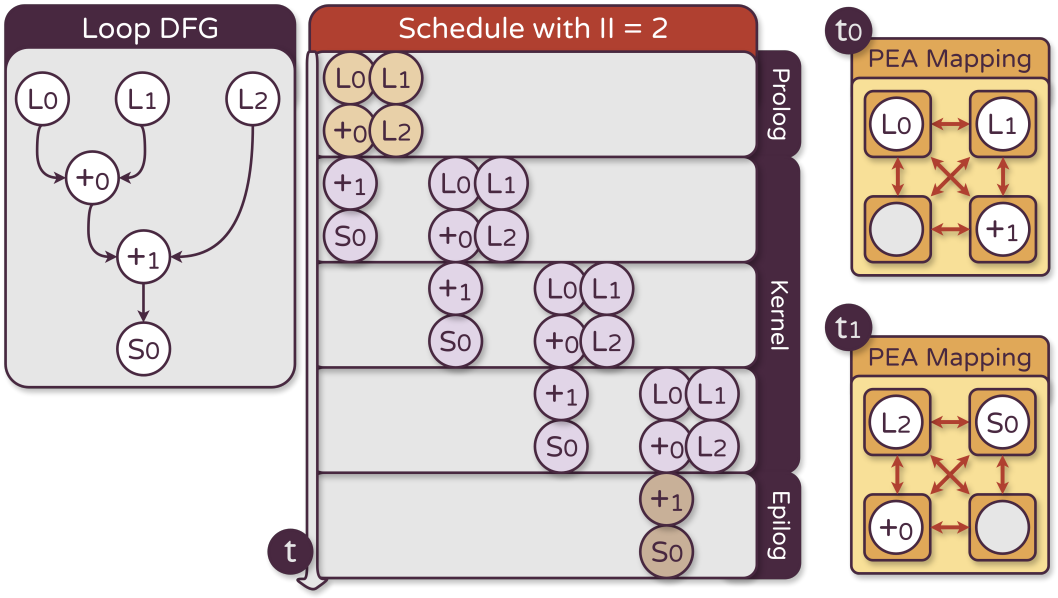


Fig. 2.11 The loop DFG on the left is modulo scheduled with $II = 2$ on the 2×2 PEA on the right. In the considered case $RecMII = 1$ as no loop-carried dependencies are present, and $ResMII = \lceil \frac{6}{4} \rceil = 2$, hence the kernel can be mapped with $II = max(ResMII, RecMII) = 2$. Therefore, the *kernel* is composed of a set of instructions that span through two consecutive cycles.

case is II equal to one, but this is often prevented by resource limits or loop-carried dependencies. Nonetheless, some CGRAs [11, 26, 20] employ a fully static model that only supports II equal to one, which corresponds to a single configuration that remains active for the entire kernel execution. Examples of modulo scheduling are illustrated in Figure 2.10 and Figure 2.11

While modulo scheduling is common, other CGRAs [17, 22, 19] implement static scheduling using a PC-centric mechanism. In these designs, each PE still changes its operation and connections on a cycle basis, but configurations are fetched from the CSS using a PC rather than being expressed as an explicitly unrolled schedule.

Overall, in static scheduling, the operations executed by PEs and the required interconnections are determined at compile time, with limited control being performed at runtime in hardware. Therefore, this approach facilitates hardware implementation of the CGRA. Nonetheless, a possible drawback is that the complexity is shifted from hardware towards the compilation stage.

On the other hand, **dynamic scheduling** assigns configurations to CGRA elements at runtime based on the current system state and on the availability of resources.

This is conceptually similar to superscalar processors, which schedule instructions dynamically using mechanisms such as prediction and reservation stations. CGRAs that adopt this approach [28, 40, 48, 49] are typically more complex than statically scheduled designs. The additional hardware control and runtime decision logic reduce the burden on the compiler, since fewer decisions must be fixed at compile time. Some surveys [50] suggest that dynamically scheduled CGRAs are becoming more common, drawing an analogy with the CPU evolution from VLIW to out-of-order execution. However, CGRAs based on static scheduling remain widely adopted because they can achieve strong area and energy efficiency, and because static compilation is still practical, even though it often requires careful design and continued improvement.

As regards the **execution mechanism**, it describes how operations are carried out on the CGRA once a configuration is active. If operations follow an order that is fully determined at compile time, the CGRA uses **sequential execution**. In this model, the compiler must ensure that operands are available exactly when needed, since PEs execute according to a predefined schedule. This increases compiler complexity, but it highly simplifies hardware because it avoids implementing extensive synchronization among PEs and between the PEA and the DSS.

In **dataflow execution**, instead, a PE performs an operation only when its input operands are available and marked as valid. This approach moves part of the coordination task to hardware, which must propagate ready and valid conditions and enforce correct synchronization. However, it reduces the need for the compiler to precisely time operations within a configuration. CGRAs that adopt this model are often described as *dataflow* or *elastic* architectures [26, 18].

2.2.7 Code Coverage & Target Applications

In the previous sections, we discussed architectural choices and computation models adopted in CGRAs. Many of these decisions are closely tied to the types of workloads and algorithmic constructs that a CGRA can support, which will be referred to as **code coverage**. This section outlines the main trends in CGRA code coverage in recent years and summarizes the advantages and drawbacks of the corresponding design approaches. Furthermore, this analysis is a crucial premise to the motivations behind the CGRA proposals and explorations carried out in this thesis.

As highlighted by several surveys [51, 52, 50], CGRAs have traditionally been used in computation- and data-intensive domains. As a matter of fact, their architecture offers substantial parallel compute resources, but it is not naturally suited to complex control-flow handling. Therefore, early CGRA designs [9–11] focused on loop-based kernels, including single loops and loop nests, that could be spatially unrolled onto the PEA with no control flow or only very simple control operations. Although this structure is conceptually simple, it appears in many workloads across different application areas. As a result, CGRAs have often been developed as domain-specific accelerators, targeting families of kernels within a given field. Representative domains include *Digital Signal Processing* (DSP) [7, 6], *Hardware Security* [53, 54], *Graph Processing* [55], and *Deep Learning* [16, 17, 56]. Most of these designs combine static scheduling with either sequential or dataflow execution. This combination keeps PE microarchitecture relatively simple while still supporting the targeted kernel classes.

Nonetheless, over time, a clear trend has been to expand CGRA code coverage toward more complex algorithms [21, 20, 19], which, in this thesis, will be referred to as **Dynamic Control Programs** (DCoPs). The motivation behind this choice is mainly increasing the fraction of an application that can run on the accelerator, so as to reduce host–accelerator context switching and enhance energy efficiency. DCoPs include workloads whose control flow and memory access patterns cannot be fully determined at compile time. Typical cases are loops with data-dependent bounds, memory accesses with data-dependent indices, and conditional execution through *If-Then-Else* constructs driven by runtime values. To support these patterns, several works enhance the control subsystem within the PEs and add mechanisms for dynamic control and memory behavior, and some of these techniques are discussed in subsection 2.2.1. These extensions usually increase area and power compared to earlier "classic" CGRAs, but multiple strategies have been proposed to keep the overhead limited while improving applicability and maintaining strong energy efficiency across a broader set of workloads.

2.3 Considerations & Motivations

As discussed in the previous section, many recent CGRA proposals aim to expand code coverage toward DCoPs by introducing architectural mechanisms for data-

Table 2.1 Kernel classification across CGRA architectures. For each architecture, the number of kernels (**N**) and the corresponding percentage over the total supported workloads (**%**) are reported for each kernel class. Kernel classes are SCoP (Static Control Programs), IA (codes containing Indirect Addressing), CC (codes containing Complex Control) and SC (codes containing Simple Control).

CGRA	**SCoP**		**IA**		**CC**		**SC**		**Total**
	N	**%**	**N**	**%**	**N**	**%**	**N**	**%**	
Riptide [20]	5	45.5	3	27.3	2	18.2	1	9.1	11
SoftBrain [28]	2	28.6	3	42.9	1	14.3	1	14.3	7
R-BLOCKS [19]	11	78.6	0	0.0	0	0.0	3	21.4	14
CASCADE [1]	11	100.0	0	0.0	0	0.0	0	0.0	11
UE-CGRA [18]	3	60.0	0	0.0	0	0.0	2	40.0	5
VecPac [12]	8	72.7	2	18.2	0	0.0	1	9.1	11
Plasticine [40]	9	69.2	2	15.4	2	15.4	0	0.0	13
DA-CGRA [57]	9	100.0	0	0.0	0	0.0	0	0.0	9
HyCUBE [23]	5	62.5	0	0.0	0	0.0	3	37.5	8
IPA [22]	9	69.2	0	0.0	1	7.7	3	23.1	13
Overall	**72**	**70.6**	**10**	**9.8**	**6**	**5.9**	**14**	**13.7**	**102**

dependent control flow and irregular memory accesses. To understand how these capabilities translate into practice, an analysis has been carried out regarding the type of accelerated workloads among 10 recent influential CGRAs. Although the examined works consistently claim support for kernels that are more complex than those targeted by "classic" CGRAs, the reported workloads set tells a more intricate scenario. Authors of these works describe their targets as kernels that combine one or more of the following features: regular and irregular loop nests at different nesting levels, data-dependent control flow, and affine and irregular memory accesses. However, as summarized in Table 2.1, the analysis of the kernels accelerated by these CGRAs shows that only a small fraction actually exercises these advanced features. In particular, **just 13.7% of the workloads include only minimal control flow**, typically a single trivial *If-Then-Else* inside the innermost loop, while **only 9.8% use indirect addressing** and **only 5.9% exhibit non-trivial control flow**.

On the other hand, the vast majority of accelerated workloads (71%) fall into the class that, in compiler theory, is commonly referred to as **Static Control Parts** or **Static Control Programs** (SCoPs). SCoPs encompass workloads whose control

flow is fully determined at compile time and whose memory accesses are affine. In other words, control-flow statements (*If-Then-Else*, etc.) can only rely on loop induction variables (e.g., loop iterators), and array indices are expressed as affine functions of loop induction variables, meaning linear expressions. This class not only largely overlaps with the family of structured kernels traditionally targeted by "classic" CGRAs, but it is also broader and more general.

This observation leads to a key point: despite the push toward DCoP support, modern CGRAs remain largely evaluated, and often primarily used, on SCoP-like workloads. This is not accidental. As noted in prior work [28], the workloads that are typically most attractive for hardware acceleration tend to share the following features: (1) long phases with high computational intensity, (2) small instruction footprints with simple control flow, and (3) predictable memory access and reuse patterns. These properties closely match, and can be seen as a subset of, the characteristics of SCoPs. Moreover, such workloads also cover a large portion of nowadays important compute kernels in edge applications, including many building blocks used in modern AI and signal-processing pipelines.

As a conclusion, the current trend in CGRA design towards increased code coverage can be absolutely valuable, but CGRAs remain largely tied to SCoPs as CGRA's full computation advantage is primarily unlocked on this set of kernels. Therefore, hardware mechanism that are added to enlarge code coverage can limit this advantage, incurring lower area and energy efficiency as well as reduced performance regarding SCoPs acceleration. This mirrors an observation made for other programmable accelerators, as highlighted in [28]. Single-Instruction Multiple-Data (SIMD) and GPUs can execute well kernels whose characteristics are listed above, but maintaining broad generality often introduces efficiency issues. Similarly, pushing CGRAs toward extensive generality risks paying overheads that only marginally improve code coverage, while degrading the efficiency of the dominant workload class that CGRAs still primarily accelerate in practice.

Based on the discussed insights, the work carried out in this thesis explores a complementary point in the design space compared to the recent trend in CGRA designs. Rather than maximizing coverage of arbitrary program regions, it focuses on maximizing performance and area and energy efficiency for the large and practically important subset of workloads identified as SCoPs. In other words, instead of pursuing maximum generality within the CGRA fabric, this thesis aims to fully

exploit CGRAs' fundamental strengths by specializing for a more restricted but widely applicable workload class, i.e. SCoP kernels, encompassing most of the typically accelerated kernels in nowadays edge applications.

To achieve high performance together with strong energy and area efficiency, the two CGRAs proposed in this thesis are designed around three key implementation principles.

(1) **Exploitation of DAE Paradigm**

In SCoP workloads, memory access patterns are always **affine**, meaning that array indices (A) can be expressed as linear combinations of loop induction variables (IV), as shown in Equation 2.2.

$$A(IV_i) = \sum_{i}^{N_{IV}} k_i \cdot IV_i \tag{2.2}$$

Affine address expressions are well suited for efficient hardware implementation using dedicated AGUs. This observation motivates the adoption of the DAE paradigm, in which all address computation and memory access functionality is offloaded from the PEA to specialized units. Therefore, under DAE, the PEA is reserved exclusively for arithmetic operations, while data movement is handled more efficiently by purpose-built AGUs.

This separation improves energy efficiency and reduces pressure on the PEs, since AGIs are no longer mapped onto the PEA. As a result, fewer operations have to compete for PE resources, lowering the ResMII and, in turn, the overall II. As a consequence, this results in higher throughput for SCoP workloads. Furthermore, DAE can be combined with a carefully designed memory data layout to minimize or totally eliminate bank access conflicts, further improving performance.

(2) **Control-Free PEA**

SCoPs are characterized by control flow that is fully determined at compile time. Consequently, no control divergence can occur during kernel execution. The only *If-Then-Else* constructs that may appear have conditions expressed as functions of loop IVs and, therefore, are statically predictable. This property eliminates the need for PEs to support data-dependent control flow, dynamic branching, or jump instructions, together with the associated context switching.

By removing control-flow support, the PE microarchitecture is significantly simplified. In addition to offloading address generation, this choice also removes control-related instructions from the PEA, further reducing the number of operations that must be scheduled on the PEs. This directly contributes to lower ResMII and II, thus enhancing throughput, while also improving area efficiency. In addition, static control decisions can be handled by the AGUs, which already have direct access to loop induction variables and, if carefully designed, to loop structure information.

③ **Fully-Static Scheduling**
The removal of address-generation and control-flow responsibilities from the PEA enables highly compact and regular schedules. Building on this insight, the proposed CGRAs are designed to support SCoP workloads with an II equal to one. Achieving $II = 1$ implies the use of a fully static schedule, in which a single configuration of the CGRA is sufficient for the entire execution of a kernel.

This design choice maximizes performance while further improving energy and area efficiency. Since CGRA elements do not need to switch among multiple configurations on a cycle-by-cycle basis, as in modulo-scheduled designs with II greater than one, configuration storage and control overhead are reduced. The result is a simpler architecture that is optimized for high-throughput execution of structured workloads.

2.4 The X-HEEP Platform

This section introduces the X-HEEP framework [3] and describes its main architectural components. In this thesis, X-HEEP has been adopted as the integration platform for the proposed CGRAs and has served as the reference environment for their evaluation and comparison. *X-HEEP is an open-source 32-bit RISC-V microcontroller platform* that can be exploited for the design of heterogeneous System-on-Chips (SoC). Its key strengths lie in its high degree of **configurability** and **extensibility**, which enable designers to tailor the system to their specific performance, area, and power requirements.

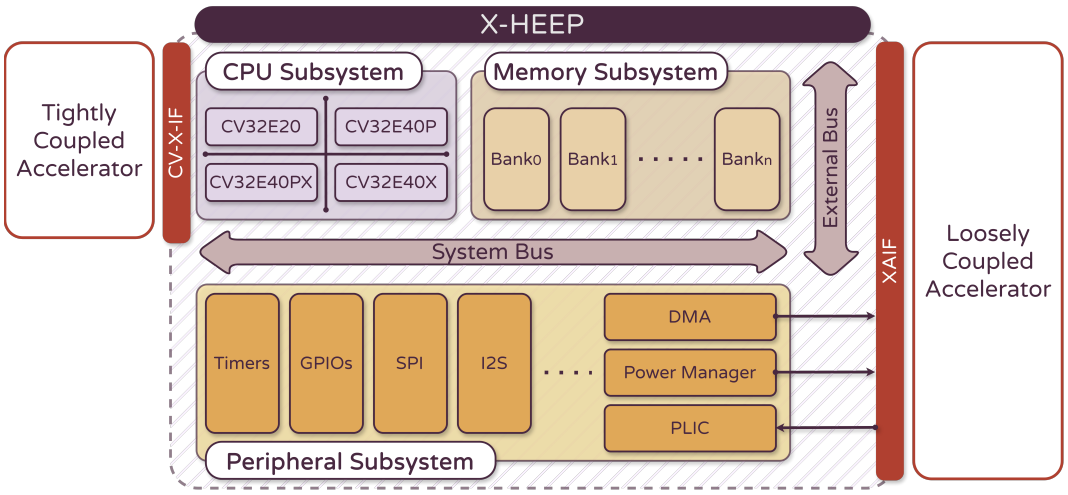


Fig. 2.12 X-HEEP.

2.4.1 Platform Configurability

From the configurability perspective, X-HEEP allows extensive customization of its internal architecture. Designers can select and parameterize several Intellectual Properties (IPs), either developed at the Embedded Systems Laboratory (ESL) of EPFL (where X-HEEP has been designed) or provided by third parties. In the following, the main subsystems and components of X-HEEP are presented, along with their customization capabilities

CPU Subsystem The framework supports four CPU cores developed by the OpenHW Foundation: CV32E20, CV32E40P, CV32E40PX, and CV32E40X. The CV32E20 core [58] targets control-oriented applications and prioritizes low power and reduced complexity. On the other hand, the CV32E40P and CV32E40PX [59] are optimized for compute-intensive workloads, featuring support for floating-point operations and the XCVPULP ISA extension, which enhances throughput through the integration of hardware loops, SIMD instructions, and other compute-oriented mechanisms. The CV32E40PX further provides the CORE-V-XIF interface to integrate co-processors implementing custom RISC-V ISA extensions, as discussed in subsection 2.4.2. The CV32E40X also supports this interface but removes floating-point and XCVPULP extensions, enabling a different trade-off between area, power, and computational capabilities. This rather wide range of CPUs allows designers

to select the most suitable core according to their specific performance/power/area needs.

Memory Subsystem The memory subsystem is also highly configurable. Designers can select both the number and size of memory banks, as well as the supported data layout strategy. X-HEEP supports either *contiguous* or *interleaved* data placement. In a contiguous layout, each data structure is stored sequentially within a group of banks. In contrast, in an interleaved placement, adjacent data elements are distributed across adjacent banks. Therefore, the latter approach is particularly beneficial in systems integrating accelerators or multiple bus masters, as it reduces memory access conflicts and increases effective throughput when concurrent accesses are performed.

Peripheral Subsystem X-HEEP gathers the different internal IPs within a *peripheral subsystem*. The designer can easily select which IPs to integrate into the X-HEEP-based SoC, although some of them are essential, such as the boot ROM and the SoC controller. These component and other essential services are grouped into an *always-on* domain, also encompassing a fast interrupt controller and the Direct Memory Access (DMA) unit. In addition, the peripheral subsystem can include a platform-level interrupt controller (PLIC), timers, a power manager, and commonly used communication interfaces, including GPIO, I2C, I2S, and SPI. To optimize for power, these other peripherals can be clock- or power-gated when not in use. Nonetheless, it is important to highlight that also the DMA can be clock-gated although part of the always-on domain. Also, each peripheral can be selectively removed to save area. The power manager can implement clock gating, power gating, and memory retention techniques.

A particularly important component for the CGRAs presented in this thesis is the DMA subsystem [60]. Edge and AI-oriented data-intensive applications require low-latency, energy-efficient data transfers with flexible addressing capabilities. To address these needs, the X-HEEP DMA features a highly configurable architecture. Designers can specify the number of DMA channels, the number of master ports on the system bus, and the capabilities of each channel. Channels may either have dedicated master ports or share them, a choice that directly impacts area, power consumption, and system bus complexity, as increasing the number of masters

increases crossbar complexity. Each DMA channel can be configured with advanced *addressing features*, including support for both 1D and 2D transfers with optional *zero-padding*. These capabilities are particularly useful when transferring data between system memory and accelerators that require specific internal data layouts. Overall, the DMA subsystem provides extensive flexibility to balance throughput, area, and architectural complexity.

System Bus The CPU, memory and peripheral subsystems communicate through a system bus based on the Open Bus Interface (OBI). The bus topology can also be configured to implement either a *one-to-M* or an *N-to-M* topology. In the *one-to-M* configuration, only one master can access the bus at a time, minimizing hardware resources. In contrast, the *N-to-M* configuration implements a fully connected crossbar, enabling multiple masters to access multiple slaves concurrently and maximizing parallelism and throughput. This choice allows designers to trade off resource utilization against performance.

2.4.2 Platform Extensibility

Beyond configurability, X-HEEP also provides strong extensibility mechanisms to enable the realization of heterogeneous SoC designs. Two main integration interfaces are available for incorporating external accelerators, and they are discussed in the following:

(1) The **eX-tendible Accelerator InterFace (XAIF)** defines a set of ports that enable the integration of **loosely-coupled** accelerators, which are accelerators that operate independently of the CPU execution. When such accelerators have to be included, X-HEEP provides an additional external bus, which is in turn connected to the main system bus. Loosely-coupled accelerators can attach to this external bus through master ports, allowing them to access X-HEEP's internal memory-mapped resources, and through memory-mapped slave ports, which are accessible by the CPU and other masters to configure the accelerator or transfer data to it. Furthermore, X-HEEP enables a direct connection between loosely-coupled accelerators and DMA channels via a FIFO-based interface. This approach allows computational capabilities to be integrated directly within DMA data transfers, leveraging the DMA's memory

access features without increasing the complexity of the system bus. The XAIF also provides interrupt lines to connect accelerators to either the fast interrupt controller or the PLIC, as well as control signals that allow the power manager to apply clock- or power-gating to the integrated accelerators.

(2) The **Core-V eXtension Interface (CV-X-IF)**, developed by the OpenHW Foundation, supports the integration of **tightly-coupled** accelerators. These accelerators are directly embedded into the CPU pipeline and interact closely with the core's execution pipeline. They are typically used to implement custom ISA extensions, enabling low-latency communication with the CPU and efficient support for application-specific instructions.

Part II

Mage and Rogue CGRAs

Chapter 3

Introduction & Summary

In the following sections, the two CGRA architectures developed in this work are presented, namely **Mage** and **Rogue**. Both CGRA architectures are built based on the **DAE paradigm** introduced in subsection 2.2.4, although they explore two complementary points in the design space.

The first CGRA, **Mage**, is presented in chapter 4 and it explicitly identifies SCoP workloads as primary targets for DAE CGRA-based acceleration. In this setting, Mage exploits an internal multi-bank scratchpad memory, a dedicated address-generation unit, and a fully static execution model to sustain conflict-free accesses and high-throughput execution. In chapter 4, section 4.2 introduces the internal architectural organization of Mage along with its main configurable components. After that, section 4.3 discusses the adopted kernel scheduling and mapping methodology and, finally, section 4.4 evaluates Mage conducting a design-space exploration quest. Therefore, several Mage implementations are generated exploiting its versatile template-based design, and they are evaluated across multiple SCoP workloads, highlighting the various area-performance-energy trade-offs. In addition, subsection 4.4.5 evaluates Mage on an end-to-end Temporal Convolutional Network application. Finally, subsection 4.4.6 reports comparisons of Mage against state-of-the-art standard and DAE CGRAs, highlighting its superiority in throughput, area- and energy-efficiency.

The second CGRA architecture, **Rogue**, is presented in chapter 5 and it has been developed as a streaming DAE CGRA tailored for the acceleration of integer approximations of activation functions. Compared to Mage, Rogue leverages a lighter

DAE architecture, offloading the implementation of memory and address-generation subsystems to the system it is integrated into. Therefore, Rogue encompasses a FIFO-based interface for I/O communication to be exploited for integrating it with external LSUs or, more importantly, system DMAs [60, 61]. In addition, Rogue combines a dedicated instruction set capturing recurring computational patterns of integer activations with a lightweight streaming architecture that can also support conventional GeMM kernels. As a result, Rogue has been integrated in X-HEEP-based SoCs as a DMA-coupled streaming accelerator for efficient on-the-fly execution of integer activation kernels. In chapter 5, the DMA-coupled integration model developed in X-HEEP is described in subsection 5.3.1, along with the first streaming peripheral integrated into an X-HEEP-based taped-out SoC exploiting this interface. Then, section 5.4 and section 5.5 present Rogue's ISA and architecture, respectively, while section 5.6 discusses Rogue's execution model and the constraints it imposes on the workload mapping process. Finally, section 5.7 conducts a design space exploration analyzing various Rogue implementations across several activation and GeMM kernels in terms of throughput, area and energy-efficiency, while subsection 5.7.8 discusses the comparisons against specialized accelerators for activations and other standard and DAE CGRAs. Also, section 5.8 presents the integration of Rogue as a DMA-coupled streaming accelerator in an X-HEEP-based SoC taped out on *TSMC* 65nm technology.

Overall, Mage and Rogue address two complementary acceleration scenarios. On one hand, **Mage** is particularly effective at delivering high throughput on SCoP and GeMM kernels, thanks to the use of a tailored data layout in its internal SpM that avoids memory-access conflicts, although this incurs a non-negligible area cost due to the integration of SRAM macros. On the other hand, **Rogue** adopts a lighter architecture, which enables lower system-level area overhead while still providing significant throughput (although lower than Mage's), area efficiency, and energy efficiency on both activation kernels and GeMM workloads. Therefore, the two CGRAs explore different trade-offs within the DAE CGRA design space, each exposing various advantages and limitations that are discussed in detail in the following sections. Both Mage and Rogue are provided as open-source at `https://github.com/vlsi-lab/Mage-and-Rogue`. As will be discussed in the following sections, they can be both generated starting from template files based on `hjson` configuration files. Depending on the used configuration file, an implementation of Mage or Rogue can be generated.

Chapter 4

Mage: A DAE CGRA Tailored for Static Control Applications

4.1 Motivations and Related Works

As discussed in section 2.3, a large portion of recent work on CGRAs has aimed at extending code coverage towards increasingly general program constructs, mainly by including support for DCoPs. While this direction improves applicability, it often introduces mechanisms that reduce throughput and area/energy efficiency on the structured SCoP kernels for which CGRAs typically provide the largest benefit. At the same time, in practice, the vast majority of accelerated workloads still falls within the SCoP class, which also dominates application domains that are currently of great interest, such as ML. This motivates the investigation of CGRA designs explicitly tailored for SCoPs.

With respect to SCoP acceleration, a key inefficiency of conventional CGRAs lies in the large amount of resources spent on memory-address generation. In particular, when the datapath is fed by a multi-bank SpM, additional address-related computations, involving bank selection and intra-bank addressing, can significantly increase the number of AGIs mapped onto the PEA, thus reducing PE utilization for arithmetic operations, as highlighted in [1]. DAE CGRAs address this issue by separating access and execute datapaths. Hence, memory access and address generation are handled by a dedicated unit, often referred to as AGU, while the PEA only focuses on dataflow arithmetic operations. For SCoPs, this separation

is particularly beneficial, since both addressing and static-control decisions can be expressed as affine functions of loop IVs, thus being amenable to efficient hardware implementation.

In the literature, some DAE CGRA proposals have explored this direction. **CASCADE** [1] couples affine stream engines in its AGU with an internal multi-bank SpM to minimize access conflicts, but its AGU targets relatively simple patterns, mainly incremental addressing within up to two perfectly nested loops. Moreover, while not identifying SCoPs as its primary target, CASCADE is only evaluated on SCoP workloads. **STRELA** [26] is another DAE CGRA that integrates input/output memory nodes in its AGU to perform accesses to the main system memory. However, its AGU also relies on relatively simple access patterns and, without an internal SpM, it cannot fully co-design address generation with the SpM data layout. Therefore, although attaining lower area occupation, STRELA's throughput still suffers from memory access conflicts. Moreover, although STRELA supports some DCoP workloads, its best performance is reported on data-driven workloads with fixed control flow, which largely fall within the SCoP class. **NX-CGRA** [17] is another recent DAE CGRA that, similarly to STRELA, integrates Memory Operation Blocks (MOBs) to access main system memory, while also providing support for packed-SIMD computation on reduced-precision data (`int8` and `int16`) and non-linear activation functions. As explicitly stated in the related paper, NX-CGRA places a highly specific focus on GeMM workloads and activation functions, which still belong to the SCoP workload set. Nonetheless, NX-CGRA's throughput remains bounded by access conflicts due to non-optimal data placement in main system memory.

Therefore, most existing CGRA works adopting the DAE model either still pursue broad code coverage, in line with the current trend, or do not clearly identify SCoPs as the primary acceleration target. Nonetheless, in practice, they are largely evaluated on SCoP workloads and, in some cases, explicitly report their best results on this class of kernels. In contrast, **Mage** is proposed as a DAE CGRA that explicitly targets SCoP kernels and is designed to fully exploit the DAE paradigm in this setting. Mage differs from prior DAE designs along three main axes:

1. **More expressive affine access generation**: Mage integrates a *multi-address generation unit* (AGU) whose address computation relies directly on loop IVs provided by a *hardware-loop unit*. Therefore, multiple concurrent loads/stores

can be sustained, while supporting **deeper loop nests** and more **complex array access patterns**.

2. **Static-control offload to the AGU**: since control decisions in SCoPs are described by affine functions of induction variables, the AGU can efficiently handle them by *gating the issuing of loads/stores accordingly*. This reduces memory-related power consumption, while also allowing the PEA to focus only on data-related computation.

3. **Conflict-free AGU→SpM interaction**: Mage supports GMP-compliant data layouts within its internal multi-bank SpM, and constrains GMP parameters to powers of 2 in order to efficiently implement bank/address generation in hardware within the AGU. This allows the AGU to perform conflict-free accesses to the SpM, thus targeting high-throughput execution without the overheads typically required to support dynamic behavior.

Based on these insights, **Mage** is proposed as a **template-based DAE CGRA tailored for SCoP workloads**. Mage combines a GMP-compliant multi-bank SpM with an AGU that computes affine access patterns and can also implement static-control decisions driven by hardware-loop induction variables, so that the PEA is mainly dedicated to data processing. Mage adopts a *fully-static mapping model* that targets high performance and area/energy efficiency by avoiding memory access conflicts and maximizing data reuse within the PEA. As a result, Mage achieves the highest throughput among all considered DAE and standard CGRAs, as discussed in subsection 4.4.6, obtaining almost $2\times$ the throughput of the GeMM-specific NX-CGRA (5237.6 MOPS vs. 3040 MOPS), which is the best-performing design among the considered ones, on matrix multiplication using an equally sized PEA implementation. Moreover, Mage significantly outperforms other CGRAs in terms of area efficiency, reaching up to **13.8 GOPS/mm**^{2} on `int8` matrix multiplication. At the same time, despite its higher power budget, Mage achieves energy efficiency results comparable to those of other DAE and standard CGRAs designed for low-power execution.

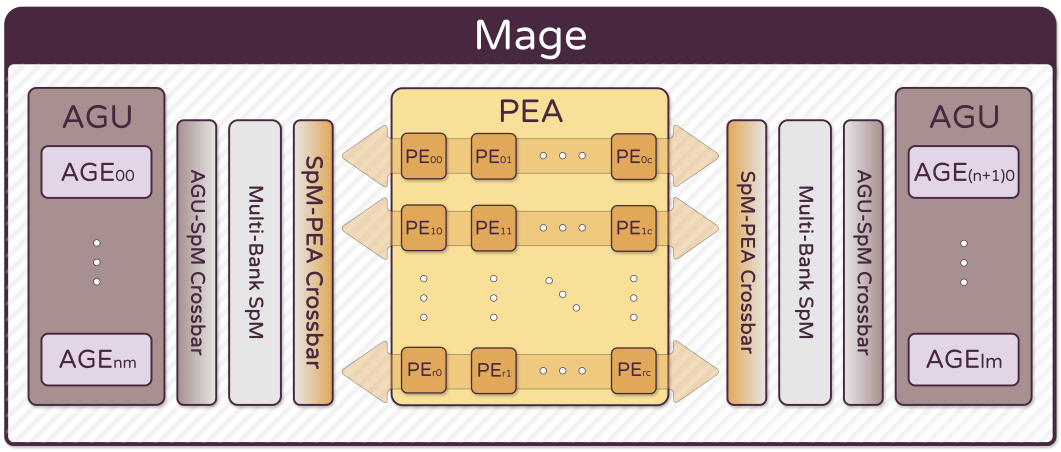


Fig. 4.1 Overview of Mage template-based architecture encompassing PEA, AGU, SpM and crossbars.

Summary

The remainder of this chapter is organized as follows. Section 4.2 provides a detailed description of Mage's internal components, mainly focusing on the AGU, PEA, SpM, and crossbars. Next, section 4.3 presents the methodology adopted to map SCoP workloads onto Mage. Then, section 4.4 reports a design-space exploration based on Mage's template-based design, presenting throughput, area, energy, as well as area and energy efficiency results across the considered Mage variants. In the same section, Mage is also evaluated on an end-to-end application, namely a Temporal Convolutional Network model, and compared against state-of-the-art CGRAs. Finally, section 4.5 concludes the chapter.

4.2 Mage architecture

As a DAE CGRA, Mage separates the access and execute datapaths, interposing a multi-bank SpM between them. The access path is implemented by a multi-address generation unit (AGU) driven by a hardware-loop unit that generates loop IVs. Using the IVs, the AGU issues multiple concurrent load and store accesses to the SpM, enabling high-bandwidth data delivery to the execute side. The execute path is implemented by the PEA, which comprises PEs that strictly focus on arithmetic operations and are designed to sustain high-throughput pipelined data computation,

without implementing any control-flow handling strategy. Mage's SpM is interposed between AGU and PEA as shown in Figure 4.1, splitting AGU and SpM on the two sides of the PEA to ensure balanced bandwidth across PEA's rows. In addition, crossbars are present between AGU and SpM, an between SpM and PEA.

Mage's architecture is **highly configurable**, including PEA, AGU and SpM sizing, crossbar complexity between AGU and SpM, and between SpM and PEA, and heterogeneous PE placement. This flexibility is implemented through a **template RTL description** based on `SystemVerilog` and the `Python mako` template library. Mage is described using `.sv.tpl` files, where `SystemVerilog` code is combined with embedded `Python`. An external `Python` script evaluates the embedded code and emits synthesizable `.sv` files. The script takes as input the template parameters that drive generation. This choice may result in slightly more complex template description files (the `.sv.tpl` files), but the generated `.sv` files are easier to read than using native `SystemVerilog` generative constructs, especially for regular architectures as CGRAs.

In this section, Mage architecture is presented, with a detailed description of its main internal components, together with the template parameters used for RTL generation and the configuration registers used to program Mage.

4.2.1 Multi-Bank ScratchPad Memory Subsystem

Mage includes an internal multi-bank SpM placed between the access and execute paths. Using multiple banks increases the bandwidth available to the execute path compared to a single-bank SpM. The template-based design of Mage allows configuring both the number of banks (`n_banks`) and the bank size in words (`n_words`). In the current implementation, all banks must share the same size and they are modeled as SRAM macros with a single read port and a single write port.

Arrays stored in the SpM subsystem are laid out according to the **GMP model** discussed in subsection 2.2.4, which is briefly recalled here. GMP defines the layout of an array within the SpM using two parameters: ① the **block size** B_s, which specifies how many consecutive array elements form a block, and ② the **number of banks** N_b over which blocks are distributed. Given the linear index x of an element in the array, the target bank id B_{id} and the bank address B_{addr} are computed as:

$$B_{id}(x) = \left\lfloor \frac{x}{B_s} \right\rfloor \%N_b, \quad B_{addr}(x) = \left\lfloor \frac{x}{N_b \cdot B_s} \right\rfloor \cdot B_s + x\%B_s$$

Mage constrains both B_s and N_b to powers of two, so that bank selection and address computation can be implemented efficiently using bit-shifting and mask operations. Therefore, *this constraint highly simplifies the AGU design*, since affine access patterns can be transformed into GMP-compliant bank/address pairs with low overhead.

Finally, *GMP exposes a direct handle to avoid bank conflicts completely*. As a matter of fact, by selecting B_s and N_b appropriately for each array accessed in the workload, and by tuning the schedule of the application DFG accordingly, accesses issued by the AGU can be arranged to target distinct banks whenever parallel memory operations are required.

4.2.2 Address Generation Unit

Mage's access path is implemented by the AGU, whose internal structure is shown in Figure 4.2. The AGU is responsible for generating the addresses required by load/store streams, translating them into GMP-compliant bank/address pairs for the multi-bank SpM, and selectively issuing memory operations based on static-control decisions.

Unlike other DAE CGRAs that exploit dedicated standalone functions for address generation, a key aspect of Mage's AGU is its *integration with a hardware-loop unit*. This unit holds the state of the loop nest, and it allows the AGU to base **affine access patterns computation directly on loop IVs**, thus enabling highly flexible address generation. In addition, direct access to loop IVs is also exploited to implement **static-control decisions** that gate load and store operations. The AGU is composed of several internal blocks, discussed in the following, which handle hardware loop management, affine expression evaluation, GMP translation, and memory operations control.

- **Hardware Loop Unit** *(HLU)* is the core of the AGU, since it *generates the loop IVs* associated with the configured loop nest. During execution, it advances the loop state and produces, at every cycle, a new set of IVs corresponding to the current iteration. Then, these IVs are consumed by the rest of the access path

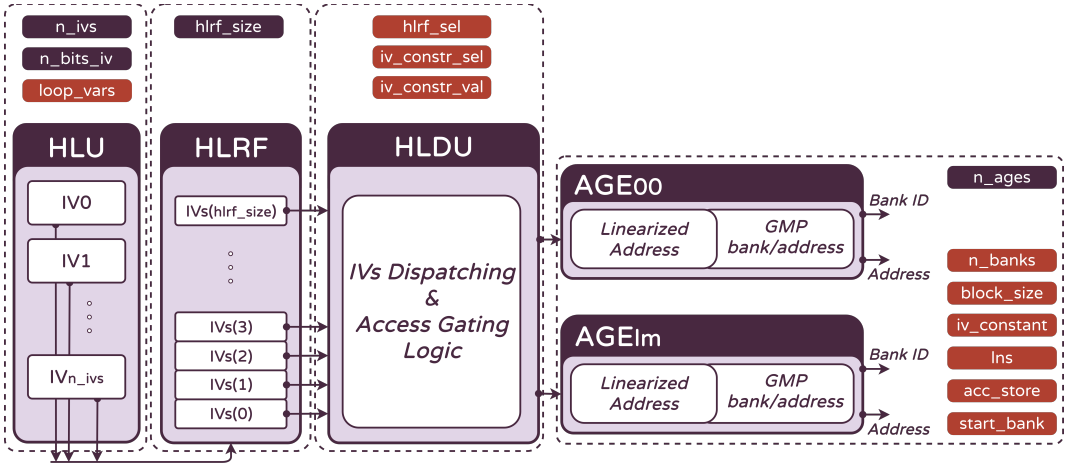


Fig. 4.2 Mage's AGU and its internal components, namely HLU, HLRF, HLDU and AGEs. Template parameters and configuration registers related to the various module are also reported, accordingly to Figure 4.4 and Figure 4.3.

and act as the common input for two key tasks. On one hand, they drive the evaluation of affine expressions used for address generation. On the other hand, they enable IV-driven static-control handling, since conditions that depend on IV values can be evaluated to gate load/store operations without involving the execute path. In its current implementation, Mage supports fixed loop initial and final values, with fixed increment. However, it could be easily extended to support decrementing loops or increments/decrements based on loop IVs.

The HLU is parameterized at design time. Template parameters select the maximum number of IVs it can generate (`n_ivs`) and the bit-width of each IV (`n_bits_iv`). As a consequence, these parameters also influence the complexity of the remaining AGU blocks. As a matter of fact, increasing either the number of IVs or their width increases the cost of the arithmetic and selection logic used by the address-generation engines and by the control logic that operates on IV values. In addition, on the configuration side, bounds and increments for each loop within the nest of the considered workload are set using configuration registers (`loop_vars`).

- **Hardware Loop Register File** *(HLRF)* is a *register file used to buffer the IVs generated by the HLU*. Instead of keeping only the current loop state, it stores the IV sets of the N most recent loop iterations, hence it retains the last N loop

states in terms of their IVs. The depth of this buffer is selected at design time through the template parameter `hlrf_size`.

Keeping multiple IV sets is useful because each address-generation operation needs to start its execution on the related time slot assigned by the kernel scheduling phase (refer to section 4.3). By reading the appropriate entry from the HLRF, an address generation engine can start producing addresses for the related accesses at the required time without stalling the loop progression in the HLU. After all address-generation operations are initiated, the HLRF continuously provides address generation engines with properly-timed loop IVs, so as to ensure correct conflict-free access to SpM as indicated by the kernel schedule. This discussion is complemented by the kernel mapping description addressed in section 4.3.

As a result, `hlrf_size` also impacts kernel mappability. As a matter of fact, it bounds the maximum time slot that can be assigned to memory operations. A larger HLRF provides more freedom for the mapping stage to align accesses with the compute schedule and to resolve conflicts through memory operations skewing, whereas a smaller HLRF restricts these choices.

- **Hardware Loop Dispatcher Unit** *(HLDU)* is placed between the HLRF and the address generation engines (AGEs) discussed in the next point. Its first role is to *provide each configured AGE with the IV set it is intended to use*. Since the HLRF stores multiple loop states, the HLDU selects, for every AGE, which HLRF entry to read and forwards the corresponding IV set to that AGE. This selection is programmed through configuration registers (`hlrf_sel`) and requires, for each AGE, a selector that is $log_2(\texttt{hlrf\_size})$ bits wide.

  In addition to IV selection, the HLDU implements IV-driven static control to gate memory operations. For each AGE, configuration registers specify which IV is used for the control condition (a $log_2(\texttt{n\_ivs})$-bit selector stored in `iv_constr_sel`) and the comparison value (an `n_bits_ivs`-bit constant stored in `iv_constr_val`). Based on this programmed condition, the HLDU decides whether the load/store operation associated with the AGE must be issued. In the typical use case, memory operations are enabled only for a specific value of the selected IV, while all inner IVs are zero, so that the operation is issued once per outer-loop iteration.

This mechanism is particularly useful for store operations associated with accumulations. In that case, the store AGE must be enabled only when the accumulation is completed, so the HLDU gates the store operation and allows it to be issued only at the required loop point.

- **Address Generation Engines** *(AGEs)* are independently configurable units that *compute affine access patterns from the IV set received from the HLDU*. For a given DFG, each AGE is associated with one memory operation, i.e., a load or a store targeting a specific array mapped in the multi-bank SpM. Starting from the selected IVs, the AGE first computes the linearized array address and then translates it into a GMP-compliant bank identifier and intra-bank address. The total number of AGEs can be defined through a template parameter (`n_ages`).

  In the current implementation, an AGE supports an affine address function over up to `n_ivs` induction variables:

$$A(IV_0, IV_1, \ldots, IV_{n\_ivs-1}) = \sum_{i=0}^{n\_ivs-1} S_i \cdot IV_i + C \tag{4.1}$$

  where S_i is the stride associated with IV_i. Each stride S_i is programmed through a `n_bits_iv`-bit field in the AGE configuration registers. Internally, the AGE multiplies each IV_i by its corresponding stride using `n_bits_iv`-bit multipliers, and then sums the partial products using a reduction tree. To limit the critical path, the current design inserts pipeline registers between the multiplication stage and the reduction stage. Since the reduction tree depth increases with `n_ivs`, additional pipeline stages may be required when scaling to larger values of `n_ivs`. At the end, a `n_bits_iv`-bit constant is also added, which is configured using `iv_constant`.

  Once the linear address $A(IV_0, IV_1, \ldots, IV_{n\_ivs-1})$ is computed, the AGE derives the bank index and the address within the bank using the GMP rules recalled in subsection 4.2.1. This translation requires the array-specific GMP parameters B_s and N_b, which are also provided through the AGE configuration registers (`n_banks` and `block_size`). Hence, the final output of the AGE is a pair (B_{id}, B_{addr}) that directly targets the correct SpM bank and location for the corresponding load/store. Each AGE also outputs, with each (B_{id}, B_{addr}) pair,

a load/store bit to identify the type of operation and a validity bit, asserted if the operation must be effectively executed.

After configuration, and assuming that the mapping of the DFG is performed so that parallel accesses target distinct banks, as described in section 4.3, the set of AGEs can produce bank/address pairs every cycle without stalls due to memory-access conflicts.

Figure 4.2 shows the internal organization of Mage's AGU and its AGEs, together with the required template parameters and configuration registers. These parameters and registers are further detailed in Figure 4.3 and Figure 4.4.

AGU Template Parameters

HLU

n_ivs	Total number of loop IVs generated by HLU, i.e., maximum supported loop nest depth.
n_bits_iv	Number of bits of each loop IV.

HLRF

hlrf_size	Depth of the HLRF, i.e., number of buffered IV sets retained from the most recent loop iterations.

AGEs

n_ages	Total number of instantiated AGEs, i.e., the maximum number of concurrent load/store streams the AGU can sustain.

Fig. 4.3 Template parameters to define the structure of the AGU.

AGU Configuration Registers

HLU

loop_vars	Bounds and increments for each loop in the nest.

HLDU

hlrf_sel	For each AGE, selects which HLRF entry provides the IV set to the AGE.
iv_constr_sel	For each AGE, selects the IV, within the selected HLRF entry, to be checked to gate the associated memory operation.
iv_constr_val	Comparison value for the selected "constrained" IV.

AGE

n_banks	GMP N_b for the array of the AGE (power of 2).
block_size	GMP B_s for the array of the AGE (power of 2).
start_bank	Base bank offset for the target array, used to place the array starting block onto a selected bank subset.
lns	Selects whether the AGE issues a load or a store.
acc_store	Set if the AGE performs a store associated with an accumulation, enabling accumulation control toward the PEs on the same row.
iv_constant	Constant term added to the linearized address.

Fig. 4.4 Configuration registers of Mage used by the AGU.

4.2.3 Processing Element Array

Mage's execute path is implemented by the PEA, which constitutes the main compute fabric of the architecture. This section describes the PEA with a top-down

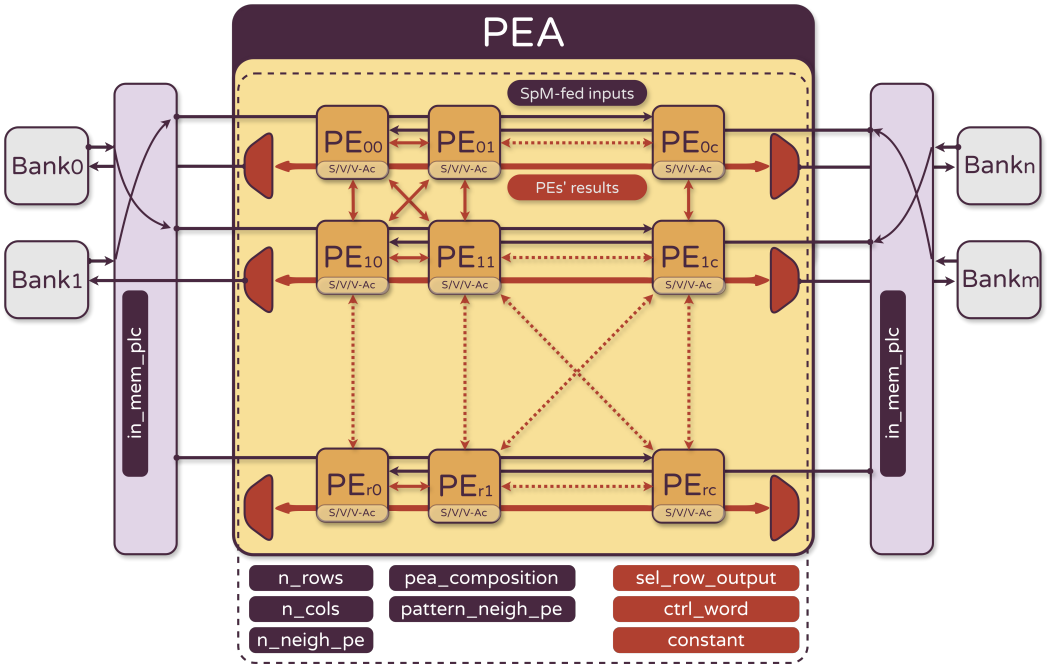


Fig. 4.5 Mage's PEA template-based structure. Template parameters and configuration registers related to the various module are also reported, accordingly to Figure 4.7 and Figure 4.6.

approach. First, it introduces array-level aspects, including template-based PEA sizing, PE placement, and the supported interconnection topologies. Then, it moves to the PE level, presenting the available microarchitectures and the instructions they implement.

Template-based PEA Structure & Connectivity Mage uses template-based generation for the PEA hardware description. Template parameters define ① the array dimensions (number of rows and columns), ② the PE interconnect topology, and ③ the PE variant instantiated at each array location.

As shown in Figure 4.1, Mage is typically organized with AGEs and SpM banks distributed on both sides of the PEA. While this symmetric placement is not a strict constraint, it is the natural configuration used to maximize bandwidth toward the array by balancing memory access resources across the fabric. Under this organization, an effective rule of thumb is to size the PEA such that the number of rows matches half of the total number of AGEs and SpM banks, so that each side can feed a corresponding portion of the array. However, template parameters

`n_rows` and `n_cols` can be used to select the number of rows and columns of the PEA, respectively.

As regards connectivity, the interconnect template supports any topology that is a subset of nearest-neighbor links along the horizontal, vertical, and diagonal directions. This includes the N2N topology, which is also used in the Mage instances discussed in later sections. In this case, template parameters include support for specifying the number of neighbors of each PE (`n_neigh_pe`), as well as the interconnection topology to be applied (`pattern_neigh_pe`). Besides neighbor-to-neighbor communication, PEs can also receive operands from the SpM. In particular, all PEs in the same row share access to the same set of SpM-fed inputs. The accessible inputs for each row are selected through the template parameter `in_mem_plc`. Note that this parameter selects among the outputs of the crossbars placed between the SpM banks and the PEA, which are described in subsection 4.2.4. As regards outputs, each row includes multiplexers on both the left and right sides that select, independently for each side, which PE result in that row is forwarded as the row output. In order to provide selectors to these multiplexers, configuration registers are employed (`sel_row_output`).

Finally, Mage can instantiate heterogeneous arrays. A placement template parameter (`pea_composition`) selects, for each PE at row i and column j, which type of PE to be used, thus enabling the generation of heterogeneous PEAs for exploring different area-performance trade-offs.

Processing Element Architecture This section describes the internal structure of Mage's PEs, focusing on ① the adopted ISA and its implementation in the available FUs, ② local storage and input-operand selection, and ③ the mechanism used to support accumulation operations.

Since a key requirement for Mage is sustaining $II = 1$, the PE datapath is designed to be **fully pipelined**. Therefore, even when an instruction requires a latency larger than one clock cycle, the PE is structured so that it can accept new operands every cycle, and produce a new result every cycle once the pipeline is filled. In other words, PEs internal structure implements all instructions in a fully pipelined fashion, sustaining one new output per cycles once the pipeline is filled, even though instructions can have variable latencies, which are taken into account during the scheduling phase described in section 4.3.

Table 4.1 Instructions supported by Mage PEs, along with their opcode and vector mode in the employed ISA and their function. Although not related to the ISA by definition, latencies are also reported for the different PE variants Mage implements.

Name	Opcode	Vector Mode	Function	Latency		
				Scalar	Vector	Vector Accumulation
NOP	00000	00	-	-	-	-
FW_A	00111	00	Forwards first FU's input	1	1	1
FW_B	01000	00	Forwards second FU's input	1	1	1
ADD32	00010	00	$a + b$ on int32 data	1	1	1
SUB32	00011	00	$a - b$ on int32 data	1	1	1
MUL32	00001	00	$a \cdot b$ on int32 data	1	2	2
LSH32	00100	00	$a << b$ on int32 data	1	1	1
ARSH32	00101	00	$a >> b$ on int32 data	1	1	1
LRSH32	00110	00	$a >> b$ on int32 data	1	1	1
ADD16	00010	01	$a + b$ on int16 data	-[a]	1	1
SUB16	00011	01	$a - b$ on int16 data	-[a]	1	1
MUL16	00001	01	$a \cdot b$ on int16 data	-[a]	1	1
LSH16	00100	01	$a << b$ on int16 data	-[a]	1	1
ARSH16	00101	01	$a >> b$ on int16 data	-[a]	1	1
LRSH16	00110	01	$a >> b$ on int16 data	-[a]	1	1
ADD8	00010	10	$a + b$ on int8 data	-[a]	1	1
SUB8	00011	10	$a - b$ on int8 data	-[a]	1	1
MUL8	00001	10	$a \cdot b$ on int8 data	-[a]	1	1
LSH8	00100	10	$a << b$ on int8 data	-[a]	1	1
ARSH8	00101	10	$a >> b$ on int8 data	-[a]	1	1
LRSH8	00110	10	$a >> b$ on int8 data	-[a]	1	1

[a] Scalar PEs can perform these operations but considering inputs and output as int32, thus without performing the required truncation on the specified data format.

The ISA implemented by Mage's PEs defines a 16-bit instruction word composed of two 4-bit fields for selecting the FU input operands, 5 bits for the opcode, 2 bits for selecting the **packed-SIMD** mode (also referred to as *vector mode*), and 1 bit for enabling accumulation from ADD, when supported by the PE. The supported operation set is mainly centered on shifts, addition/subtraction, and multiplication, which is sufficient to cover a wide range of SCoP kernels and, in particular, General Matrix Multiplication (GeMM) workloads. Table 4.1 reports the operations supported by Mage, together with their opcode and vector mode fields in the ISA. The main difference between instructions with the same functionality, such as ADD32, ADD16, and ADD8, lies in the input data type they operate on and in the corresponding result truncation. Furthermore, for 8- and 16-bit inputs, the packed-SIMD modes

of `ADD`, `SUB`, `MUL`, and shift instructions can be exploited to increase parallelism, as discussed below. Moreover, the opcode space leaves room for extending the ISA with additional instructions when needed.

Mage can generate heterogeneous PEAs to explore different area-performance trade-offs. In practice, the PEA can instantiate **three PE variants**, which mainly differ in the supported packed-SIMD modes and in the availability of accumulation. The variants are described in the following, highlighting the FU organization and how it covers the corresponding ISA subset.

1. **Scalar PEs** implement the full ISA in scalar form, i.e., operating on 32-bit inputs only and without packed-SIMD modes. Their FU includes one 32-bit adder/subtractor, one 5-bit left barrel shifter, and one 32-bit multiplier. The left shifter is also used to support right shifts through a bit-reversal strategy. This operator set covers the ISA described in Table 4.1 and can also support straightforward extensions, such as comparison operations, or fused multi-operator instructions (e.g., a `MULADD`) that can be beneficial for mapping density.

2. **Vector PEs** implement all ISA instructions, while supporting packed-SIMD execution on 8- and 16-bit data, as well as 32-bit computation. As a results, their FU is capable of performing *one 32-bit operation, two parallel 16-bit operations, or four parallel 8-bit operations*, depending on the selected SIMD mode, always employing a pipelined execution fashion. This is achieved through the following operators:

   (a) A **36-bit partitioned adder/subtractor**, inspired by the adder present in the Arithmetic-Logic Unit (ALU) of the RISC-V *CV32E40P* CPU. Two 32-bit inputs are split into four 8-bit slices and mapped onto bit ranges $[1{:}8]$, $[10{:}17]$, $[19{:}26]$, and $[28{:}35]$ of the 36-bit adder, thus creating four independent 8-bit **lanes** separated by guard bits. Bits $(0, 9, 18, 27)$ are used to control carry propagation across lane boundaries. When a 32-bit add/sub is required, these bits are set to allow full carry propagation. When two 16-bit or four 8-bit operations are required, the corresponding boundary bits are cleared to block carry propagation and keep the lanes independent. Bit 0 also selects between addition and subtraction. Packed-

SIMD `ADD` and `SUB` have a one-cycle latency, while also remaining fully pipelined.

(b) **Three 16-bit multipliers and one 8-bit multiplier** used to implement packed-SIMD `MUL`. For four 8-bit multiplications, the operands are split into four 8-bit slices and dispatched to the four multipliers. For two 16-bit multiplications, the operands are split into two 16-bit halves and mapped onto two of the 16-bit multipliers. For 32-bit multiplication, the FU computes the lower 32 bits of the product using three 16-bit partial products. Let $A = A_{31:16}\,2^{16} + A_{15:0}$ and $B = B_{31:16}\,2^{16} + B_{15:0}$. The lower 32-bit result can be written as:

$$M = A_{15:0} \cdot B_{15:0} + \left(A_{15:0} \cdot B_{31:16} + A_{31:16} \cdot B_{15:0}\right) \cdot 2^{16} \qquad (4.2)$$

The three 16-bit multipliers compute $A_{15:0} \cdot B_{15:0}$, $A_{15:0} \cdot B_{31:16}$, and $A_{31:16} \cdot B_{15:0}$ in parallel. Their outputs are then combined through an adder stage, after applying the required 2^{16} shift to the cross terms. In the current implementation, the partial products are first temporarily stored in registers, and the final combination is performed in the following cycle, resulting in a two-stage pipeline for 32-bit `MUL` while still sustaining one result per cycle after pipeline is filled.

(c) A *partitioned left barrel shifter*, inspired by the shifter of the RISC-V *CV32E40P* CPU's ALU. It implements `LSH` (left shifting) on 32-bit data with a 5-bit shift amount, on 16-bit data with two independent 4-bit shifters, and on 8-bit data with four independent 3-bit shifters. Right-shift instructions (`ARSH` and `LRSH`) are implemented through a bit-reversal strategy, combined with either arithmetic or logical fill as required. All shift instructions execute in one cycle and remain fully pipelined.

3. **Vector Accumulation PEs** implement the full ISA, including packed-SIMD instruction, providing also support for accumulation. It is important to highlight that additional operators are needed to implement accumulation while maintaining fully-pipelined execution. Therefore, these PEs use the same operator set as *Vector PEs*, while also extending it with additional adders to support accumulation on 8-, 16-, and 32-bit data without breaking the fully pipelined execution model. For 32-bit accumulation, the FU feeds back the

current sum to the partitioned adder/subtractor. For 16-bit and 8-bit accumulation, additional adders are used to combine the lane results produced by the 36-bit adder. An additional partitioned 18-bit adder is used to sum the two 16-bit lanes or, in the 8-bit case, to sum two pairs of 8-bit lanes. Furthermore, a final 8-bit adder is used to combine the two partial sums when accumulating 8-bit data. With this organization, lane reduction and feedback are performed in parallel with the main FU pipeline, so the PE can continue accepting new inputs every cycle while accumulation progresses in a pipelined manner.

As regards accumulation, PEs implementing it receive an additional 1-bit control input directly from the AGEs on the same row (i.e., from the two AGEs placed on the left and right sides of that row). This signal, if asserted, indicates when the current accumulation must be closed. In this case, the PE finalizes the current accumulation, makes it available for write-back to the SpM, and then starts a new accumulation window. *This is the only direct connection between the access and the execute paths.*

As regards inputs, all three PE variants receive the same input sources:

1. **Neighbor results**: up to 8 inputs, depending on the selected PEA connectivity.
2. **SpM subsystem inputs**: values delivered by the SpM through the SpM $\leftrightarrow$PEA crossbar described in subsection 4.2.4. Each SpM-fed input has an identifier, and the set of identifiers visible to each PE row is selected through the template parameter `in_mem_plc`.
3. **Constant operand**: a 32-bit register accessible from the external system.
4. **Local feedback**: the PE's own result, used for accumulations.

Two input multiplexers select the FU operands from the available sources. While the exact number of selectable sources depends on the instantiated topology and on `in_mem_plc`, the 4-bit selection fields allow up to 16 choices per operand, which covers the supported configurations. On the output side, the FU result is broadcast to PE's neighbors, so it can be consumed by adjacent PEs according to the selected connectivity.

PEA Configuration Registers

PEA

sel_row_output	For each PEA row and for each side, selects which PE result to expose to the PEA → SpM crossbar.

PE

ctrl_word	PE's control word including instruction opcode and two operand selectors for the FU's inputs.
constant	32-bit constant operand register, accessible from the external system, not writable by the PE.

Fig. 4.6 Configuration registers of Mage used by the PEA and PEs.

PEA Template Parameters

Sizing

n_rows	number of PEA rows
n_cols	number of PEA columns

Composition

pea_composition	List where entry i selects the PE type instantiated at position $row = \lfloor i/C \rfloor$ and $column = i \bmod C$, with C the number of PEA columns

Connectivity

n_neigh_pe	Number of neighbor PEs considered by each PE for inter-PE communication
pattern_neigh_pe	Neighborhood pattern: list of neighbor positions defining, for each PE, which surrounding PEs are reachable.
in_mem_plc	Row-wise selection of SpM-fed inputs: for each PEA row, lists which outputs of the SpM→PEA crossbars are exposed as input sources to the PEs in that row.

Fig. 4.7 Template parameters to define the structure of the PEA.

4.2.4 Crossbars and Access-Execute Channels

Mage employs template-configurable crossbars to connect the access and execute paths around the multi-bank SpM. A first set of crossbars is placed between the AGU and the SpM subsystem, enabling AGEs to issue load/store requests to the SpM ($AGU \rightarrow SpM$). A second set of crossbars is placed between the SpM subsystem and the PEA, enabling SpM banks to feed operands to the PEs and enabling PE results

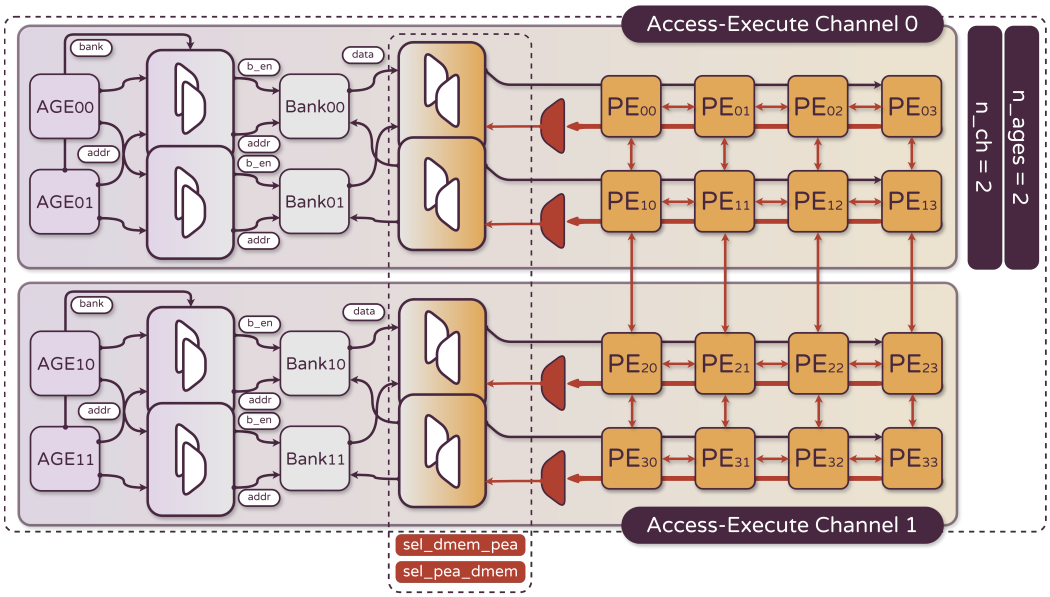


Fig. 4.8 Mage AGU→SpM and SpM↔PEA template-based structure. For better understanding, *n_ch* and *n_ages* have been set to two in the reported illustration. Each AGE and bank has two subscripts, the first indicating the channels it belongs to, and the latter indicating its index within the channel. For the Mage implementation reported in figure, other two channels are present on the right side, but they are not shown for better clarity. Template parameters and configuration registers related to the various module are also reported, accordingly to Figure 4.10 and Figure 4.9. Referring to subsection 4.4.1, the reported Mage variant is *4x4_4x2*, without the two right-side channels.

to be written back to the SpM (*SpM* ↔ *PEA*). In Figure 4.8, two access-execute channels are illustrated, along with its internal composition.

To keep interconnect complexity scalable, Mage does not necessarily build a single global crossbar spanning all AGEs, banks, and PEA rows. Instead, template parameters partition AGEs, SpM banks, and PEA rows into **access-execute channels**. Each channel defines a *local communication domain*. In other words, a group of AGEs can access only a corresponding group of SpM banks, and those banks, in turn, are connected to a corresponding set of PEA rows. For each channel, Mage instantiates the required *AGU* → *SpM* and *SpM* ↔ *PEA* crossbars. This structure reduces multiplexers size and routing pressure, while still providing enough flexibility for workloads mapping.

The number of channels is set through the template parameter `n_ch`. Within each channel, additional template parameters define the size of the AGE, bank, and PEA-row groups. In its current implementation, Mage supports *equally sized groups*

across channels through a single parameter (`n_age_per_ch`), which defines the number of AGEs per channel, while also implicitly fixing the number of banks and PEA rows per channel. This choice promotes balanced use of AGEs across multiple banks and balanced bandwidth toward the connected PEA rows. Nevertheless, the same template-based approach could be extended to support heterogeneous channels, if required by specific mappings or system constraints. In addition, while this constraint applies to the internal channel composition, different access-execute channels can be generated with different size. In the following, a more detailed description of Mage crossbars is provided, as well as the available trade-offs.

AGU $\rightarrow$ SpM crossbars The *AGU* $\rightarrow$ *SpM* crossbar routes each AGE request to the target SpM bank within the selected access-execute channel. The destination bank is determined by the AGE itself, since each AGE computes a GMP-compliant bank index along with the address for that bank. As a result, the crossbar selection is driven by the bank index generated by the AGE. Mage's computation model guarantees that, once a valid mapping is found, the schedule avoids bank conflicts by construction and the design sustains $II = 1$ for maximum throughput.

SpM $\leftrightarrow$ PEA crossbars On the *SpM* $\rightarrow$ *PEA* direction, crossbars select which bank outputs are exposed as operand inputs to each PEA row. Conversely, on the *PEA* $\rightarrow$ *SpM* direction, crossbars select which PE results are written back to which banks. Differently from the *AGU* $\rightarrow$ *SpM* case, selections are not produced dynamically by the PEs. On the other hand, they must be set through configuration registers, as they depend on the mapping of the DFG onto the PEA, and they remain constant during workload execution. In other words, these crossbars implement the static connections between memory and the PEA rows required by the mapped schedule.

Trade-offs The access-execute channel partitioning exposes a direct trade-off between flexibility and cost. Larger channels increase routing flexibility and simplify mapping, but they also increase crossbar sizes and selector width. Smaller channels reduce area and routing complexity, but they restrict which AGEs can access which banks and which rows can consume which bank outputs. In the evaluated configurations, channels are chosen so that the intended kernels can be mapped without bank

conflicts while keeping the interconnect local and regular. In addition, section 4.4 also evaluates the area impact of increasingly complex access-execute channels.

SpM ↔ PEA Crossbars Configuration Registers

Selectors

sel_dmem_pea	For each PEA row, selects which SpM bank output (within the same access-execute channel) is forwarded as an input source to the PEs of that row.
sel_pea_dmem	For each SpM bank, selects which PEA row output (within the same access-execute channel) is forwarded to the bank write-data input.

Fig. 4.9 Configuration registers of Mage used by the SpM ↔ PEA crossbar.

Crossbars Template Parameters

SpM ↔ PEA & AGU → SpM Crossbar

n_ch	Number of access-execute channels.
n_age_per_ch	List of **n_ch** elements, in which entry i sets the channel size, i.e., the number of AGEs, SpM banks, and PEA rows that belong to channel i.

Fig. 4.10 Template parameters to define the structure of the SpM ↔ PEA and AGU → SpM crossbars.

4.3 Kernel Mapping and Scheduling Methodology

Mage targets acceleration of SCoP workloads by separating memory access operations from data-related ones, and by relying on a multi-bank SpM with GMP-compliant data layouts to avoid memory access conflicts. In order o maximize

throughput while preserving area and energy efficiency, Mage supports fully-static kernel mapping on AGU and PEA with $II = 1$. Therefore, one loop iteration is started every cycle, and PEA, AGU and crossbars configuration remains constant throughout the execution of the kernel. As a results, *kernel mapping consists in finding a static assignment of memory operations to AGEs, compute operations to PEs, and data layout in SpM such that the access path generates conflict-free SpM accesses every cycle, while the execute path sustains a fully pipelined dataflow*. In this section, constraints to be applied to kernels mapping are discussed, along with the adopted methodology flow to be adopted for mapping workloads onto Mage. It is important to highlight that the described methodology has been *manually applied* for mapping kernels analyzed in the results section (section 4.4). It should be noted that the reported benchmark set, while covering a range of loop structures and access patterns, was selected based on what could be mapped by hand. A rigorous assessment of the general tractability of this scheduling and data-layout problem is left to future work on automated mapping.

Before describing the mapping flow, a valid mapping on Mage must satisfy a set of constraints on both access and execute paths:

1. Access **Resource limits:** the number of load/store operations issued per cycle cannot exceed the number of available total AGEs.
2. Access **Conflict-free memory access:** in each cycle, the set of accesses generated by active AGEs must target distinct SpM banks. This is enforced by the combination of GMP-compliant layouts and a schedule that avoids concurrent accesses to the same bank.
3. Execute **Resource limits:** the number of data-related operations per cycle cannot exceed the number of available total PEs.
4. Execute **PE connectivity:** compute nodes must be mapped onto PEs such that operands can be routed through the selected topology and through the SpM→PEA inputs exposed to each row.
5. Execute **Pipeline consistency:** since $II = 1$ requires that new data is consumed every cycle, the mapped dataflow must be arranged so that operator latencies and routing delays are aligned.

Taking into account the discussed constraints, the kernel mapping methodology follows four steps discussed in the following. For better understanding, an example is also provided, following the methodology employed for mapping the matrix multiplication (*matmul*) kernel onto a Mage implementation having a 4×2 PEA and a 4×2 AGU.

Algorithm 1 Matrix Multiplication (*matmul*) `C` code with Unroll Factor of 2 on the inner loop.

```
1: for i ← 0 to A_rows − 1 do
2:     for j ← 0 to B_columns − 1 do
3:         for k ← 0 to A_columns − 1 (k = k+2) do
4:             C[i][j] ← C[i][j] + A[i][k] · B[k][j] + A[i][k+1] · B[k+1][j]
5:         end for
6:     end for
7: end for
```

① **C code implementation.** Although this step can be considered as a preliminary one, it is fundamental to have a clear idea of the access and execute operations to be carried out on Mage. Hence, this step mainly consists in **implementing a `C` version of the workload to be deployed on Mage**, along with the selected Unroll Factor (UF) on the target loop. This process allows to check whether the kernel can be effectively mapped onto Mage by verifying that:

1. the kernel is a **SCoP workload**, hence array indices and any static-control predicates are affine functions of loop IVs.
2. the nested **loop structure is supported**, hence Mage's HLU implements the required number of loops, with iterators having fixed initial and final values, with fixed increment, as described in subsection 4.2.2.
3. the **operators are available** according to the employed Mage's PEs.
4. the **array access patterns are supported** by Mage's AGEs, as described in subsection 4.2.2.

In addition, this step is also useful to verify constraint 1 and 3, by just counting the number of arithmetic operations and load/store array accesses, as they must

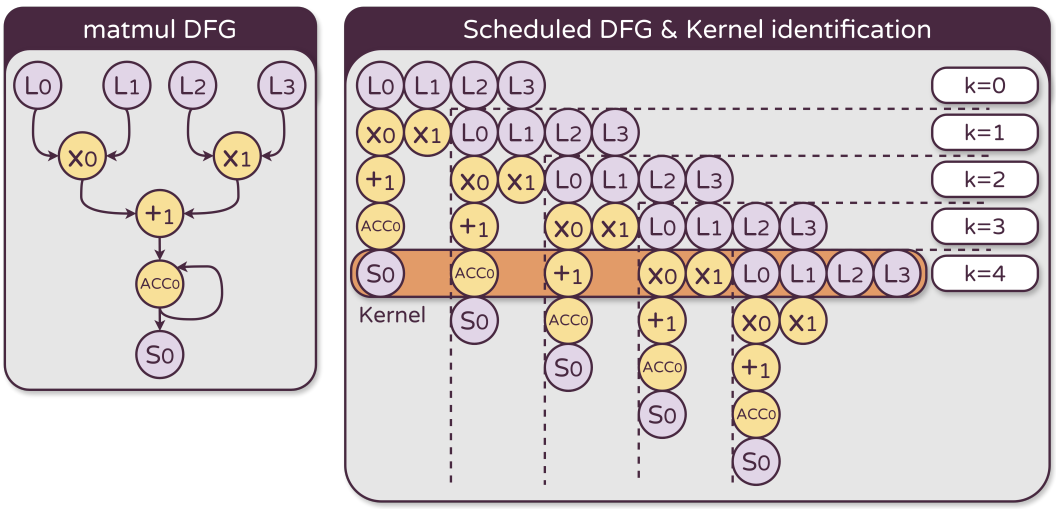


Fig. 4.11 *matmul* DFG with $UF = 2$ along with the related ALAP scheduling and kernel identification.

all concurrently fit both AGU and PEA, respectively. For the considered *matmul* example, Algorithm 1 provides the `C` implementation of the workload with $UF = 2$ on the inner loop.

② **DFG extraction.** Starting from the `C` implementation, **the workload is translated into a DFG representation**, as shown in Figure 4.11. The loop nest is expressed as a DFG including computation nodes (arithmetic operations) and memory nodes (load/store operations) available in each iteration. This representation explicitly identifies the involved arrays and the required load/store streams, and, most importantly, the **data dependencies** among load/store operations and arithmetic ones, as well as the presence of reductions, such as *accumulations*.

③ **Scheduling and Data Layout.** This step mainly consists in **assigning each operation in the DFG with a time slot** and ensure that a **GMP-compliant data layout can be assigned to each array**. First, a tentative scheduling of operations is carried out using an **As Late As Possible** (ALAP) algorithm. Therefore, each node is assigned the last possible execution time slot that satisfies its data dependencies, thus creating a *scheduled DFG*. Once this is done for a single iteration, as execution on Mage employs $II = 1$ and a new iteration is started at each cycle, the scheduled DFG is replicated a number of times needed to identify the **kernel**, as shown in Figure 4.11. In this context, *the kernel is the set of operations belonging to different iterations that are repeatedly executed until the end of execution.* Therefore, this

set of operations constitutes the group of load, store and arithmetic operations to be mapped onto PEA and AGU.

Once the kernel is identified, constraint 2 can be checked. As a matter of fact, knowing the set of concurrent load and store operations allows to verify whether a valid (B_s, N_b) couple can be found for each accessed array to ensure conflict-free execution with the initial ALAP schedule. However, it is important to note that array indices must be adapted to the related iteration within the kernel. In the provided example, **different arrays are accessed in different iterations** (C matrix is accessed at iteration 0, while A and B are both accessed at iteration 5), thus no inter-iteration constraints are present on data layout. However, two load operations must be concurrently carried out for both matrices A and B, and this implies constraints on data layout for both matrices. As for matrix A, L_0 and L_1 are associated to $A[i][k]$ and $A[i][k+1]$, which transform in $A[i][k+5]$ and $A[i][k+6]$ considering that they are executed at iteration 5 in the kernel. However, this transformation is not needed here, as the additional offset is the same for both arrays. Hence, the distance $d_{L_0-L_1} = 1$, which is the distance between the two indices used to access matrix A. Since the distance between two concurrent indices is always 1 for A, a valid (B_s, N_b) couple for A data layout can be $(1,2)$, which implies interleaving A on two consecutive banks in Mage's SpM. As regards matrix B, the two load accesses are $B[k+5][j]$ and $B[k+6][j]$, whose distance is $d_{L_2-L_3} = B_{columns}$. In this case, the couple $(1,2)$ can be applied if B has an odd number of rows, while, if it had an even amount of rows, zero-padding would be needed and loop bounds changed to emulate a different number of columns.

Nevertheless, a possible data-layout allowing conflict-free execution is found, and the process can go to the next step. On the other hand, if a conflict-free solution could not be found, a different schedule would be needed for the DFG, most of the time requiring the moving of access operations in different time slots, thus making the scheduling problem an iterative process.

④ **Mapping.** This step mainly requires **assigning each load/store operation with an AGE, as well as each arithmetic operation with a PE and each array with an SpM group of banks**. Therefore, in this step, constraints 4 and 5 can be checked, ensuring that operations can be mapped onto PEs such that inputs, intermediate results and outputs can flow in the PEA sustaining fully-pipelined execution enabled by the conflict-free access to SpM. After carrying out these

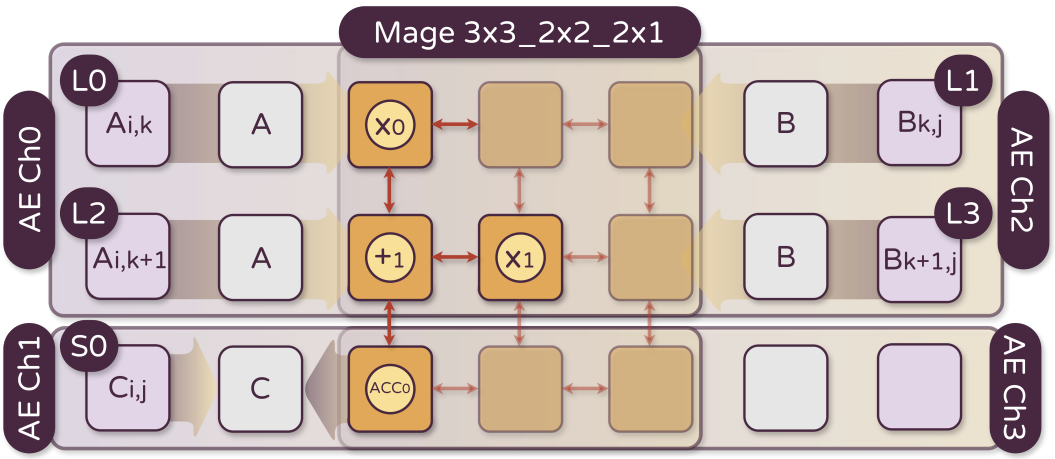


Fig. 4.12 Mapping of *matmul* DFG with $UF = 2$ on a Mage implementation with 2 non-homogeneous access-execute channels, in the mage variant *3x2_2x2_2x1* presented in subsection 4.4.1.

assignments, configuration is extracted for AGEs, PEs and other components, such as the HLU and the crossbars.

As for AGEs, they are configured with the strides of the related access operations (mirroring the array indices in the `C` code), as well as the N_b and B_s of the corresponding array in SpM. Also, the set of HLRF IVs they operate on is chosen according to the time slot they are scheduled onto. In addition, if the related load or store operation needs to be constrained to be executed only on certain iteration, AGEs and HLDU are configured accordingly. As for PEs, their instruction word is composed according to their assigned operation and set on inputs. Moreover, SpM↔PEA crossbars' selector are set in order to ensure the proper connections among banks and PEs operating on input data. Furthermore, AGU's HLU is configured with the proper loop bounds.

Overall, once a mapping is found with the discussed methodology satisfying the constraints, Mage runs in steady state with $II = 1$. Therefore, AGEs and PEs remain statically configured, bank conflicts do not occur due to proper scheduling, and the access and execute paths deliver maximum throughput after pipeline fill. Therefore, for each workload, execution cycles always closely match the total amount of loop nest iterations, which is further reduced in packed-SIMD execution is performed with vector PEs.

4.4 Results & Discussion

In this section, Mage is evaluated across multiple architecture variants and workloads. First, subsection 4.4.1 summarizes the considered Mage implementations and the explored design knobs, such as PEA, AGU and SpM size, access-execute channel partitioning, and scalar vs. vector PE composition. Then, subsection 4.4.2 describes the evaluation methodology, implementation flow, and the set of workloads used to evaluate Mage. Based on this setup, subsection 4.4.3 discusses area occupation and scaling trends across the different implementations. Finally, subsection 4.4.4 reports performance and efficiency results, including speedup, throughput (MOPS), area efficiency (MOPS/mm^2), and energy efficiency (MOPS/mW), with additional in-depth case studies on two specific kernels that are widely present in Transformer and Convolutional Networks models, as well as an end-to-end evaluation on a Temporal Convolutional Network application.

4.4.1 Mage Implementations

Leveraging Mage's template-based design, several implementations have been generated and analyzed to prove Mage's design flexibility and explore the various trade-offs. The considered variants mainly differ in the scaling of PEA, AGU and SpM resources, as well as in the crossbar complexity between these components. By comparing these implementations, the explored design space highlights the trade-offs between kernel mapping flexibility, achievable parallelism and performance, and area and energy efficiency.

The studied variants are named according to the following scheme. The first $R \times C$ field indicates the PEA size. The second (and, when present, the third) $M \times N$ field describes the access-execute channel partitioning, in which M is the number of channels and N is the channel size, which indicates the number of AGEs, SpM banks, and PEA rows associated with each channel. This choice directly determines the complexity of crossbars within a channel. When a third $M \times N$ field is present, channels are not all homogeneous in size, and two channel classes are instantiated with different crossbar complexity, as shown is the Mage implementation used in Figure 4.11.

For each of the Mage implementations described in this section, two PEA compositions are generated. One variant employs only scalar PEs, while the other variant integrates vector PEs and vector-accumulation PEs in selected positions, enabling accumulation support while balancing area occupation. In the remainder of this chapter, variants employing only scalar PEs are labeled with an additional *_s* in tables and figures, while variants including vector PEs are labeled with an additional *_v*. When no suffix is used, the description refers to both the scalar and vector implementations. The studied Mage variants are summarized in the following.

- **2x2_2x2** uses a 2×2 PEA and 2 access-execute channels of size 2, resulting in small crossbars and a compact access/execute organization. The vector variant includes a vector-accumulation PE at position $(1,0)$.
- **2x4_2x2** uses a 2×4 PEA with 2 channels of size 2, increasing PEA capacity while keeping the access-execute channel granularity unchanged. The vector variant includes a vector-accumulation PE at position $(1,0)$.
- **3x2_2x2_2x1** uses a 3×2 PEA and non-homogeneous channels, with 2 channels of size 2 for the upper PEA rows and 2 channels of size 1, thus mixing wider channels with fully dedicated ones. The vector variant includes vector-accumulation PEs at positions $(1,0)$ and $(2,0)$.
- **4x2_4x2** uses a 4×2 PEA and 4 channels of size 2, thus increasing PEA size while keeping crossbars small across multiple channels. The vector variant includes vector-accumulation PEs at positions $(1,0)$, $(2,0)$, and $(3,0)$.
- **4x2_2x4** uses a 4×2 PEA and 2 channels of size 4, reducing the number of channels but increasing flexibility in each channel, at the cost of larger crossbars. The vector variant includes vector-accumulation PEs at positions $(1,0)$, $(2,0)$, and $(3,0)$.
- **4x4_4x2** uses a 4×4 PEA and 4 channels of size 2, thus combining a larger PEA with relatively small per-channel crossbars. The vector variant includes vector-accumulation PEs at positions $(1,0)$, $(2,0)$, and $(3,0)$.
- **4x4_2x4** uses a 4×4 PEA and 2 channels of size 4, thus increasing routing flexibility compared to the previous variant. The vector variant includes vector-accumulation PEs at positions $(1,0)$, $(2,0)$, and $(3,0)$.

- **5x2_2x4_2x1** uses a 5×2 PEA and non-homogeneous channels, with 2 channels of size 4 for upper PEA rows and 2 channels of size 1, combining high-flexibility channels with low-cost dedicated ones. The vector variant includes vector-accumulation PEs at positions $(1,0)$, $(2,0)$, $(3,0)$, and $(4,0)$.

The considered subset of implementations allows investigating how mapping flexibility (as well as the related area/performance/energy metrics) varies with the scaling of PEA and AGU/SpM resources, and the access-execute channel partitioning, by also comparing configurations where PEA, AGU, and SpM sizes are kept fixed while only channel granularity is changed.

For all the described Mage implementations, the HLDU is configured to support up to 4 nested loops, with induction variables up to 8 bits wide. The motivations behind this choice are twofold. First, supporting 4 nested loops covers a broad range of relevant workloads. Second, four 8-bit IVs fit naturally in a 32-bit configuration register, simplifying configuration and storage. The corresponding limitation is that each loop bound is constrained to the range $[0, 2^8)$. Nonetheless, this bound is not restrictive for the workloads considered in this thesis, and larger iteration spaces could still be handled by properly rewriting array addressing functions and/or exploiting strides for each iterator. As regards SpM sizing, all banks store 1024 32-bit words in all variants, corresponding to 4 KiB per bank.

4.4.2 Experimental setup

To evaluate the considered Mage variants, each configuration is generated through Mage's template-based flow and integrated into an X-HEEP-based microcontroller system as a loosely coupled accelerator. Across all implementations, the X-HEEP platform is configured with 320 KiB of main system memory, organized as ten 32 KiB banks that support interleaved data placement, the OpenHW Foundation CV32E40P RISC-V core [59] implementing the *XCVPULP* ISA extensions (e.g., SIMD operations on *int16* and *int8* data, hardware loops, etc.), and a peripheral subsystem including a single DMA channel and other components, such as timers, SPI, I2C and I2S subsystems.

All Mage variants, together with their corresponding X-HEEP systems, have been synthesized using *Synopsys Design Compiler* targeting a commercial 65 nm technology node, at 250 MHz under worst-case conditions (slow-slow corner, 1.08 V,

Table 4.2 Benchmarked SCoP kernels used to evaluate Mage.

Kernel	Description
bicg	computes two matrix-vector products ($q = A\,p$ and $s = A^T\,r$) using *two nested loops*.
atax	computes $y = A^T\,(A\,x)$ through an intermediate vector, thus using *an outer loop with two consecutive inner loops over the same iterator, i.e., an imperfectly nested loop structure*.
firstdiff	computes adjacent differences of a 1D signal using *a single loop*.
hydro	computes the hydrodynamics fragment combining scaled neighbor terms ($x[k] = q + y[k]\,(r\,z[k+10] + t\,z[k+11])$) with *a single loop*.
gemm	computes the general matrix multiplication with output matrix initialization and scaling ($C = \beta C + \alpha AB$), using *an outer loop with an imperfectly nested structure, i.e. a single inner loop over j to initialize C followed by a perfectly nested j˘k loop nest for computing the resultin C.*.
matmul	computes baseline matrix multiplication without α/β scaling ($C = AB$) using *three perfectly-nested loops*.
conv1d	computes 1D convolution with *two perfectly-nested loops*.
conv2d	computes 2D convolution with *three perfectly-nested loops*.
dconv1d	computes dilated 1D convolution on multiple input and output channels using *four nested loops with bias addition outside of the main inner loop body*.

125 °C) using only low-V_t cells. In addition, post-synthesis simulations have been performed with *Siemens QuestaSim* to collect switching activity in `vcd` files, which have been exploited by *Synopsys PrimeTime* to estimate power consumption during the execution of the considered benchmarks for both Mage implementations and the related CPU baselines.

Mage is evaluated on **nine SCoP kernels** taken from, or inspired by, the *PolyBench* and *Livermore* suites, and they are summarized in Table 4.2. All kernels are mapped onto Mage using different *unroll factors* (UFs) applied to the innermost loop, in order to explore the impact of parallelism on throughput and resource usage. For each mapping, the configuration of AGU, PEA, and crossbars, together with the required GMP-compliant SpM layouts, is extracted and encapsulated in dedicated

C wrapper functions. For every benchmark, a *C* test program is built in which ① Mage is programmed through memory-mapped configuration registers, ② the DMA is exploited to transfer input data from X-HEEP main memory to Mage's SpM with the required layout, and ③ the same kernel is executed on the CV32E40P to be used as a baseline.

Correctness is validated by comparing Mage outputs against the reference results produced by the CV32E40P execution. Execution cycles are measured for both Mage and the CV32E40P using a timer peripheral, and speedup is computed from the measured cycle counts. Switching activity is collected by triggering `vcd` dumping directly from *C* code. Dedicated functions toggle a GPIO in the X-HEEP peripheral subsystem, and the *SystemVerilog* testbench uses this signal to start/stop `vcd` annotation around the region of interest.

As regards power consumption calculation, Mage power is computed from its internal switching activity, while system-level measurements include the CV32E40P CPU, main system memory, and the system bus contributions. The DMA contribution is not included in Mage power for two reasons. First, it is not part of Mage itself and depends on the host SoC integration. Second, at application level, the required data movement may be dominated by transfers involving external memories, in which case a DMA-like transfer is present regardless of whether the computation is run on Mage or on the CPU.

Finally, all kernels are evaluated on Mage variants featuring only scalar PEs and are compared against compiler-generated code targeting the *XCVPULP* extensions on the CV32E40P. Variants including vector PEs are evaluated on *matmul* and *dconv1d* and are compared against highly optimized CPU implementations based on *builtin* functions that explicitly exploit *XCVPULP* SIMD instructions for `int16` and `int8` data. Although the compiler can infer some SIMD and other instructions inicluded in the *XCVPULP* extension, manual tuning provides consistently higher performance.

4.4.3 Evaluation of Area Occupation and Scaling

This subsection evaluates the area occupation of the considered Mage implementations, and discusses how area scales with the main architectural knobs, namely PEA sizing, number of SpM banks, access-execute channel partitioning, and the adoption of vector PEs. **Area breakdown** results for both scalar and vector variants are

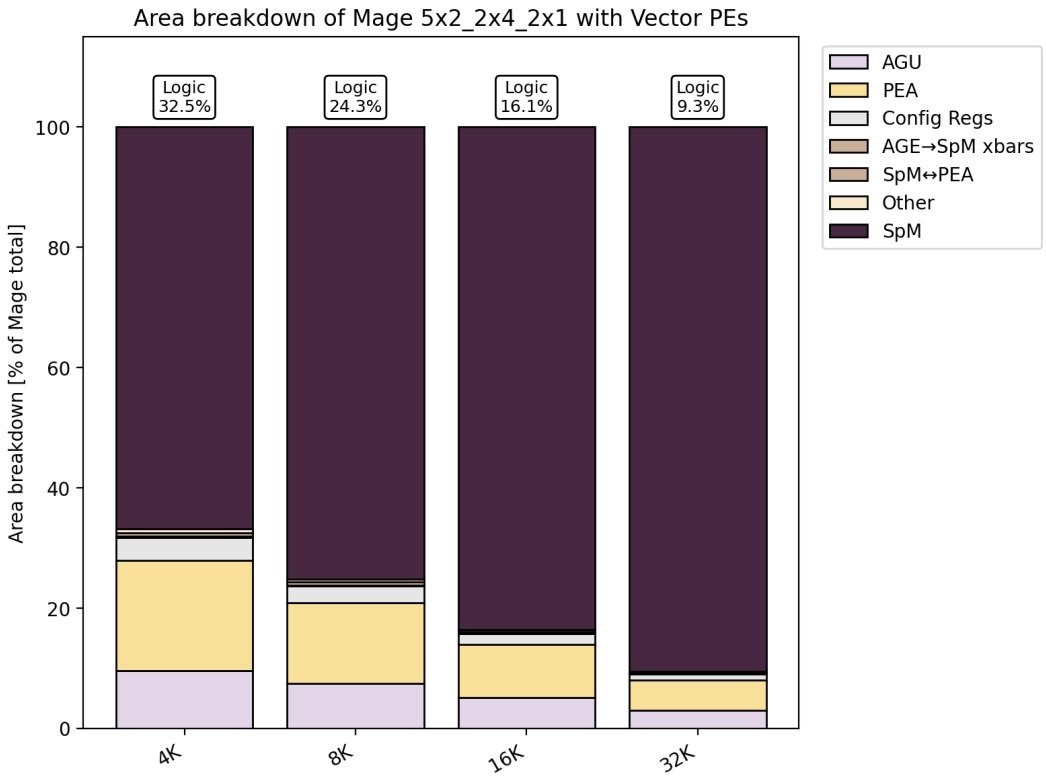


Fig. 4.13 Area breakdown of Mage **5x2_2x4_2x1_v** throughout scaling of memory banks from 4KiB to 32KiB each.

reported in Table 4.3. Moreover, Figure 4.14 and Figure 4.15 provide a more detailed view of the internal breakdown for *4x2_2x4* in the scalar and vector configurations, respectively.

Across all implementations, **the SpM is the dominant area contributor**. Its percentage occupation ranges from 55.15% in *2x4_2x2_v* up to 70.17% in *5x2_2x4_2x1_s*. The evolution of SpM occupation percentage depends on whether the size of compute fabric increases while keeping the same memory capacity, or the memory subsystem itself is scaled. When the number of banks remains constant and the PEA is enlarged by increasing the number of columns, the SpM fraction drops because logic grows while memory stays fixed. This is visible, for instance, when moving from *2x2_2x2* to *2x4_2x2*. In this case, the SpM area remains $150.48 \times 10^3 \mu m^2$, while the PEA roughly doubles, so the SpM fraction decreases from 68.32% to 59.34% in scalar variants and from 65.32% to 55.15% in vector variants.

Table 4.3 Mage area breakdown across **scalar** and **vector** variants (unit: $\mu m^2 \times 10^3$).

Scalar architectures

Component	2x2_2x2_s		2x4_2x2_s		3x2_2x2_2x1_s		4x2_2x4_s	
	Area	%	Area	%	Area	%	Area	%
AGU	26.24	11.91%	26.25	10.35%	35.79	11.04%	45.70	10.56%
PEA	30.73	13.95%	60.75	23.96%	45.78	14.12%	61.88	14.30%
Config Regs	10.53	4.78%	13.82	5.45%	13.94	4.30%	17.49	4.04%
AGE→SpM	0.16	0.07%	0.16	0.06%	0.16	0.05%	0.73	0.17%
SpM↔PEA	0.87	0.39%	0.87	0.34%	0.87	0.27%	3.65	0.84%
SpM	150.48	68.32%	150.48	59.34%	225.72	69.62%	300.96	69.53%
Total	**220.27**	100.00%	**253.59**	100.00%	**324.21**	100.00%	**432.87**	100.00%

Component	4x2_4x2_s		4x4_2x4_s		4x4_4x2_s		5x2_2x4_2x1_s	
	Area	%	Area	%	Area	%	Area	%
AGU	45.17	10.50%	45.69	9.15%	45.16	9.10%	53.96	10.06%
PEA	61.95	14.40%	122.07	24.43%	121.92	24.56%	76.99	14.36%
Config Regs	17.46	4.06%	23.95	4.79%	23.95	4.82%	21.37	3.99%
AGE→SpM	0.32	0.07%	0.73	0.15%	0.32	0.06%	0.73	0.14%
SpM↔PEA	1.74	0.40%	3.65	0.73%	1.74	0.35%	3.65	0.68%
SpM	300.96	69.98%	300.96	60.25%	300.96	60.62%	376.20	70.17%
Total	**430.06**	100.00%	**499.55**	100.00%	**496.51**	100.00%	**536.13**	100.00%

Vector architectures

Component	2x2_2x2_v		2x4_2x2_v		3x2_2x2_2x1_v		4x2_2x4_v	
	Area	%	Area	%	Area	%	Area	%
AGU	26.29	11.41%	26.25	9.62%	35.79	10.54%	45.69	10.07%
PEA	40.72	17.68%	80.00	29.32%	61.15	18.00%	82.63	18.21%
Config Regs	10.58	4.59%	13.86	5.08%	14.02	4.13%	17.56	3.87%
AGE→SpM	0.16	0.07%	0.16	0.06%	0.16	0.05%	0.73	0.16%
SpM↔PEA	0.87	0.38%	0.87	0.32%	0.87	0.26%	3.65	0.81%
SpM	150.48	65.32%	150.48	55.15%	225.72	66.46%	300.96	66.33%
Total	**230.36**	100.00%	**272.88**	100.00%	**339.66**	100.00%	**453.71**	100.00%

Component	4x2_4x2_v		4x4_2x4_v		4x4_4x2_v		5x2_2x4_2x1_v	
	Area	%	Area	%	Area	%	Area	%
AGU	45.17	10.02%	45.68	8.47%	45.16	8.42%	53.97	9.60%
PEA	82.69	18.34%	161.51	29.96%	161.74	30.15%	103.09	18.33%
Config Regs	17.53	3.89%	24.09	4.47%	24.05	4.48%	21.42	3.81%
AGE→SpM	0.32	0.07%	0.73	0.14%	0.32	0.06%	0.73	0.13%
SpM↔PEA	1.74	0.39%	3.65	0.68%	1.74	0.32%	3.65	0.65%
SpM	300.96	66.75%	300.96	55.82%	300.96	56.10%	376.20	66.90%
Total	**450.87**	100.00%	**539.13**	100.00%	**536.43**	100.00%	**562.32**	100.00%

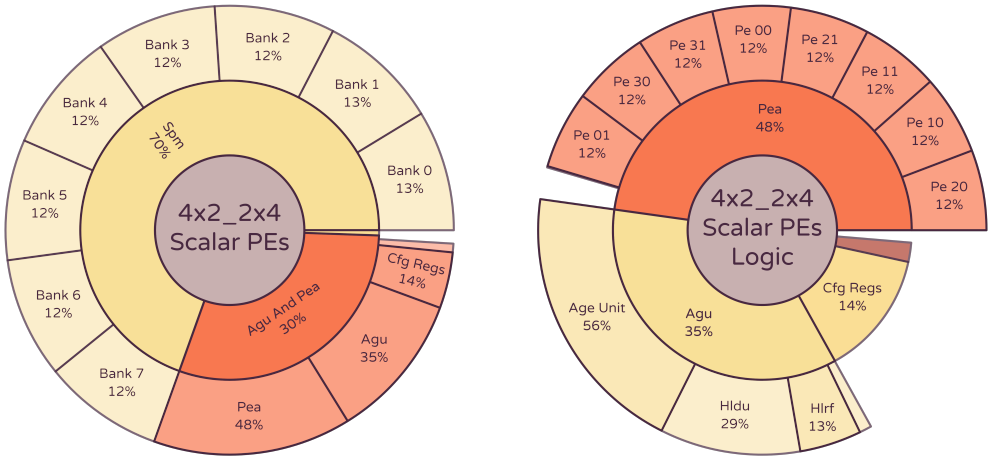


Fig. 4.14 Area breakdown of Mage **4x2_2x4_s**.

Conversely, when increasing the number of PEA rows, the explored variants also increase the number of SpM banks. Therefore, since SpM is the major area contributor and each bank is 4 KiB, the SpM area scales almost linearly with the number of banks: 150.48, 225.72, 300.96, and 376.20 $\times 10^3 \mu m^2$ for 4, 6, 8, and 10 banks, respectively. As a result, **total area increases close to linearly along with the bank number, unless the PEA is also enlarged column-wise**. A similar trend can be observed when increasing the size of each bank, as shown in Figure 4.13 for *5x2_2x4_2x1_v*. In this case, scaling the capacity of each bank from 4 KiB to 32 KiB does not cause a significant increase in AGU area. Indeed, as long as the number of supported nested loops and the IV bit-width remain unchanged, enlarging the bank only requires exploiting a larger portion of the computed address bits. As a result, this sweep progressively shifts the area budget toward memory, while the logic fraction drops from 32.5% to 9.3%. It is also worth highlighting that keeping the same IV bit-width while increasing the per-bank size is not useless. For example, a 32 KiB bank can store up to a 64×128 matrix, which can still be traversed using 8-bit IVs. Although the main set of implementations adopts 4 KiB banks, this sweep is useful to highlight the expected dominance of memory as capacity increases, while both the AGU and the PEA remain unchanged.

The AGU contribution is comparatively stable, remaining around 9-12% of the total area across variants, and its absolute area increases when the number of AGEs increases, as it is expected. The detailed breakdown in Figure 4.14 and

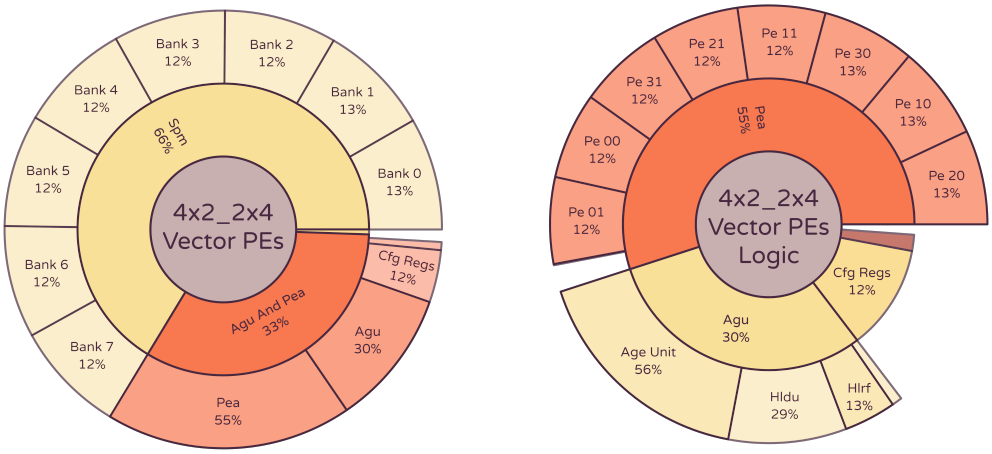


Fig. 4.15 Area breakdown of Mage **4x2_2x4_v**.

Figure 4.15 shows that the AGU area is dominated by the AGE unit (about 56% of the AGU logic), which is mainly due to the multiplier/adder resources used for affine address generation. At the same time, the HLDU contribution is non-negligible (about 29%), due to IV-set selection from the HLRF and the per-AGE gating logic that implements IV-driven constraints. Finally, the AGU fraction decreases in vector variants because the PEA becomes larger, even though the AGU absolute area remains unchanged, as the access side of Mage is not scaled. A further trend emerges when considering only the combinational area (i.e., excluding the SpM macros). **When scaling both AGU and PEA, the relative contribution of the AGU within the overall logic decreases**. For instance, along the *2x2_2x2* $\rightarrow$ *4x2_2x4* $\rightarrow$ *5x2_2x4_2x1* sequence, the AGU percentage drops from about 38% of logic to about 35% and then about 34%. This indicates that, in the explored configurations, PEA-related logic grows faster than the access-side logic when scaling the architecture.

As expected, **the PEA area scales with array size, and its fraction of the total increases when compute is scaled without increasing SpM capacity**. In scalar variants, the PEA share grows from 13.95% in *2x2_2x2* to about 24.4-24.6% in the *4x4* configurations. Introducing vector and vector-accumulation PEs further increases PEA area. For each architecture, moving from scalar to vector increases PEA area by about 32-34% (e.g., 121.92$\rightarrow$161.74$\times 10^3 \mu m^2$ for *4x4_4x2* and 76.99$\rightarrow$103.09$\times 10^3 \mu m^2$ for *5x2_2x4_2x1*). This is consistent with the area of the employed PEs, where **vector and vector-accumulation PEs** (9.6 and 10.3

$\times 10^3$ μm^2, respectively) **are larger than scalar PEs** (7.3×10^3 μm^2). However, since the SpM dominates the overall area, the total area-overhead of vectorization is smaller, ranging from 4.58% up to 8.04% across the considered variants. The effect is clearly visible in Figure 4.14 and Figure 4.15. When moving from scalar to vector in *4x2_2x4*, the PEA percentage occupation within the logic breakdown increases (48% $\rightarrow$ 55%), while the SpM fraction decreases (70%$\rightarrow$66%) but remains the largest contributor. Finally, when scaling designs by adding rows and banks (*2x2_2x2* $\rightarrow$ *4x2_2x4* $\rightarrow$ *5x2_2x4_2x1*), the PEA fraction remains relatively stable (around 14%) because both compute and memory are scaled together and memory remains dominant.

Regarding the remaining components, configuration registers consistently account for about 4-5% of total area. Moreover, crossbar logic remains negligible, as *AGE*$\rightarrow$*SpM* and *SpM*$\leftrightarrow$*PEA* crossbars together stay below 1% in all configurations. Therefore, **the effect of access-execute channel partitioning on area is limited**. For example, wider channels increase crossbar area (e.g., *SpM*$\leftrightarrow$*PEA* is 3.65 vs. 1.74 $\times 10^3$ μm^2 in *4x2_2x4* vs. *4x2_4x2*), but the total area difference remains small (432.87 vs. 430.06 $\times 10^3$ μm^2 in scalar). This indicates that channel partitioning is mainly a knob for mapping flexibility, wiring locality, and timing closure, rather than a primary contributor to area occupation.

Overall, these results show that **Mage area scaling is primarily governed by SpM capacity**, and secondly by PEA size. **Vector PEs substantially increase the PEA area**, but their impact on the total remains moderate due to the dominance of the SpM. At the same time, the **crossbar and configuration overheads are rather small**, suggesting that the access-execute channel mechanism provides structural flexibility with limited area cost.

4.4.4 Benchmarks Evaluation

This section evaluates the performance of the considered Mage variants on the kernels introduced in subsection 4.4.2. Results are reported in terms of **speedup and energy savings with respect to the CV32E40P baseline** executing either compiler-generated code targeting the *XCVPULP* ISA extensions (*scalar* CPU baseline), or manually optimized code (*simd* CPU baseline). These results are complemented with **throughput and area/energy efficiency metrics**, namely MOPS, area efficiency

Table 4.4 Achievable UFs on Mage architectures and AGU and PEA utilization percentages.

Arch	UF	atax		bicg		firstdiff		hydro		conv1d		conv2d		gemm	
		AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA
2x2_2x2	**1**	75	50	—	—	75	25	—	—	50	75	50	75	100	100
2x4_2x2	**1**	75	25	—	—	75	12.5	100	62.5	50	37.5	50	37.5	100	50
3x2_2x2_2x1	**1**	50	33.3	100	83.3	50	16.7	—	—	33.3	50	33.3	50	66.7	66.7
	2	—	—	—	—	—	—	—	—	83.3	66.7	83.3	66.7	—	—
4x2_4x2	**1**	37.5	25	75	62.5	37.5	12.5	—	—	25	37.5	25	37.5	50	50
	2	—	—	—	—	75	25	—	—	62.5	50	62.5	50	—	—
4x2_2x4	**1**	37.5	25	75	62.5	37.5	12.5	—	—	25	37.5	25	37.5	50	50
	2	62.5	50	—	—	75	25	—	—	62.5	50	62.5	50	75	87.5
	3	—	—	—	—	—	—	—	—	—	—	—	—	—	—
4x4_4x2	**1**	37.5	12.5	75	31.25	37.5	6.25	50	31.25	25	18.75	25	18.75	50	25
	2	—	—	—	—	75	12.5	100	62.5	62.5	25	62.5	25	75	43.75
	3	—	—	—	—	—	—	—	—	—	—	—	—	100	62.5
4x4_2x4	**1**	37.5	12.5	75	31.25	37.5	6.25	50	31.25	25	18.75	25	18.75	50	25
	2	62.5	25	—	—	75	12.5	100	62.5	62.5	25	62.5	25	75	43.75
	3	—	—	—	—	—	—	—	—	—	—	—	—	100	62.5
5x2_2x4_2x1	**1**	30	20	60	50	30	10	—	—	20	30	20	30	40	40
	2	50	40	—	—	60	20	—	—	50	40	50	40	60	70
	3	—	—	—	—	—	—	—	—	70	80	70	80	—	—

(MOPS/mm^2), and energy efficiency (MOPS/mW). The discussion is organized in two parts. First, the above metrics are presented for scalar Mage variants across seven kernels, namely *bicg*, *atax*, *firstdiff*, *hydro*, *conv1d*, *conv2d*, and *gemm*. Second, *matmul* and *dconv1d* are analyzed in more depth, since they are used to compare both scalar and vector Mage implementations and to study additional aspects, including the impact of configuration overhead, power breakdown across Mage components, and the scaling of performance and efficiency with input size.

Scalar Mage: Multi-Kernel Evaluation - *Performance*

This section evaluates performance (speedup and throughput) obtained by scalar Mage implementations on the seven SCoP kernels listed above. Each kernel can be mapped onto the considered scalar variants using different **Unroll Factors** (UFs), where UF denotes the degree of unrolling applied to the innermost loop to expose parallelism onto the CGRA. Therefore, the maximum UF is not an algorithmic property of the kernel alone, but depends on the resources and bandwidth provided by the selected Mage variant.

Table 4.4 reports, for each scalar Mage implementation, the UFs that can be applied to each workload. The table also includes the corresponding utilization of AGEs and PEs within the AGU and PEA, respectively. As expected, **PEA size is a**

Table 4.5 Maximum speedup obtained by Mage per kernel and unroll factor (UF). Execution cycles are also reported for both Mage and the CV32E40P CPU.

UF	Metric	firstdiff		gemm		hydro	
		Speedup	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**
1	CPU cycles Mage cycles	5.796×	6219 1073	8.534×	1007 118	8.558×	8900 1040
2	CPU cycles Mage cycles	**11.01**×	6219 565	15.14×	4815 318	**16.18**×	8900 550
3	CPU cycles Mage cycles	—	— —	**21.87**×	13754 629	—	— —

UF	Metric	atax		bicg		conv1d		conv2d	
		Speedup	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**	**Speedup**	**Cycles**
1	CPU cycles Mage cycles	5.751×	12445 2164	**11.50**×	12424 1080	9.311×	7616 818	11.03×	75658 6858
2	CPU cycles Mage cycles	**11.05**×	12443 1126	—	— —	13.36×	7616 570	16.45×	75658 4600
3	CPU cycles Mage cycles	—	— —	—	— —	**24.41**×	7616 312	**32.50**×	75658 2328

primary driver of UF scaling. Small arrays, such as *2x2_2x2_s*, only support low UFs, since the available number of PEs bounds the number of operations that can be executed per cycle. When keeping PEA size fixed, increasing the number of **AGEs and SpM banks can enable higher UFs by increasing memory bandwidth**, as observed when comparing *2x4_2x2_s* and *4x2_4x2_s*. Finally, **the access-execute channel granularity also affects mapping capabilities**. Wider channels increase the flexibility between AGEs, banks, and PEA rows, which can unlock higher UFs for some kernels, as in the comparison between *4x2_4x2_s* and *4x2_2x4_s*. For a given kernel and fixed UF, utilization generally decreases when scaling PEA and AGU resources, since a larger portion of the available hardware remains unused. On the other hand, **for a fixed implementation, increasing the UF tends to improve the exploitation of both PEA and AGU resources**. As a result, *smaller* variants are expected to be more area efficient at low UFs, whereas *larger* Mage implementations become advantageous when higher UFs are exploitable, since they combine higher performance with high PEA and AGU utilization, thus leading to good area efficiency, as will be demonstrated in section 4.4.4.

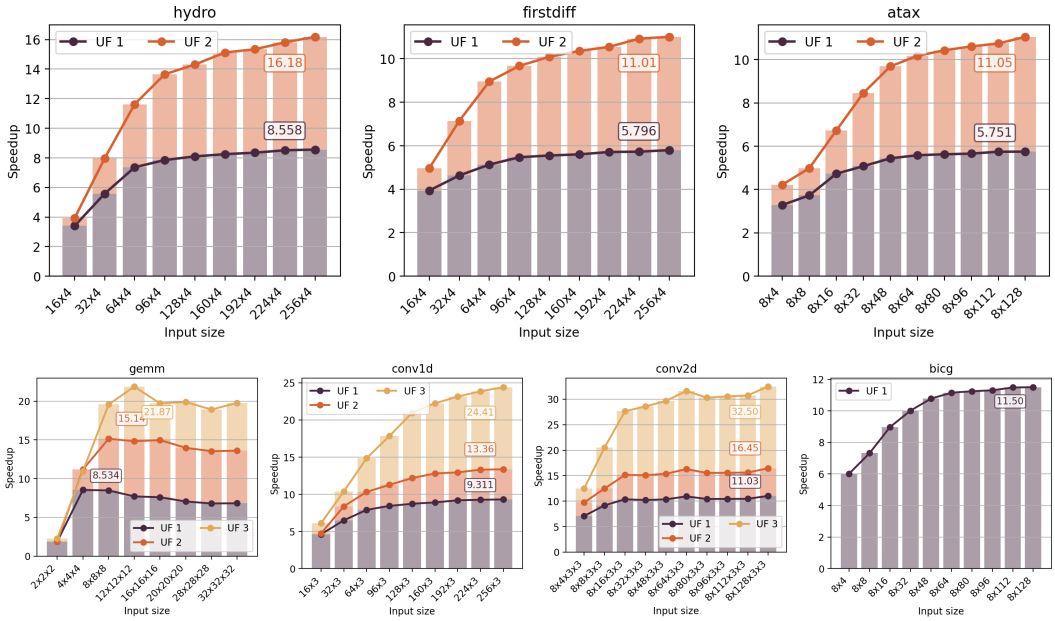


Fig. 4.16 Speedup scaling of scalar Mage across input sizes and unroll factors. X-axis labels report the main kernel dimensions: for *atax*, *bicg*, *firstdiff*, and *hydro* the label is $N \times M$, where N and M are the number of iterations for the two loops used to describe the kernel. For *conv1d*, it is $L \times K$, where L is the signal length and k the kernel size. For *conv2d* it is $H \times W \times K$, where H is the input matrix height, W the input matrix width, and K the filter height and width. For *gemm*, it is $N \times K \times M$, where N and M are the number of rows and columns of the output matrix, and K is the number of columns of the input matrix.

① **Speedup.** Table 4.5 summarizes the **maximum speedup achieved over the CV32E40P *scalar* baseline** for each kernel and supported UF, together with the corresponding cycle counts for both Mage and the CPU. In addition, Figure 4.16 complements the table by showing how speedup evolves along with input size for all kernels. It is important to note that Mage cycles account only for Mage run time and excludes configuration overhead, which is analyzed separately in section 4.4.4. Also, when considering end-to-end applications, configuration time can be often be "hidden" by either setting Mage configuration once at the beginning, or running it while other tasks are being performed, such as DMA transfers. Moreover, the tables reported hereinafter only show the UF and the related speedup for each kernel, as the same UF on different scalar Mage architectures yields the same speedup.

Overall, Mage achieves consistent speedups across the considered SCoP kernels, with peak values ranging from about 11.01× (*firstdiff*) up to 32.50× (*gemm*). The highest speedups are observed for *conv2d*, *conv1d*, and *gemm*, which reach **32.50**×, **24.41**×, and **21.87**×, respectively. These peaks are enabled by the fact that these

kernels can be mapped with UF3 on the larger Mage scalar variants, thus increasing the amount of useful computation performed per cycle. At UF2, *hydro* reaches a comparable speedup to the already mentioned kernels (**16.18**×), while *firstdiff* and *atax* remain lower. Two factors explain these differences. First, kernels with higher arithmetic intensity (e.g., *hydro*, *conv1d/2d*, *gemm*) typically achieve higher utilization of both the PEA and the AGU compared to kernels like *firstdiff*. Therefore, **speedup obtained by Mage compared to the CPU increases alongside the computational intensity of the performed workload**. Second, *atax* is the only kernel among the considered set that requires two successive Mage runs due to its data dependencies, which reduces the achievable speedup even when each run sustains $II = 1$. This effect can be seen by comparing *atax* at UF2 with *bicg* at UF1. Despite comparable computational structure, *bicg* reaches a similar speedup although being executed at a lower UF, since it requires only a single Mage run. Therefore, **speedup obtained by Mage lowers when multiple runs and re-configuration are needed for a workload**. As expected, for all workloads, increasing the UF causes speedup to substantially increase.

The scaling trends in Figure 4.16 show two recurring behaviors. First, **speedup grows quickly at small input sizes and then approaches a plateau as size increases**. Two reasons motivate this behavior. The first lies in the different scaling of the CPU baseline, as CV32E40P cycles tend to grow faster in the smallest instances and then scale more regularly, at slower rate, at larger sizes. In addition, fixed run-time effects inside Mage, such as pipelines fill and flush, have a higher relative weight for small problems and further reduce the initial speedup. As input size increases, Mage cycles get closer to the ideal steady-state trend ($\frac{N_{\text{iter}}}{UF}$), and speedup converges to its asymptotic value. Second, **increasing UF improves speedup whenever the selected Mage variant can sustain the extra parallelism and the required SpM bandwidth**. For *atax*, *firstdiff*, and *hydro*, moving from UF1 to UF2 yields an almost 2× improvement, while for *conv1d* and *conv2d* the availability of UF3 provides an even larger gain. In the convolution kernels case, it is important to point out that the gap between UF1 and UF2 is much smaller than the one between UF2 and UF3. This is mainly because UF2 requires zero-padding to support a filter size of 3, effectively leading to an execution equivalent to using a filter of size 4.

At small sizes, the separation between UF curves is smaller, again because fixed run-time effects represent a larger fraction of Mage cycles. A kernel-specific behavior is observed for *gemm*, where speedup slightly decreases at larger sizes,

Table 4.6 Maximum MOPS obtained by Mage @250MHz per kernel and unroll factor (UF). Mage cycles and the number of operations at the maximum MOPS point are also reported.

UF	Metric	firstdiff		gemm		hydro	
		MOPS	**Cycles/Ops**	**MOPS**	**Cycles/Ops**	**MOPS**	**Cycles/Ops**
1	Mage cycles Operations	238.6	1073 1024	998.2	32826 131072	1231	1040 5120
2	Mage cycles Operations	**453.1**	565 1024	1992	16450 131072	**2327**	550 5120
3	Mage cycles Operations	—	— —	**2894**	11323 131072	—	— —

UF	Metric	atax		bicg		conv1d		conv2d	
		MOPS	**Cycles/Ops**	**MOPS**	**Cycles/Ops**	**MOPS**	**Cycles/Ops**	**MOPS**	**Cycles/Ops**
1	Mage cycles Operations	946.4	2164 8192	**948.1**	1080 4096	463.9	818 1518	410.1	6858 11250
2	Mage cycles Operations	**1819**	1126 8192	—	— —	665.8	570 1518	611.4	4600 11250
3	Mage cycles Operations	—	— —	—	— —	**1216**	312 1518	**1208**	2328 11250

especially for UF1 and UF2. This is consistent with a more favorable scaling of the CPU baseline in that region, combined with size-dependent overheads in Mage due to **zero-padding** needed to match the mapped schedule, an effect that is even more visible at UF3. In addition, the regular structure of *gemm* allows the compiler to more effectively exploit *XCVPULP* features such as hardware loops, improving the CPU baseline and reducing the relative speedup at larger sizes.

② **Throughput.** Table 4.6 reports the *maximum throughput* achieved by Mage at 250 MHz in terms of MOPS (Mega Operations Per Second), for each kernel and supported UF. The table also reports the Mage cycle count and the number of operations evaluated at the maximum-MOPS point. MOPS is computed as:

$$\text{MOPS} = \frac{N_{\text{ops}}}{Cycles_{\text{Mage}}/f} \cdot 10^{-6} \quad \text{with} \quad f = 250\,\text{MHz}.$$

where N_{ops} is the number of arithmetic operations in each kernel, to ensure architecture-independent comparisons with other state-of-the-art CGRAs. In addition, Figure 4.17 shows how MOPS evolves with input size and, similarly to speedup, confirms that peak values are reached once execution is dominated by the steady-

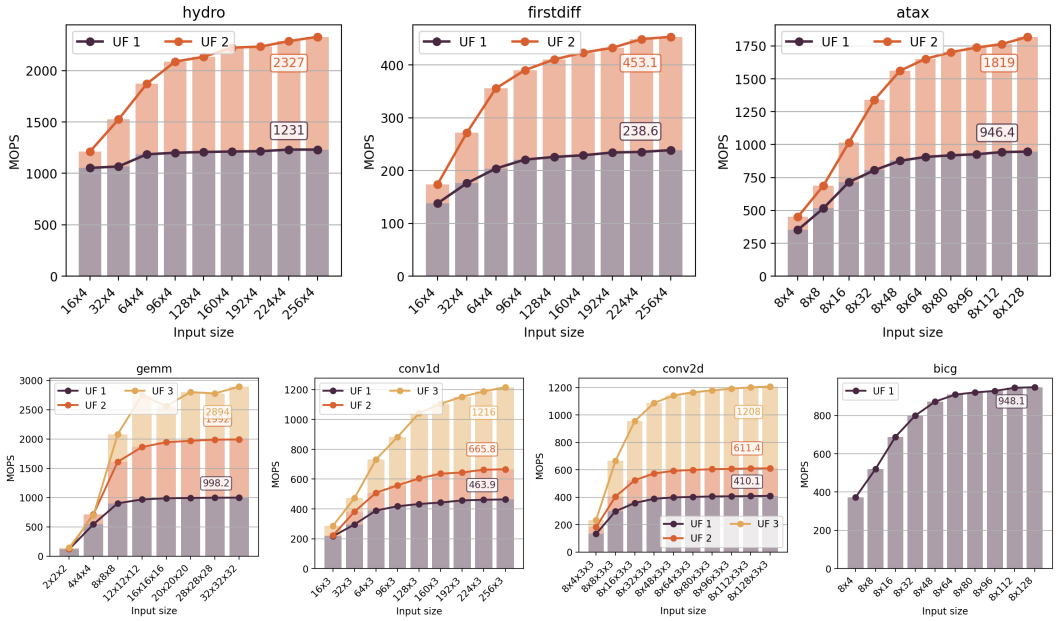


Fig. 4.17 MOPS scaling of scalar Mage across input sizes and unroll factors. X-axis labels report the main kernel dimensions, explained in Figure 4.17.

state region. While speedup provides a direct comparison against the CPU baseline, MOPS offers a standalone throughput metric that is also used for the State-of-the-Art comparison in subsection 4.4.6.

Across the benchmark set, peak throughput spans from a few hundred MOPS up to about 2.9 GOPS. The highest value is achieved by *gemm*, which reaches **2894** MOPS at UF3, followed by *hydro* at **2327** MOPS with UF2. A relevant point in interpreting these results is that MOPS depends on both the achieved cycle count and the number of counted operations. In particular, *gemm* and *hydro* have a high operation count per evaluated instance (N_{ops} in Table 4.6), which reflects their high arithmetic intensity. Therefore, this translates into higher MOPS than kernels such as *conv1d* and *conv2d*, where N_{ops} is smaller for the evaluated instances. The MOPS scaling curves mirror the speedup trends. **MOPS increases with input size** and then saturates as fixed run-time effects are amortized, and **increasing UF increases MOPS** whenever the mapping can sustain the additional parallelism and memory bandwidth (e.g., *hydro* 1231→2327 MOPS, *atax* 946.4→1819 MOPS, and *conv1d*/*conv2d* reaching about 1.2 GOPS at UF3). As for *gemm*, the UF3 curve is less monotonic, consistently with the speedup discussion, but it still reaches the largest peak throughput.

Table 4.7 Maximum area efficiency (MOPS/mm^2) for each kernels and UF on **scalar** Mage architectures.

Variant	atax		bicg	conv1d			conv2d			firstdiff		gemm			hydro	
	UF1	UF2	UF1	UF1	UF2	UF3	UF1	UF2	UF3	UF1	UF2	UF1	UF2	UF3	UF1	UF2
2x2_2x2_s	**4297**	—	**4304**	2106	—	—	1862	—	—	**1083**	—	4532	—	—	—	—
2x4_2x2_s	3732	—	3739	1829	—	—	1617	—	—	940.8	—	3936	—	—	**4853**	—
3x2_2x2_2x1_s	2919	—	2924	1431	2054	—	1265	1886	—	735.9	—	3079	**6144**	—	—	—
4x2_2x4_s	2186	4202	2190	1072	1538	—	947.4	1412	—	551.2	1047	2306	4602	—	—	—
4x2_4x2_s	2201	—	2205	1079	1548	—	953.6	1422	—	554.8	1054	2321	—	—	—	—
4x4_2x4_s	1894	3641	1898	928.7	1333	—	820.9	1224	—	477.6	907.0	1998	3988	5793	2464	4659
4x4_4x2_s	1906	—	1910	934.4	1341	—	826.0	1231	—	480.5	912.6	2010	4012	5829	2479	4687
5x2_2x4_2x1_s	1765	3393	1769	865.3	1242	**2269**	764.9	1140	**2253**	445.0	845.1	1862	3715	5398	—	—

Overall, the presented speedup and throughput results show that Mage benefits from unrolling when sufficient PEA, AGU and SpM resources are available, and that performance improves with input size until reaching a steady-state regime dominated by the $II = 1$ pipeline.

Scalar Mage: Multi-Kernel Evaluation - *Area Efficiency*

In Table 4.7, **peak area efficiency** is reported in MOPS/mm^2 across all scalar Mage variants for all kernels. It is computed employing the peak MOPS points for each workload along with the area of each Mage architecture. This metric captures how much throughput is delivered per unit area, hence it is useful to compare both different architectural sizes for the same kernel and UF, and the effect of enabling higher UF whenever the architecture can sustain them. Also, it is useful to compare Mage with other state-of-the-art CGRAs in subsection 4.4.6.

The first trend that can be analyzed is that, **for a fixed UF, area efficiency generally decreases as the architecture grows**. This is clearly visible for the UF1-only kernel *bicg*: *2x2_2x2_s* provides the highest values (4304 MOPS/mm^2), and larger variants progressively drop down to 1769 MOPS/mm^2 on *5x2_2x4_2x1_s*. The main reason is that, while larger architectures provide more resources and flexibility, the UF1 mappings do not exploit enough the additional parallelism to compensate for the increased area. Therefore, **for low-UF mappings, the smallest Mage implementation tends to be the most area efficient**.

A second trend is that **enabling higher UFs can mitigate, and in some cases overcome, the increased area of larger architectures**, and the most evident case is *gemm*. While UF1 yields 4532 MOPS/mm^2 on *2x2_2x2_s*, moving to UF2 on *3x2_2x2_2x1_s* raises area efficiency to **6144** MOPS/mm^2, and UF3 on the larger 4×4 variants still sustains very high values 5793-5829 MOPS/mm^2. In this case, the higher UF increases the compute density of the schedule. since *gemm* also has a large N_{ops}, the resulting MOPS increase is large enough to improve MOPS/mm^2 despite the increased area.

Also *conv1d* and *conv2d* show a similar behavior, with UF3 on *5x2_2x4_2x1_s* achieving better area efficiency (2269 and 2253 MOPS/mm^2 respectively) than UF1 on *2x2_2x2_s* (2106 and 1862 MOPS/mm^2 respectively) due to the fact that increased parallelism outweighs the larger CGRA area. However, even at UF3 these kernels remain less area efficient than *gemm* and *hydro*. The main motivation is that the evaluated convolution kernels expose a lower arithmetic intensity (lower operation count) than *gemm* and *hydro*. In other words, the compute density seen by the PEA is lower, so peak MOPS/mm^2 is in turn lower.

Finally, **access-execute channel partitioning has a negligible impact on area efficiency**, as expected from the area breakdown discussion. Architectures that differ mainly in channel width show nearly identical values (*4x2_2x4_s* vs. *4x2_4x2_s*, and *4x4_2x4_s* vs. *4x4_4x2_s*). At the same time, it is important to highlight that channel width enables higher UFs to be applied. Therefore, **scaling channel width allows higher UFs without incurring large area penalties in Mage crossbars implementation.** This also indicates that, in the explored range, area efficiency is dominated by SpM and PEA sizing and by the achievable UF, rather than by crossbar area.

Overall, Table 4.7 highlights two conclusions: ① **for low-UF mappings, minimizing area is key and small arrays tend to maximize MOPS/mm^2**, while ② **for compute-dense kernels where higher UF is feasible, unrolling can significantly increase area efficiency even on larger Mage variants**.

Scalar Mage: Multi-Kernel Evaluation - *Energy Efficiency*

In this section, Mage scalar variants' energy efficiency is evaluated through energy savings with respect to software execution on the CV32E40P CPU, as well as in

Table 4.8 Maximum energy savings obtained by **scalar** Mage per kernel and UF compared to the *scalar* CPU baseline.

Variant	atax		bicg	conv1d			conv2d			firstdiff		gemm			hydro	
	UF1	UF2	UF1	UF1	UF2	UF3	UF1	UF2	UF3	UF1	UF2	UF1	UF2	UF3	UF1	UF2
2x2_2x2_s	**9.2**	—	—	**13.6**	—	—	**12.5**	—	—	**9.0**	—	**10.3**	—	—	—	—
2x4_2x2_s	9.1	—	—	13.5	—	—	12.4	—	—	8.9	—	10.2	—	—	**15.4**	—
3x2_2x2_2x1_s	8.9	—	—	13.2	**13.2**	—	12.1	**10.7**	—	8.7	—	10.0	—	—	—	—
4x2_2x4_s	8.3	**12.6**	**9.7**	12.0	12.6	—	10.9	10.1	—	7.9	**12.4**	9.2	**9.7**	—	—	—
4x2_4x2_s	8.6	—	9.6	12.7	12.9	—	11.8	10.3	—	8.4	12.0	9.8	—	—	—	—
4x4_2x4_s	8.1	12.3	9.4	11.7	12.7	—	10.7	10.1	—	8.1	12.2	9.1	9.5	10.2	13.5	**20.0**
4x4_4x2_s	8.4	—	9.6	12.6	12.9	—	11.7	10.3	—	8.5	12.1	9.6	9.7	**10.4**	14.0	19.9
5x2_2x4_2x1_s	8.0	12.2	9.5	11.7	12.5	**22.0**	10.7	10.0	**14.3**	7.9	11.9	8.9	9.5	—	—	—

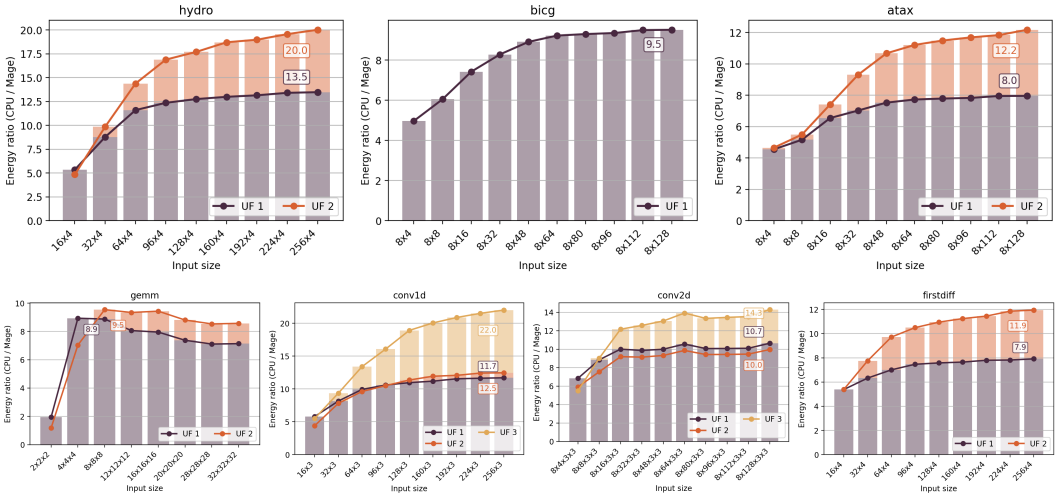


Fig. 4.18 Energy scaling of **scalar** Mage across input sizes and UF. X-axis labels report the main kernel dimensions, explained in Figure 4.16.

terms of MOPS/mW. Energy savings are reported as the energy ratio $\frac{E_{\text{CPU}}}{E_{\text{Mage}}}$, where E_{CPU} is the energy consumed by the CV32E40P baseline (including the system contributions described in subsection 4.4.2) and E_{Mage} is the energy consumed during Mage execution. Therefore, values larger than 1 indicate that Mage is more energy efficient than the CPU. In addition, energy efficiency is also discussed in terms of MOPS/mW, providing the possibility to compare Mage with other state-of-the-art CGRAs.

① **Energy Savings.** Table 4.8 summarizes the **maximum energy savings achieved on each kernel and UF across the considered scalar Mage implementations with respect to the *scalar* CPU baseline**. Also, Figure 4.18 illustrates how energy savings scale along increasing input sizes. However, plots are not all referring to the same architecture. As regards *atax*, *bicg*, *conv1d*, *conv2d*, *firstdiff*, and *gemm*, the scaling plots refer to energy savings achieved by *5x2_2x4_2x1_s*, while *hydro*

is reported on *4x4_2x4_s*. This choice is driven by the fact that *5x2_2x4_2x1_s* is the variant reaching maximum UF on nearly all kernels (apart from *gemm*), while since it does not support *hydro* due to PEA sizes, *4x4_2x4_s* is chosen to provide the scaling on that kernel. Across all kernels, **energy savings increase with input size and then saturate**. Therefore, this behavior matches the trend observed for speedup, even though, in this case, it is determined by both execution time and power consumption. Nevertheless, power presents negligible differences across different sizes of the same problem, both for Mage and for the CV32E40P scalar baseline. As a result, the overall trend remains very similar to the one of speedup. Hence, as the input size increases, fixed run-time effects become progressively negligible, and the energy ratio approaches an asymptotic value.

On *5x2_2x4_2x1_s*, the largest savings are obtained by the convolution kernels, especially on UF3. It is important to highlight that, for *conv1d*, energy savings slightly increase between UF1 and UF2, from about 11.7× at UF1 to 12.5× at UF2, while achieving a substantial increase towards UF3, where they reach up to **22.0**×. Therefore, the **zero-padding observed for speedup are reinforced when power is accounted for**, with only UF3 providing the additional parallelism needed to overcome the growth in power. This effect is even stronger on *conv2d*, where UF2 achieves lower savings than UF1, although the gap being small. Nevertheless, both convolution show a size-dependent saturation trend, which is visible for *bicg*, *atax*, *hydro* and *firstdiff*. In these workloads, moving from UF1 to UF2, whenever possible, consistently improves energy savings. In particular, significant savings are achieved on *hydro* (18.17 ×), confirming that **higher energy savings are achieved on arithmetic-intense workloads**. As for *gemm*, it shows a narrower range of savings, saturating around 8-10×, which is consistent with a stronger CPU baseline on this kernel. In addition, the same decreasing trend is observed for energy savings, as already discussed for speedup.

While increasing UF on a fixed variant generally leads to higher energy savings, the smallest variant consistently achieves the highest savings at a fixed UF, as shown in Table 4.8. Nonetheless, **for a given UF, the reduction in savings when scaling the architecture is limited and is offset by the higher savings enabled at larger UFs**. For instance, on *conv1d*, energy savings drop from 13.6× on *2x2_2x2_s* to 11.7× on *5x2_2x4_2x1_s* at UF1, but the latter variant enables 22.0× savings on UF3.

Table 4.9 Peak MOPS/mW achieved by **scalar** Mage architectures for kernels and related UFs.

Variant	atax		bicg	conv1d			conv2d			firstdiff		gemm			hydro	
	UF1	UF2	UF1	UF1	UF2	UF3	UF1	UF2	UF3	UF1	UF2	UF1	UF2	UF3	UF1	UF2
2x2_2x2_s	**162.1**	—	—	**80.0**	—	—	**53.6**	—	—	**41.6**	—	**122.8**	—	—	—	—
2x4_2x2_s	160.1	—	—	79.2	—	—	53.1	—	—	41.1	—	120.9	—	—	**242.3**	—
3x2_2x2_2x1_s	156.2	—	—	77.3	**77.7**	—	51.7	**46.0**	—	40.0	—	118.6	—	—	—	—
4x2_2x4_s	146.7	**222.4**	**83.2**	70.4	74.0	—	46.8	43.4	—	36.5	**57.3**	109.2	**130.2**	—	—	—
4x2_4x2_s	150.9	—	82.4	74.5	75.4	—	50.8	44.3	—	38.6	55.3	116.1	—	—	—	—
4x4_2x4_s	143.2	216.5	81.0	68.5	74.6	—	46.0	43.4	—	37.3	56.1	108.6	127.7	137.2	212.6	**315.8**
4x4_4x2_s	147.2	—	82.4	74.1	75.7	—	50.0	44.3	—	39.3	55.8	114.5	130.2	**139.8**	220.2	314.5
5x2_2x4_2x1_s	140.2	214.5	81.7	68.6	73.1	**129.1**	45.7	42.8	**61.3**	36.4	55.0	106.3	127.7	—	—	—

Overall, scalar Mage provides substantial energy savings over the CV32E40P *scalar* baseline across the benchmark set, up to 22×. **Higher UFs generally improve energy savings, but the plots also show that this is not guaranteed for every intermediate UF, especially if zero-padding is needed**. Depending on the kernel, a UF increase can raise average power enough to offset part of the cycle reduction, which explains the occasional crossings between curves. At the same time, **while smaller variants may achieve better savings on low UFs, larger variants obtain significantly higher savings enabling execution on higher UFs**.

① **Energy Efficiency.** In addition to energy saving against the *scalar* CPU baseline, energy efficiency is also evaluated through peak MOPS/mW, reported in Table 4.9 for scalar Mage architectures. This metric also allows for comparison of Mage variants against other state-of-the-art CGRAs.

Two trends can be identified. First, whenever higher UFs are supported, MOPS/mW generally increases, indicating that the **throughput gain from unrolling outweighs the corresponding increase in average power**. This can be observed across almost all kernels, with the convolution kernels again confirming the same behavior shown for energy savings. As a matter of fact, *conv2d* shows energy efficiency reduction from UF1 to UF2 (45.7 → 42.8 MOPS/mW), before reaching up to 61.3 MOPS/mW at UF3. Second, **for a fixed UF, energy efficiency slightly decreases on larger implementations**. For example, at UF1 *atax* spans from 162.1 to 140.2 MOPS/mW across increasingly larger variants. Although the switching resources remain the same and others are clock-gated, this behavior is expected as a slight increase in power is reasonable on larger variants, due to larger SpM, for example. Nonetheless, larger Mage variants allow higher efficiency by unlocking higher UFs, as discussed for energy savings. As a result, the **dominant knob for energy efficiency in the**

Table 4.10 Achievable UFs on Mage architectures for *matmul* and *dconv1d*, along with AGU and PEA utilization percentages.

Kernel	UF	2x2_2x2		2x4_2x2		3x2_2x2_2x1		4x2_4x2		4x2_2x4		4x4_4x2		4x4_2x4		5x2_2x4_2x1	
		AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA	AGU	PEA
matmul	1	75	50	75	25	50	33.3	37.5	25	37.5	25	37.5	12.5	37.5	12.5	30	20
	2	-	-	-	-	83.3	66.7	62.5	50	62.5	50	62.5	25	62.5	25	50	40
	3	-	-	-	-	-	-	87.5	75	87.5	75	87.5	37.5	87.5	37.5	70	60
	4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	90	80
dconv1d	1	100	50	100	25	66.7	33.3	50	25	50	25	50	12.5	50	12.5	40	20
	2	-	-	-	-	-	-	75	50	75	50	75	25	75	25	60	40
	3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	80	80

explored space is the highest sustainable UF, which allows significant scaling of energy efficiency.

Furthermore, another important consideration can be made. Compute-dense kernels such as *hydro* and *gemm* reach higher MOPS/mW values compared to other workloads, such as convolutions, at the same UFs. This can be mainly explained by the amount of arithmetic operations performed on the PEA relative to the SpM access bandwidth. For example, *gemm* and the convolution kernels exhibit similar SpM access bandwidth across UFs, but *gemm* performs a larger number of operations on the PEA, thereby achieving better energy efficiency in terms of MOPS/mW. In *hydro*, the number of operations is even higher, while the access bandwidth is lower, leading to an additional improvement in efficiency. Therefore, since SpM activity dominates power consumption, **kernels with a higher amount of arithmetic work per transferred operand can achieve more operations per watt**.

Scalar & Vector Mage: Matmul & Piecewise Conv1d

In the previous sections, results quantify scalar Mage variants on a broad SCoP workload set. On the other hand, the following discussion addresses two workloads that are central in many Transformer and Temporal Convolutional Network (TCN) pipelines, namely matrix multiplication (*matmul*) and dilated piecewise 1D convolution (*dconv1d*). The main aims of this section are three: ① compare scalar and vector Mage variants on widely-used compute-dense workloads regarding performance, power consumption, energy and area efficiency, ② evaluate performance and energy efficiency against CPU baselines that explicitly leverage the CV32E40P *XCVPULP* extensions, ③ quantify the impact of Mage configuration overhead. As for the other kernels, both *matmul* and *dconv1d* can be mapped with different UFs depending on

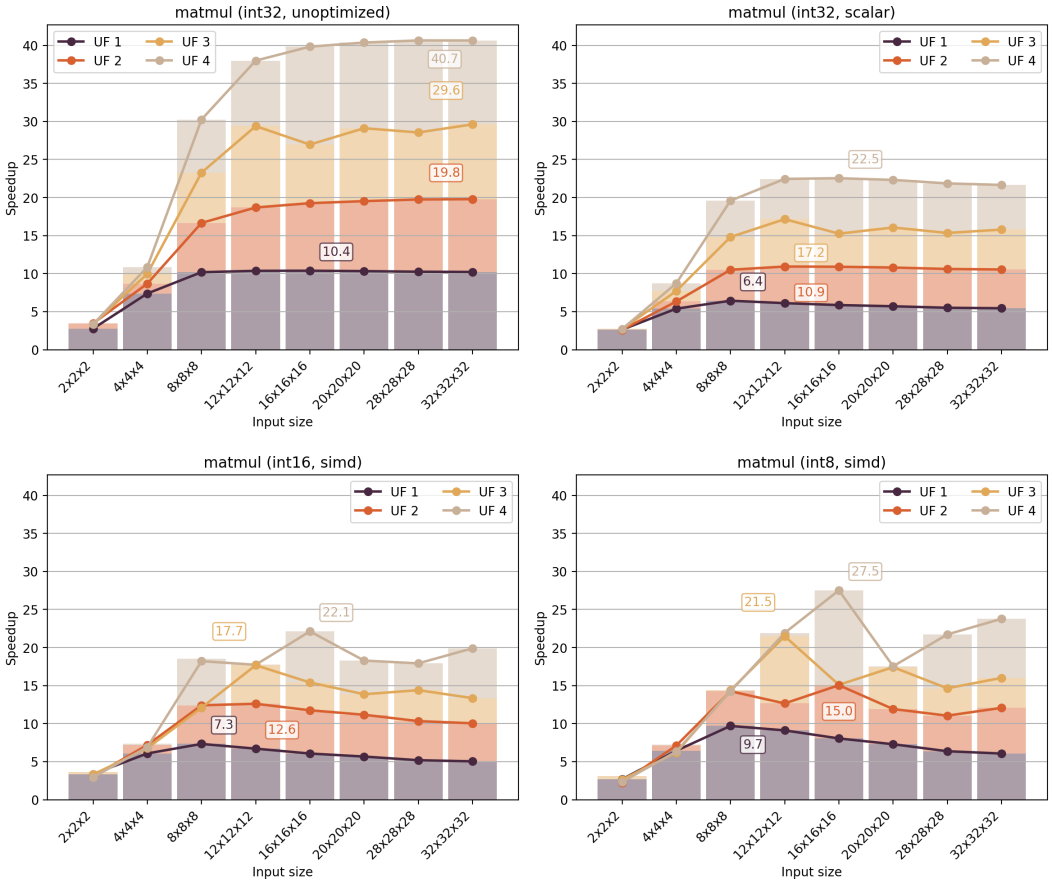


Fig. 4.19 Speedup scaling of scalar and vector Mage across input sizes and unroll factors for *matmul*. X-axis labels report the main kernel dimensions $N \times K \times M$, where N and M are the number of rows and columns of the output matrix, and K is the number of columns of the input matrix.

the available resources of the target Mage implementation, as reported in Table 4.10. For a given architecture, the same UF applies to both scalar and vector Mage variants, but vector variants can additionally exploit packed-SIMD execution. As illustrated by the table, larger Mage variants usually allow for higher UFs on both workloads. However, it is important to point out that the main driving factor for UF scaling is the SpM bandwidth and the number of PEA rows, as UFs increase along the sequence *2x2_2x2* $\rightarrow$ *3x2_2x2_2x1* $\rightarrow$ *4x2_4x2* $\rightarrow$ *5x2_2x4_2x1*. At the same time, PEA columns and access-execute channels scaling do not affect UF scaling for *matmul* and *dconv1d*.

Scalar & Vector Mage: Matmul & Piecewise Conv1d - *Performance*

① **Speedup.** Figure 4.19 reports speedup against three CPU baselines. The *unoptimized* baseline uses compiler-generated code for `int32` data without considering *XCVPULP* ISA extensions availability. The *scalar* baseline uses compiler-generated code for `int32` data exploiting the *XCVPULP* ISA. On the other hand, the *simd* baseline uses manually optimized kernels for `int16` and `int8` data that exploit *XCVPULP* instructions. As in previous sections, the reported speedups are not related to a specific scalar or vector Mage variant, as the speedup obtained by different variants on the same kernel, UF and data format is the same. Moreover, `int32` speedups apply to both scalar and vector Mage variants, while `int16` and `int8` results refer to vector variants only.

As it can be seen in Figure 4.19 and Figure 4.20, both scalar and vector Mage variants achieve substantial speedups against the *unoptimized* baseline, up to **40.65**× at UF4 for *matmul* and **43.89**× at UF4 for *dconv1d*. Nonetheless, this advantage is reduced when comparing against the *scalar* `int32` baseline, dropping to **22.54**× at UF4 for *matmul* and **29.21**× at UF3 for *dconv1d*. This reduction is expected, as even without format packing, *XCVPULP* improves CPU throughput through micro-architectural features such as hardware loops and other ISA-level optimizations, which the compiler can exploit more effectively in the *scalar* setup. As regards *matmul*, for both baselines, the speedup grows along with the input size and rapidly saturates, also showing a small decaying trend with the *scalar* baseline, although not significant. As for *dconv1d*, the growing trend is not present as the saturation region is already reached at the first point ($8 \times 8 \times 20 \times 3$). As regards `int16` and `int8`, compiler strategies are enhanced with manual optimization of the kernel. Nonetheless, vector Mage variants still achieve significant speedups in *matmul*, up to **22.12**× at UF4 for `int16` and up to **27.53**× at UF4 for `int8`. On the other hand, for *dconv1d*, peak speedup substantially increases when considering `int16` and `int8` formats, going up to **36.78**× at UF3 for `int16` and up to **53.51**× at UF3 for `int8`. These gains confirm that, even against an explicitly SIMD-optimized baseline, a fully pipelined execution with dedicated access-execute paths can provide large performance advantages.

Moreover, *dconv1d* tends to achieve higher speedups than *matmul* at comparable UFs and formats. A possible explanation is that manual SIMD optimization on CV32E40P is more effective for *matmul* than for *dconv1d*, since matrix multiplication

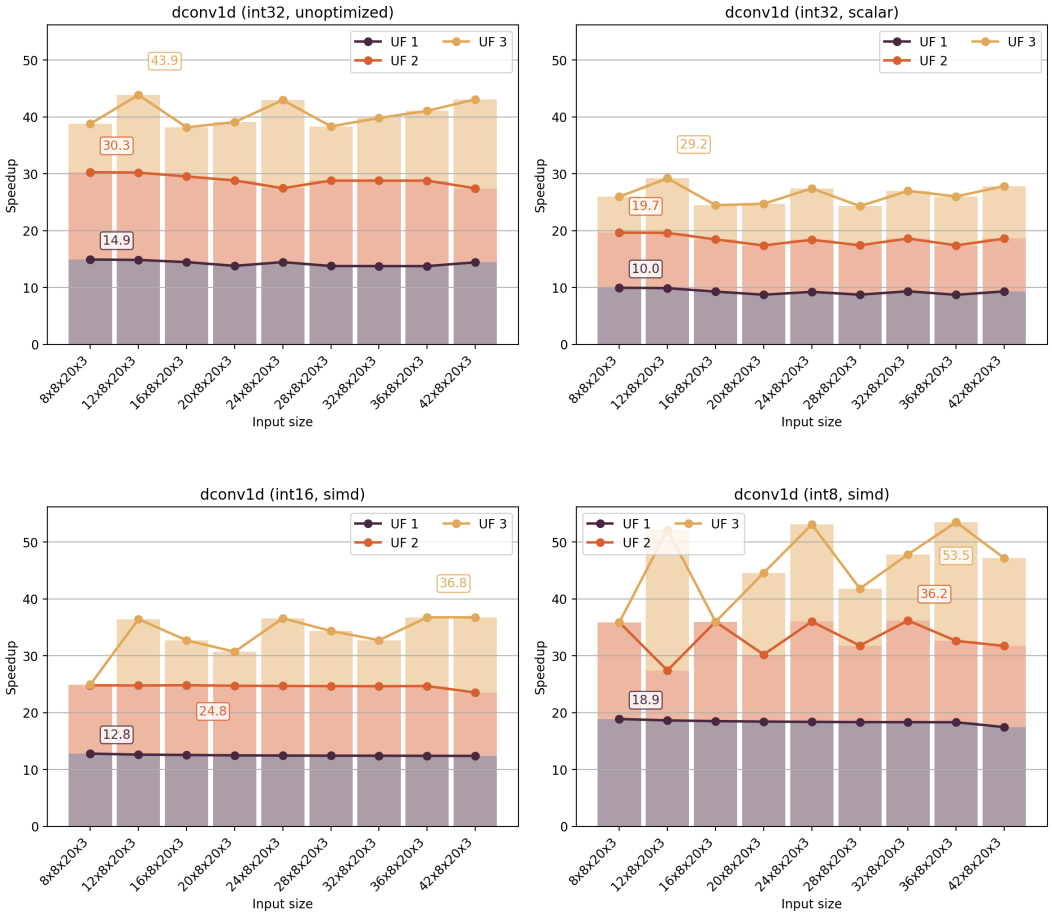


Fig. 4.20 Speedup scaling of scalar and vector Mage across input sizes and unroll factors for *dconv1d*. X-axis labels report the main kernel dimensions $Ch_{out} \times Ch_{in} \times T \times K$, where Ch_{out} and Ch_{out} are the number input and output channels, respectively, T is the length of the time sequence and K is the 1D filter size.

exposes more regular and well-studied SIMD opportunities, while *dconv1d* includes additional control and indexing structure that can reduce CPU's peak efficiency, even though SIMD operations are manually applied.

In reduced formats, beyond the size-dependent saturation already discussed, an additional effect becomes visible: **zero-padding**. Packed-SIMD execution requires packing multiple operands into a 32-bit word (2 lanes for `int16` and 4 lanes for `int8`). At the same time, mapping kernels with higher UFs on Mage requires distributing data across multiple SpM banks. As a consequence, the mapping imposes a *format- and UF-dependent tile shape* on the data layout in the SpM. When an input dimension is not a multiple of this tile shape, the remaining locations must

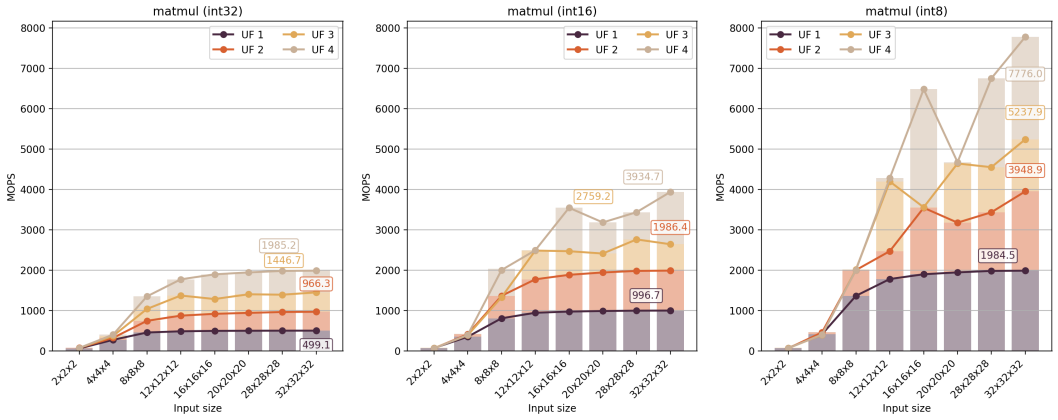


Fig. 4.21 MOPS scaling of scalar Mage across input sizes and unroll factors for *matmul*. X-axis labels report the main kernel dimensions, explained in Figure 4.19.

be padded with zeros to preserve correctness and keep the $II = 1$ schedule. This introduces extra useless computation and increases Mage run cycles. As a result, zero-padding impacts the results in two ways. First, it increases run cycles beyond what would be expected from the original problem size, which can slightly reduce both speedup and MOPS at specific sizes. Second, the effect is more pronounced at higher UFs and in packed-SIMD modes, because the mapping granularity becomes coarser and more constrained by lane packing. Therefore, padding is a primary cause of the non-monotonic points observed in the curves for both *matmul* and *dconv1d*. A concrete example is the local minimum at the $20 \times 20 \times 20$ point for *matmul* at UF4 in `int16` and `int8`. In the `int16` case, the effective tile width per cycle is $UF \cdot W_{\text{lane}} = 4 \cdot 2 = 8$ elements, which corresponds to the number of inputs fed to the PEA per cycle. Since $20 \bmod 8 = 4$, only 4 valid elements fit in the last tile and the remaining 4 must be padded, increasing the number of executed cycles and lowering throughput compared to the $16 \times 16 \times 16$ case, where $16 \bmod 8 = 0$ and no padding is needed. The same mechanism applies to $28 \times 28 \times 28$, where $28 \bmod 8 = 4$ again leads to a partially filled tile and additional padded work. Moreover, this effect is highly present in *dconv1d*, and, as expected, it becomes more dominant when considering reduced formats.

② **Throughput.** To provide a baseline-independent view of performance, throughput is also analyzed in terms of MOPS. Therefore, Figure 4.21 and Figure 4.22 report MOPS scaling along increasing input sizes, where Mage execution time accounts only for pure Mage execution cycles. As regards charts, the `int32`

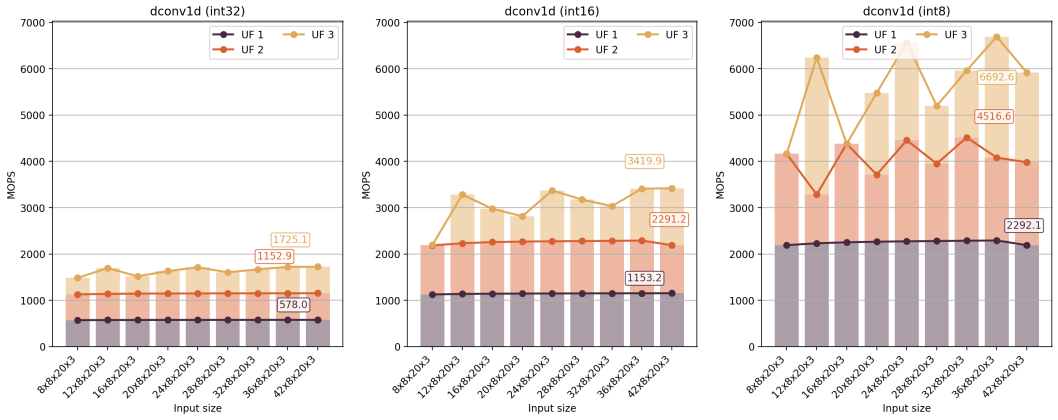


Fig. 4.22 MOPS scaling of scalar Mage across input sizes and unroll factors for *dconv1d*. X-axis labels report the main kernel dimensions, explained in Figure 4.20.

refer to the throughput achieved on both scalar and vector variants, while the `int16` `int8` ones refer to vector implementations only. As expected, **throughput increases along with UF and with the packed-SIMD width**. For `int32`, peak throughput reaches about **1985** MOPS at UF4 for *matmul*, and **1725** MOPS at UF2 for *dconv1d*. Exploiting packed-SIMD execution on vector Mage variants, throughput increases substantially, with `int16` reaching about **3935** MOPS and `int8` reaching about **7776** MOPS at UF4 for *matmul*, and **3420** MOPS in `int16` and **6693** MOPS in `int8` at UF3 for *dconv1d*. The ratios between the formats match the SIMD width (two 16-bit lanes and four 8-bit lanes), indicating that the **vector datapath translates lane-level parallelism into higher sustained throughput when access bandwidth is sufficient**. As for speedup, *matmul* throughput increases with input size and then saturates, while for *dconv1d* the saturation region is already reached at the first point. The remaining non-monotonic points, especially at high UF in reduced formats, are consistent with the padding effect discussed above.

③ **Configuration Overhead.** The results above focus on run time only, since configuration overhead can often be amortized by reusing a configuration across multiple executions or by hiding part of it behind other tasks, such as DMA transfers. Nevertheless, to provide a thorough analysis of Mage, Figure 4.23 quantifies the impact of configuration cycles on run cycles for *matmul* across UFs. While run cycles scale with matrix size, configuration cycles are approximately size-independent. Therefore, **configuration dominates small instances and becomes less relevant as size grows**. Moreover, **the configuration fraction increases with UF**, because

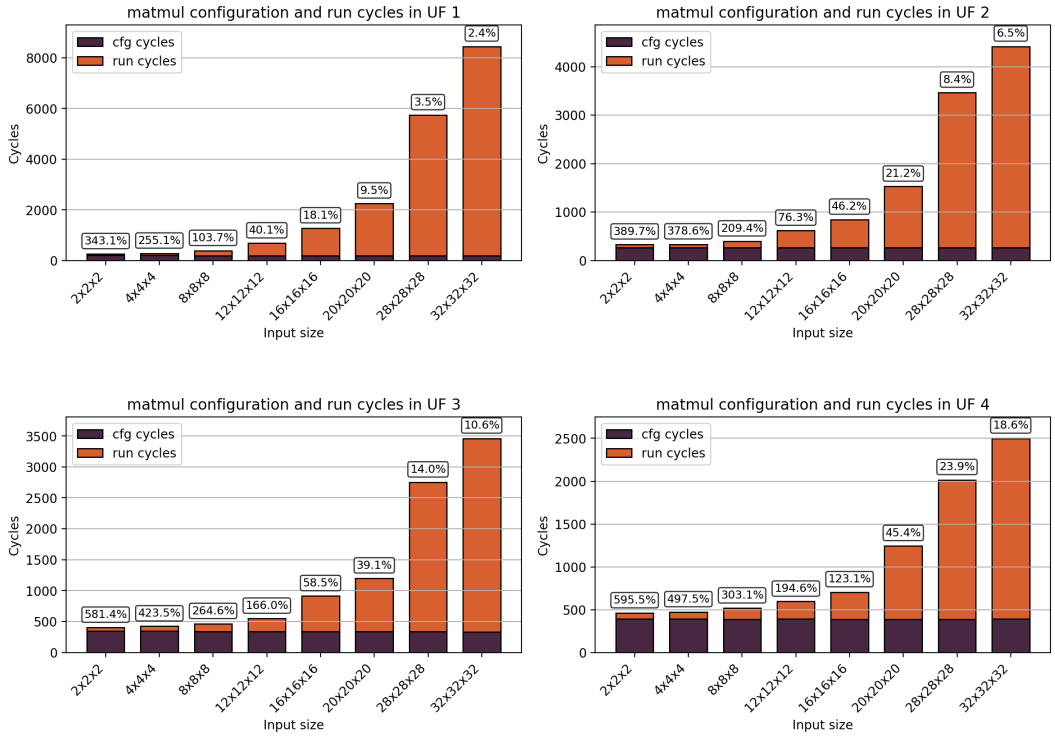


Fig. 4.23 Execution and configuration cycles for *matmul* across various UFs in `int8`

higher UFs require programming a larger portion of the AGU and the PEA, while also reducing run cycles. For example, at UF1, configuration decreases from about 343% of run cycles at the smallest size, to about 2.4% at the largest size. On the other hand, at UF4, it decreases from about 596% to about 18.6%. This highlights the following consideration: **higher UF maximizes steady-state throughput, but the benefit is fully realized only when run time is long enough to amortize configuration**. In realistic usage, **this motivates configuration reuse across repeated Mage execution**. Finally, the reported configuration times are obtained with a flexible setup that supports multiple matrix sizes and padding strategies. Therefore, if an application targets fixed shapes, this overhead can be reduced with specialized configuration functions.

Scalar & Vector Mage: Matmul & Piecewise Conv1d - *Area Efficiency*

This subsection evaluates area efficiency for *matmul* and *dconv1d* across scalar and vector Mage variants. Area efficiency is reported as MOPS/mm^2, combining the

Table 4.11 Peak MOPS/mm^2 for *matmul* and *dconv1d* on **scalar** Mage architectures.

Variant	matmul				dconv1d		
	UF1	**UF2**	**UF3**	**UF4**	**UF1**	**UF2**	**UF3**
2x2_2x2_s	**2266**	—	—	—	**2624**	—	—
2x4_2x2_s	1968	—	—	—	2279	—	—
3x2_2x2_2x1_s	1540	**2981**	—	—	1783	—	—
4x2_2x4_s	1153	2232	3342	—	1335	2664	—
4x2_4x2_s	1161	2247	3364	—	1344	**2681**	—
4x4_2x4_s	999	1934	2896	—	1157	2308	—
4x4_4x2_s	1005	1946	2914	—	1164	2322	—
5x2_2x4_2x1_s	931	1802	**2698**	**3703**	1078	2151	**3219**

peak throughput points with the area of the corresponding architecture. For scalar variants, results are reported in Table 4.11, while for vector ones they are reported in Table 4.12.

① **Scalar variants.** In Table 4.11, a trend can be identified for both kernels, and it is consistent with the one analyzed for other workloads. **For a fixed UF, area efficiency decreases as the architecture grows**, because additional area is not fully exploited unless the kernel can be mapped at a higher UF. For example, in *matmul* at UF1, area efficiency drops from 2266 MOPS/mm^2 on *2x2_2x2_s* to 931 MOPS/mm^2 on *5x2_2x4_2x1_s*. **When higher UFs become available, they compensate the larger area by increasing throughput**. For *matmul*, the peak rises with UF on every architecture that supports it, reaching **3703** MOPS/mm^2 at UF4 on *5x2_2x4_2x1_s*. For *dconv1d*, the same behavior is observed, with the peak efficiency reached on *5x2_2x4_2x1_s* with **3219** MOPS/mm^2 at UF3. Finally, architectures differing only in access-execute channel partitioning show nearly identical values, confirming that channel width has negligible impact on area efficiency compared to the dominant SpM and PEA sizing.

② **Vector variants & packed-SIMD scaling.** Table 4.12 shows that **vector variants significantly improve area efficiency for reduced formats by translating packed-SIMD width into higher MOPS at a moderate area cost**. For *matmul*, `int32` execution's area efficiency is comparable, as expected, to scalar and follows the same scaling with UF and architecture size. The main advantage appears for `int16` and `int8`. For instance, on *5x2_2x4_2x1_v*, `int16` execution reaches **6997** MOPS/mm^2 at UF4, while `int8` reaches **13828** MOPS/mm^2 at UF4. These gains

Table 4.12 Peak MOPS/mm^2 for *matmul* and *dconv1d* on **vector** Mage architectures.

matmul

Variant	int32				int16				int8			
	UF1	**UF2**	**UF3**	**UF4**	**UF1**	**UF2**	**UF3**	**UF4**	**UF1**	**UF2**	**UF3**	**UF4**
2x2_2x2_v	**2166**	—	—	—	**4327**	—	—	—	**8615**	—	—	—
2x4_2x2_v	1829	—	—	—	3653	—	—	—	7272	—	—	—
3x2_2x2_2x1_v	1469	**2845**	—	—	2934	**5848**	—	—	5843	**11626**	—	—
4x2_2x4_v	1100	2130	3189	—	2197	4378	6081	—	4374	8704	11544	—
4x2_4x2_v	1107	2143	**3209**	—	2211	4406	**6120**	—	4401	8758	**11617**	—
4x4_2x4_v	925	1792	2683	—	1849	3684	5118	—	3681	7325	9715	—
4x4_4x2_v	930	1801	2697	—	1858	3703	5144	—	3699	7361	9764	—
5x2_2x4_2x1_v	887	1718	2573	**3530**	1773	3533	4907	**6997**	3529	7023	9315	**13828**

dconv1d

Variant	int32			int16			int8		
	UF1	**UF2**	**UF3**	**UF1**	**UF2**	**UF3**	**UF1**	**UF2**	**UF3**
2x2_2x2_v	**2509**	—	—	**5005**	—	—	**9940**	—	—
2x4_2x2_v	2118	—	—	4225	—	—	8391	—	—
3x2_2x2_2x1_v	1702	—	—	3394	—	—	6741	—	—
4x2_2x4_v	1274	2541	—	2541	5050	—	5047	9950	—
4x2_4x2_v	1282	**2557**	—	2557	**5082**	—	5079	**10013**	—
4x4_2x4_v	1072	2139	—	2138	4250	—	4247	8374	—
4x4_4x2_v	1077	2150	—	2149	4271	—	4269	8416	—
5x2_2x4_2x1_v	1028	2051	**3069**	2050	4075	**6090**	4072	8028	**11923**

are consistent with the packed-SIMD lane width (2× for `int16` and 4× for `int8`), and they indicate that the slight area increase of vector implementations is compensated by a significant throughput increase. The same pattern is visible on smaller architectures at low UFs. *2x2_2x2_v* already reaches 4327 MOPS/mm^2 for `int16` and 8615 MOPS/mm^2 for `int8` at UF1.

The *dconv1d* table shows a similar packed-SIMD scaling. For architectures that support UF2, `int16` roughly doubles the `int32` area efficiency, and `int8` provides close to a 4× increase (2557→10013 on *4x2_4x2_v*). On *5x2_2x4_2x1_v*, which supports UF3, the peak reaches **6090** MOPS/mm^2 for `int16` and **11923** MOPS/mm^2 for `int8`. As for *matmul*, the almost identical numbers between *4x2_2x4_v* and *4x2_4x2_v* again confirm that channel partitioning has limited influence on area efficiency.

Overall, these results show that area efficiency is primarily driven by two factors: **the maximum UF that can be sustained on a given architecture and the available**

Table 4.13 Maximum energy saving for **scalar** Mage architectures. Results are reported for *matmul* and *dconv1d*, compared to the *unoptimized* CPU baseline.

Variant	scalar matmul (int32)			
	UF1	UF2	UF3	UF4
2x2_2x2_s	**8.9**	—	—	—
2x4_2x2_s	8.8	—	—	—
3x2_2x2_2x1_s	8.7	**9.7**	—	—
4x2_2x4_s	7.8	9.2	**10.1**	—
4x2_4x2_s	8.4	9.4	10.1	—
4x4_2x4_s	7.6	9.1	10.0	—
4x4_4x2_s	8.3	9.3	10.0	—
5x2_2x4_2x1_s	7.6	9.1	9.9	**10.7**

Variant	scalar dconv1d (int32)		
	UF1	UF2	UF3
2x2_2x2_s	**11.8**	—	—
2x4_2x2_s	11.7	—	—
3x2_2x2_2x1_s	11.5	—	—
4x2_2x4_s	10.4	13.0	—
4x2_4x2_s	11.2	**13.2**	—
4x4_2x4_s	10.2	12.9	—
4x4_4x2_s	11.1	13.1	—
5x2_2x4_2x1_s	10.1	12.8	**13.4**

packed-SIMD width. In general, for both scalar and vector implementations, the smallest architecture that supports a target UF tends to provide the best MOPS/mm^2. At the same time, UF scaling provides better area efficiency, as the increased throughput compensates for the larger area. In addition, for vector variants, reduced formats provide the largest gains. In this cases, **packed-SIMD increases throughput significantly while the incremental area cost of vector PEs is moderate**, resulting in a substantial improvement in MOPS/mm^2 for `int16` and especially for `int8`.

Scalar & Vector Mage: Matmul & Piecewise Conv1d - *Energy Efficiency*

This section complements the performance analysis of *matmul* and *dconv1d* by focusing on energy-related metrics. Energy savings are reported as the ratio $\frac{E_{\text{CPU}}}{E_{\text{Mage}}}$, where E_{CPU} is the energy consumed by the CV32E40P baselines and E_{Mage} is the energy consumed by Mage under the measurement assumptions described in subsection 4.4.2. Scalar Mage results are compared against the *unoptimized* CPU baseline, while vector Mage results are compared against the *scalar* and *simd* baselines. In addition, energy efficiency is reported in terms of MOPS/mW, which provides a baseline-independent view and allows comparison with other state-of-the-art CGRAs.

① **Energy savings.** For *matmul*, peak energy savings across scalar architectures are summarized in Table 4.13. **Across all scalar implementations, the maximum energy saving increases with the highest supported UF**, reaching **10.7** on *5x2_2x4_2x1_s* at UF4. The same trend can be observed in Figure 4.24 for

Table 4.14 Maximum energy saving for **vector** Mage architectures. Results are reported for *matmul* and *dconv1d*, compared to the *scalar* and *simd* baselines for `int32`, `int16`, and `int8`.

Variant	matmul int32				matmul int16				matmul int8			
	UF1	UF2	UF3	UF4	UF1	UF2	UF3	UF4	UF1	UF2	UF3	UF4
2x2_2x2_v	**6.0**	—	—	—	**6.9**	—	—	—	**8.5**	—	—	—
2x4_2x2_v	6.0	—	—	—	6.9	—	—	—	8.4	—	—	—
3x2_2x2_2x1_v	5.9	**5.8**	—	—	6.7	**6.8**	—	—	8.2	**7.5**	—	—
4x2_2x4_v	5.3	5.5	**6.3**	—	6.1	6.4	6.6	—	7.5	7.2	**7.7**	—
4x2_4x2_v	5.7	5.6	6.3	—	6.6	6.6	**6.6**	—	8.0	7.4	7.7	—
4x4_2x4_v	5.2	5.4	6.2	—	5.9	6.3	6.5	—	7.3	7.1	7.6	—
4x4_4x2_v	5.6	5.6	6.2	—	6.4	6.5	6.6	—	7.9	7.3	7.7	—
5x2_2x4_2x1_v	5.1	5.3	6.2	**6.3**	5.9	6.3	6.5	**6.4**	7.2	7.1	7.6	**8.4**

Variant	dconv1d int32			dconv1d int16			dconv1d int8		
	UF1	UF2	UF3	UF1	UF2	UF3	UF1	UF2	UF3
2x2_2x2_v	**7.9**	—	—	**10.7**	—	—	**15.6**	—	—
2x4_2x2_v	7.9	—	—	10.5	—	—	15.4	—	—
3x2_2x2_2x1_v	7.8	—	—	10.5	—	—	15.3	—	—
4x2_2x4_v	7.0	8.4	—	9.4	11.3	—	13.8	16.5	—
4x2_4x2_v	7.6	**8.7**	—	10.1	**11.6**	—	14.8	**17.0**	—
4x4_2x4_v	6.9	8.3	—	9.2	11.1	—	13.6	16.4	—
4x4_4x2_v	7.4	8.6	—	10.0	11.4	—	14.5	16.7	—
5x2_2x4_2x1_v	6.8	8.3	**8.8**	9.1	11.0	**11.9**	13.4	16.3	**18.0**

5x2_2x4_2x1_s, where higher UFs allow for better savings. Therefore, higher UFs substantially reduce executions cycles and, in turn, this mitigates the corresponding increase in average power. In addition, as for speedup, energy savings grow quickly from small matrices, and reach a plateau at larger sizes, while local inflections are mainly caused by zero-padding effects. As already analyzed, the trend similar to speedup is mainly caused by the fact that power has negligible differences across different sizes for both Mage (at fixed UF) and the CPU baselines.

As for vector Mage variants, peak energy savings reported in Table 4.14 are computed against the *scalar* (for `int32`) and *simd* (for `int16` and `int8`) baselines, which are significantly stronger than the *unoptimized* one. Consequently, savings are smaller in absolute value but remain consistently above 1 for moderate and large sizes. On *5x2_2x4_2x1_v*, for example, *matmul* savings at UF4 increase from 6.3× in `int32` up to **8.4**× in `int8` at UF4. As shown in Figure 4.24, the **UF ordering**

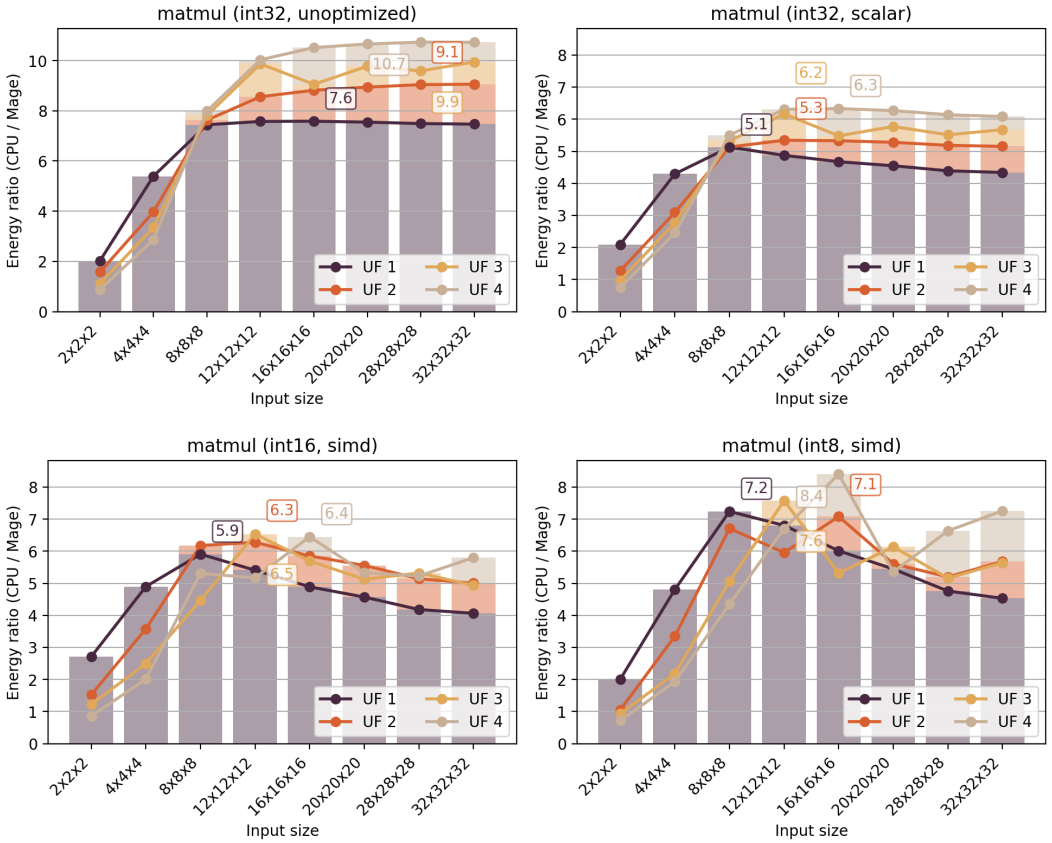


Fig. 4.24 Scaling of energy savings on *matmul* for scalar and vector Mage 5x2_2x4_2x1 variants across input sizes and unroll factors. The savings against the *unoptimized* baseline are achieved by the scalar variant, while the savings against the *scalar* and *simd* ones are obtained by the vector implementation.

is less monotonic for vector variants than scalar ones, with the curves showing stronger local variations. Besides the reduced energy gap against the *simd* baseline, this oscillating behavior is amplified by zero-padding. As observed for speedup, when approaching higher UFs and, more importantly, reduced data formats, tile granularity is coarser and extra work induced by padding becomes a significant contribution.

As regards *dconv1d*, peak energy savings for scalar Mage are summarized in Table 4.13. The best scalar result is achieved on *5x2_2x4_2x1_s* at UF3, reaching **13.4**×, and the same considerations made for *matmul* apply. However, compared to *matmul* scaling, Figure 4.25 shows a weaker dependence on input size, as savings already saturate at the smallest tested size, and remain in a relatively narrow band

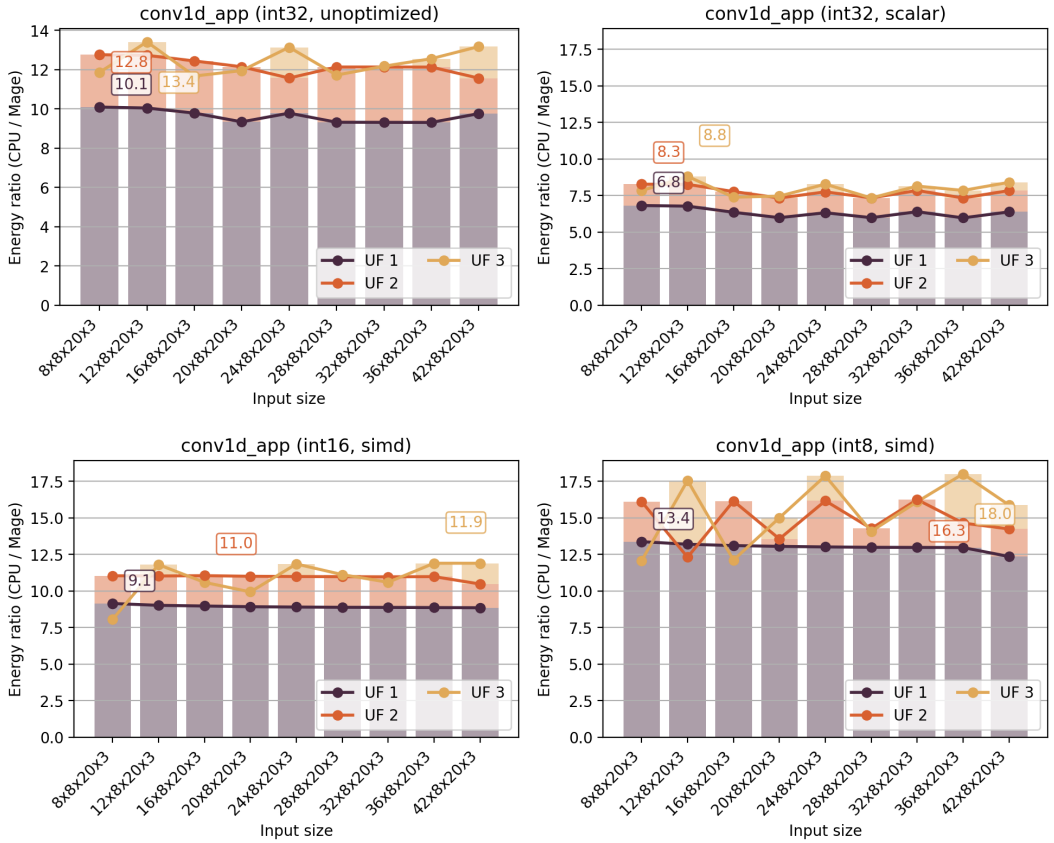


Fig. 4.25 Scaling of energy savings on *dconv1d* for scalar and vector Mage 5x2_2x4_2x1 variants across input sizes and unroll factors. The savings against the *unoptimized* baseline are achieved by the scalar variant, while the savings against the *scalar* and *simd* ones are obtained by the vector implementation.

across sizes, with several oscillations that can be attributed, also in this case, to zero-padding.

For vector Mage implementations, peak savings against the *simd* baseline are reported in Table 4.14. Despite the stronger baseline, energy savings remain substantial and consistently increase along with packed-SIMD width. On *5x2_2x4_2x1_v*, the best values are reached at UF3, with **8.88**× for `int32`, **11.9**× for `int16`, and **18.0**× for `int8`. Notably, against the *simd* baseline, *dconv1d* tends to achieve higher energy ratios than *matmul* at comparable UFs and formats. A possible explanation, again, is that manual optimization on CV32E40P is more effective for *matmul* than for *dconv1d*.

In addition, Table 4.15 reports the **power breakdown** for *5x2_2x4_2x1_v* across main components (SpM, PEA and AGU) for *dconv1d*, and it helps understand why

Table 4.15 Power breakdown of Mage for *dconv1d* on *5x2_2x4_2x1_v* across various UFs for `int32` and at maximum UF for `int16` and *int8*. Absolute power values are reported in mW.

Component		**int32 UF1**		**int32 UF2**		**int32 UF3**		**int16 UF3**		**int8 UF3**	
Main	**Sub**	**Abs [mW]**	**%**	**Abs [mW]**	**%**	**Abs [mW]**	**%**	**Abs [mW]**	**%**	**Abs [mW]**	**%**
SpM	$Bank_0$	2.26	27.07	2.23	16.28	2.20	11.46	2.17	11.79	2.06	11.64
	$Bank_1$	0.12	1.43	2.24	16.35	2.21	11.51	2.14	11.63	2.07	11.69
	$Bank_2$	9.11e-02	1.09	9.75e-02	0.71	2.19	11.41	2.14	11.63	1.98	11.19
	$Bank_3$	9.11e-02	1.09	8.92e-02	0.65	8.94e-02	0.47	8.72e-02	0.47	8.26e-02	0.47
	$Bank_4$	2.33e-02	0.28	2.33e-02	0.17	4.89e-02	0.25	5.05e-02	0.27	7.50e-02	0.42
	$Bank_5$	2.27	27.19	2.24	16.35	2.20	11.46	2.18	11.85	2.07	11.69
	$Bank_6$	9.53e-02	1.14	2.24	16.35	2.21	11.51	2.14	11.63	2.06	11.64
	$Bank_7$	9.11e-02	1.09	0.21	1.54	2.19	11.41	2.14	11.63	1.98	11.19
	$Bank_8$	9.11e-02	1.09	8.92e-02	0.65	8.94e-02	0.47	8.72e-02	0.47	8.26e-02	0.47
	$Bank_9$	2.33e-02	0.28	2.33e-02	0.17	0.27	1.41	0.38	2.05	0.69	3.92
	Total	5.16	61.74	9.48	69.22	13.70	71.34	13.51	73.43	13.15	74.32
PEA	PE_{00}	0.51	6.16	0.50	3.66	0.50	2.60	0.38	2.04	0.34	1.94
	PE_{01}	1.82e-03	0.02	2.04e-03	0.01	2.03e-03	0.01	1.96e-03	0.01	2.01e-03	0.01
	PE_{11}	1.79e-03	0.02	0.51	3.69	0.50	2.60	0.28	1.51	0.35	1.95
	PE_{21}	1.93e-03	0.02	2.04e-03	0.01	0.15	0.76	0.14	0.76	0.11	0.59
	PE_{31}	1.93e-03	0.02	1.90e-03	0.01	0.41	2.14	0.28	1.52	0.11	0.59
	PE_{41}	1.73e-03	0.02	1.73e-03	0.01	1.76e-03	0.01	1.75e-03	0.01	1.74e-03	0.01
	Total	0.52	6.27	1.02	7.41	1.56	8.12	1.08	5.85	0.90	5.10
AGU	AGE_{00}	0.14	1.65	0.14	1.00	0.14	0.71	0.14	0.73	0.13	0.73
	AGE_{01}	0.12	1.44	0.14	1.00	0.14	0.71	0.14	0.74	0.13	0.73
	AGE_{10}	1.73e-03	0.02	0.12	0.88	0.14	0.71	0.14	0.73	0.13	0.73
	AGE_{11}	1.73e-03	0.02	1.73e-03	0.01	1.73e-03	0.01	1.73e-03	0.01	1.73e-03	0.01
	AGE_{20}	1.73e-03	0.02	1.73e-03	0.01	0.12	0.62	0.12	0.65	0.12	0.68
	AGE_{30}	0.13	1.58	0.13	0.96	0.13	0.68	0.13	0.71	0.13	0.73
	AGE_{31}	0.12	1.44	0.13	0.96	0.13	0.68	0.13	0.71	0.13	0.73
	AGE_{40}	1.73e-03	0.02	0.12	0.88	0.13	0.68	0.13	0.71	0.13	0.73
	AGE_{41}	1.73e-03	0.02	1.73e-03	0.01	1.73e-03	0.01	1.73e-03	0.01	1.73e-03	0.01
	AGE_{50}	1.73e-03	0.02	1.73e-03	0.01	0.12	0.64	0.12	0.67	0.13	0.72
	Total	0.52	6.23	0.79	5.74	1.05	5.47	1.04	5.67	1.03	5.80
Overall	**Mage**	8.35	100.00	13.70	100.00	19.20	100.00	18.40	100.00	17.70	100.00

savings do not scale purely with the reduction in execution time. **Total power is dominated by the SpM across all UFs and formats**, accounting for 61.74% at `int32` UF1 and increasing to 69.22% and 71.34% at `int32` UF2 and UF3, respectively. This shift is explained by how unrolling affects the memory access footprint. At UF1, activity is concentrated on a small subset of banks, with $Bank_0$ and $Bank_5$ contributing 27% each, while other banks remain close to 1%. On the other hand, UF2 and UF3 spread accesses across more banks to sustain higher bandwidth at each cycle. As a result, **moving to higher UFs increases overall Mage power** (from 8.35 mW at `int32` UF1 to 19.2 mW at `int32` UF3), and most of this increase comes from the SpM rather than the access and execute paths. As a matter of fact, although the trend across UFs and data formats is consistent with the one observed for the SpM, the AGU and PEA contributions stay relatively stable in the range 5-6% and 5-8%, respectively.

Table 4.16 Peak MOPS/mW for **scalar** Mage architectures for *matmul* and *dconv1d*.

Variant	matmul (int32)			
	UF1	UF2	UF3	UF4
2x2_2x2_s	**65.6**	—	—	—
2x4_2x2_s	64.8	—	—	—
3x2_2x2_2x1_s	63.7	**72.1**	—	—
4x2_2x4_s	57.4	68.5	**75.3**	—
4x2_4x2_s	61.9	70.0	75.3	—
4x4_2x4_s	56.1	67.6	74.6	—
4x4_4x2_s	60.9	69.5	75.0	—
5x2_2x4_2x1_s	55.7	67.6	74.2	**80.0**

Variant	dconv1d (int32)		
	UF1	UF2	UF3
2x2_2x2_s	**75.8**	—	—
2x4_2x2_s	75.0	—	—
3x2_2x2_2x1_s	73.9	—	—
4x2_2x4_s	66.7	82.3	—
4x2_4x2_s	72.1	**83.5**	—
4x4_2x4_s	65.4	81.2	—
4x4_4x2_s	71.0	82.9	—
5x2_2x4_2x1_s	64.9	80.6	**87.6**

Furthermore, reduced formats at maximum UF (UF3) slightly *reduce* measured total Mage power compared to `int32` UF3 (18.4 mW for `int16` and 17.7 mW for `int8`), while keeping the SpM as the main contributor (73-74%). This suggests that the **higher packed-SIMD throughput is not contrasted by a proportional power increase**, mainly because the dominant SpM access pattern is unchanged, and the effective switching activity in datapaths is lower for reduced bit-width operands. This behavior is coherent with the higher energy savings observed for `int16` and `int8` when comparing vector Mage against the *simd* baseline.

② **Energy Efficiency.** While energy ratios quantify savings against the selected CPU baselines, MOPS/mW provides an independent metric that is also useful for comparisons with other State-of-the-Art CGRAs.

Results for scalar variants are summarized in Table 4.16, and they show similar trends to the ones already analyzed across implementaions and UFs. For *matmul*, energy efficiency increases with UF and peaks at **80.0** MOPS/mW on *5x2_2x4_2x1_s* at UF4, mirroring the fact that unrolling raises sustained throughput while average power grows more slowly. For *dconv1d*, the best scalar value is **87.6** MOPS/mW on *5x2_2x4_2x1_s* at UF3.

As for vector Mage, results are reported in Table 4.17 and show the expected increase with packed-SIMD width. For *matmul*, the peak reaches **76.1** MOPS/mW in `int32` at UF4, **156.8** MOPS/mW in `int16` at UF4, and **345.6** MOPS/mW in `int8` at UF4 (on *5x2_2x4_2x1*). For *dconv1d*, the best values are **83.0** MOPS/mW in `int32` at UF3, **172.7** MOPS/mW in `int16` at UF3, and **356.9** MOPS/mW in `int8` at UF3

Table 4.17 Peak MOPS/mW for **vector** Mage architectures for *matmul* and *dconv1d*.

Variant	matmul int32				matmul int16				matmul int8			
	UF1	UF2	UF3	UF4	UF1	UF2	UF3	UF4	UF1	UF2	UF3	UF4
2x2_2x2_v	**63.5**	—	—	—	**128.8**	—	—	—	**252.8**	—	—	—
2x4_2x2_v	62.9	—	—	—	127.3	—	—	—	249.6	—	—	—
3x2_2x2_2x1_v	61.9	**69.5**	—	—	125.2	**146.1**	—	—	245.6	**288.2**	—	—
4x2_2x4_v	55.8	65.7	**72.0**	—	113.1	137.9	140.8	—	222.7	276.1	**272.8**	—
4x2_4x2_v	60.2	68.0	72.0	—	121.8	141.9	**141.5**	—	238.8	282.1	272.8	—
4x4_2x4_v	54.7	64.8	70.9	—	110.5	136.1	139.4	—	217.4	272.3	270.0	—
4x4_4x2_v	59.1	67.1	71.3	—	119.5	139.9	140.1	—	234.0	278.1	271.4	—
5x2_2x4_2x1_v	54.2	64.4	70.9	**76.1**	109.8	135.1	139.4	**156.8**	215.7	270.5	268.6	**345.6**

Variant	dconv1d int32			dconv1d int16			dconv1d int8		
	UF1	UF2	UF3	UF1	UF2	UF3	UF1	UF2	UF3
2x2_2x2_v	**73.3**	—	—	**149.8**	—	—	**298.5**	—	—
2x4_2x2_v	72.6	—	—	148.2	—	—	296.1	—	—
3x2_2x2_2x1_v	71.9	—	—	146.9	—	—	292.7	—	—
4x2_2x4_v	64.9	78.4	—	132.4	162.5	—	264.1	324.9	—
4x2_4x2_v	69.9	**81.2**	—	142.5	**167.2**	—	283.0	**334.6**	—
4x4_2x4_v	63.7	77.9	—	130.0	160.2	—	260.5	322.6	—
4x4_4x2_v	68.6	80.1	—	140.3	164.8	—	278.8	329.7	—
5x2_2x4_2x1_v	63.0	77.4	**83.0**	128.6	159.1	**172.7**	256.4	320.3	**356.0**

(again on *5x2_2x4_2x1*). It is important to highlight that the scaling across data formats, at fixed UF and variant, is close or even superior to the SIMD lane count, which is coherent with the power breakdowns. Vector variants allow for higher throughput on reduced formats, closely following SIMD lane count scaling, and this is amplified, in energy efficiency, by the lower power consumption achieved on reduced data formats.

Overall, the combined view of $\frac{E_{\text{CPU}}}{E_{\text{Mage}}}$ and MOPS/mW confirms that higher UFs and packed-SIMD computation consistently improve energy efficiency on even larger Mage variants, with vector Mage implementation showing even lower power consumption on SIMD execution for `int16` and `int8`.

4.4.5 Accelerating Temporal Convolutional Networks with Mage

In order to evaluate Mage on an end-to-end application, it has been exploited to accelerate the inference of a **Temporal Convolutional Network** (TCN) model for

Algorithm 2 Dilated 1D Convolution (*dconv1d*) algorithm mapped onto Mage with Unroll Factor of 2 on the inner loop.

Require: S: scaling factor for quantization, i.e., the number of bits to shift right to approximate division by 2^S

for $o \leftarrow 0$ **to** $numChOut - 1$ **do**
  for $t \leftarrow 0$ **to** $timeLength - 1$ **do**
    $Out[o][t] \leftarrow 0$
    for $i \leftarrow 0$ **to** $\frac{numChIn}{2} - 1$ **do**
      for $k \leftarrow 0$ **to** $kernelSize - 1$ **do**
        $idx \leftarrow t + D \cdot k$
        $conv_1 \leftarrow ((In[i][idx] \times W[o][i][k]) \gg S)$
        $conv_2 \leftarrow ((In[i + \frac{numChIn}{2}][idx] \times W[o][i + \frac{numChIn}{2}][k]) \gg S)$
        $Out[o][t] \leftarrow Out[o][t] + conv_1 + conv_2$
      end for
    end for
    $Out[o][t] \leftarrow Out[o][t] + B[o]$
  end for
end for

time-series processing, following the hardware-software co-design method presented in [62]. Therefore, the main aim is to assess the benefits enabled by Mage in a realistic embedded inference pipeline, where the accelerator runs alongside CPU operators, and where data movement is managed through the system DMA.

Temporal Convolutional Networks are fully-convolutional models tailored for processing temporal sequences. Differently from Recurrent Neural Networks (RNNs), which rely on inherently sequential recurrences, TCNs use convolutional layers and expose substantially more parallelism. Two properties are particularly relevant for TCN-based inference. First, **causal convolutions** ensure that predictions at time step t depend only on current and past samples, making the model suitable for real-time execution. Second, long-range dependencies are captured through **dilated convolutions**, which expand the receptive field without requiring large filters or deep networks, but rather using *dilation*. A dilated convolution of an input sequence x with a filter f of size k and dilation rate d can be expressed as:

$$(y * f)(t) = \sum_{i=0}^{k-1} f(i) \cdot x\big(t - d \cdot i\big), \tag{4.3}$$

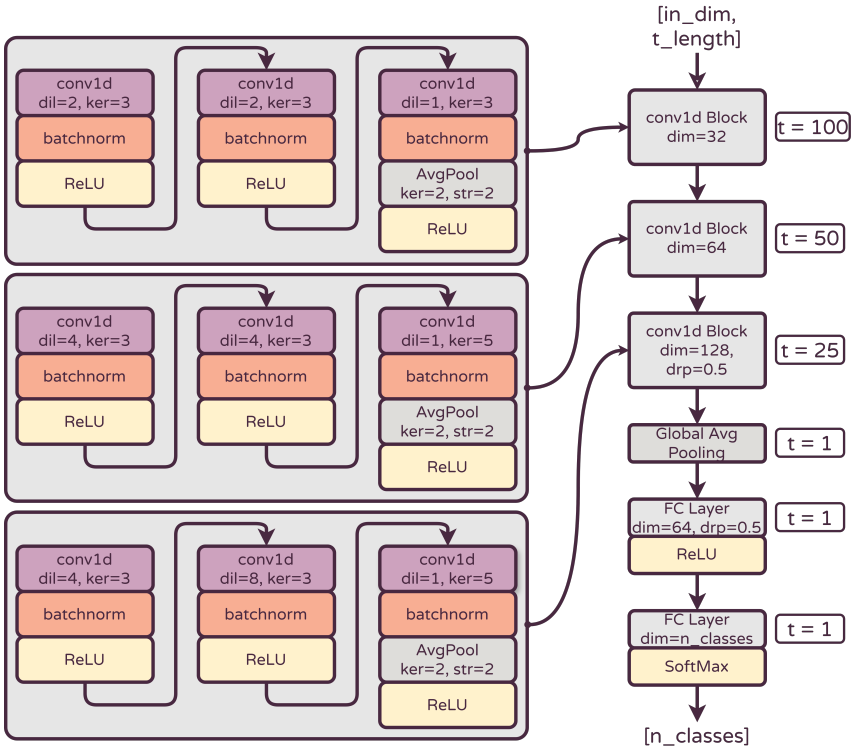


Fig. 4.26 Block diagram of the TCN pipeline considered for Mage-based acceleration.

where d determines the spacing between sampled input elements. By increasing d across layers (typically exponentially), the receptive field grows quickly while preserving compact kernels.

The specific TCN model considered for Mage-based acceleration in [62] is organized into hierarchical blocks, as shown in Figure 4.26. The TCN model mainly relies on *dilated causal 1D convolutions*, which correspond to the *dconv1d* workload analyzed in previous sections, in order to capture long-range temporal dependencies while keeping the filter size small. The pseudocode of *dconv1d* is provided in Algorithm 2 with UF2 on the inner loop, which is the version mapped onto the Mage variant used in this context. Within each block, one or more convolutional layers are present. Each layer encompasses a *dconv1d* operation, which is the compute-heavy part of the layer, followed by batch normalization and a *ReLU* or *Softmax* activation function. It is important to note that dilation rates progressively increase throughout convolutional layers, thus allowing to expand the receptive field without requiring an excessively deep network. In addition, pooling layers are selectively introduced between blocks to reduce the temporal resolution. At the end of the feature-extraction

pipeline described so far, global average pooling aggregates the temporal information into a compact representation, which is then processed by fully connected layers for classification.

It is important to point out that TCN models and, in particular, dilated convolutional layers can highly benefit from Mage acceleration, as the **flexibility of Mage's AGEs can be fully exploited to support both 3D addressing schemes needed by *dconv1d* as well as, more importantly, dilated accesses to temporal sequences**.

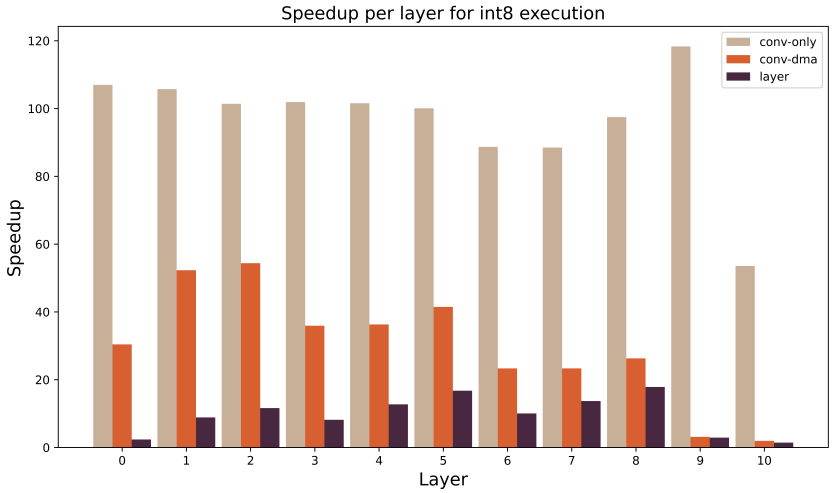
As regards the employed experimental setup, it includes both the hardware platform and the software stack used to run the TCN inference. As for the hardware platform, an X-HEEP-based microcontroller system has been designed and then deployed on a *Xilinx Pynq-Z2* FPGA board. The system mainly integrates the CV32E40P core (including a floating-point unit but without *XCVPULP* extensions), a multi-bank memory subsystem comprising 8 32-KiB banks, a DMA subsystem with a single channel, and an SPI module to interact with an external flash mainly exploited for storing weights. Mage is integrated as a loosely coupled accelerator. The instantiated Mage configuration targets vector UF2 *dconv1d* execution, whose pseudocode is provided in Algorithm 2, and it includes 8 AGEs and 8 internal SpM banks of 4 KiB each, while the execute side uses a 3×3 PEA (with 8 vector PEs and 1 vector accumulation PE) and a 4×2 access-execute channel partitioning. The same mapping strategy could also be realized on one of the variants discussed earlier that supports *dconv1d* on UF2 (e.g., *4x2_4x2_v* or *4x2_2x4_v*). The selected instance reflects the mapping choices adopted when the experiment was set up.

On the software side, two *C implementations have been developed to evaluate Mage benefits*. The first is a CPU-only baseline executing the full inference on the CV32E40P CPU without exploiting *XCVPULP* ISA extensions. The second version encompasses **acceleration of the dilated convolution layers and the fully-connected layers on Mage**, while keeping the remaining operators on the CPU. Both implementations rely on static memory allocation through a global buffer partitioning RAM into regions for parameters, inputs, and intermediate activations. Model parameters are stored in external flash and transferred through the DMA (via SPI) to main memory or, when profitable, directly to Mage internal SpM. Inference is mainly performed in fixed-point (`int8` and `int16`) for real-time operation, while selected stages between layers still execute on the CPU and may require floating-point support.

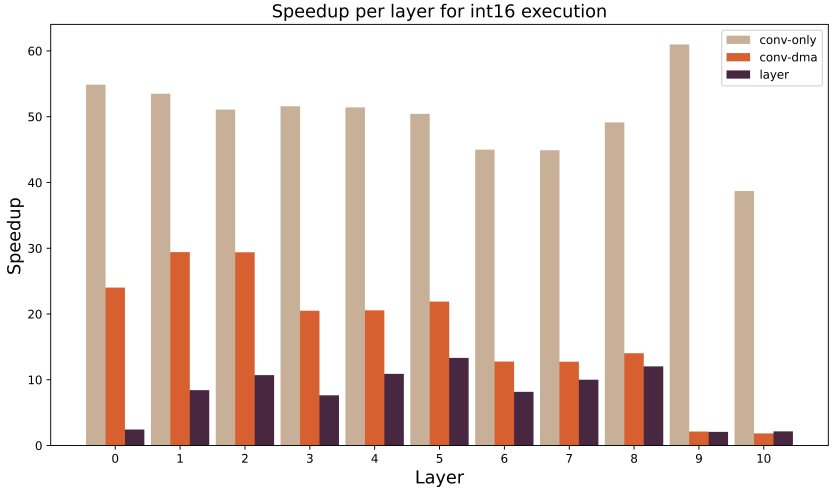
In the Mage-accelerated version, the C implementation also includes the **run-time Mage configuration** used to execute each dilated convolution, together with **tiling**. Since the full input/output tensors of a layer cannot fit into Mage internal memory, each layer is decomposed into uniformly sized tiles chosen according to the maximum number of input and output channels that can be stored. Tiles are processed using an output-stationary strategy: intermediate outputs remain in Mage until the tile completes, and only the final results are written back to system memory. At run time, for each tile, the CPU programs the DMA to move inputs and weights from system memory or flash into Mage SpM according to the required layout. Configuration overhead for the DMA is amortized by sharing the same configuration across multiple transfers. In addition, any needed **zero-padding** is handled directly by the DMA transfer, as X-HEEP's DMA provides inherent support for padding. Then, the CPU writes the configuration registers, starts Mage, and waits for completion. After execution, the produced output tile is written back to system memory via DMA and the procedure continues with the next tile. Since layers share the same structure and only differ in dimensions and dilation, Mage is only partially reconfigured between layers. Therefore, only a small set of configuration registers is updated (typically loop bounds and a subset of AGE strides), while the remaining configuration is reused. This keeps per-layer reconfiguration overhead small relative to tile run time, while still supporting layer-specific tiling and dilation parameters. In addition, packed-SIMD execution is used by selecting `int16` vector mode to process two packed `int8` values per word, while `int32` mode is used to process `int16` fixed-point data without packing.

Figure 4.27a and Figure 4.27b report per-layer speedups for the `int8` and `int16` configurations. For each layer, three speedup figures are shown: ① the speedup of the pure *dconv1d* kernel considering only the compute phase (i.e., excluding DMA transfers) (*conv-only*), ② the speedup of the overall convolution when DMA transfers are included (*conv-dma*), and ③ the overall layer speedup when also accounting for operations carried out on the CPU, outside of convolution (*layer*).

Three trends emerge consistently across layers and formats. First, the gap between *conv-only* and *conv-dma* speedups highlights the cost of data movement. If DMA transfers are considered, speedup drops, especially for smaller layers (9 and 10 in this case), where execution time spent for actual computation is far lower than the one spent transferring data. Therefore, this mainly explains the high gap between *conv-only* and *conv-dma* bars on the last two layers. In addition, it is useful



(a) Per-layer speedup for the TCN model executing on `int8` fixed-point data. Therefore, Mage implementation is carried out using `int16` vector mode, packing 2 8-bit data across the two `int16` lanes.



(b) Per-layer speedup for the TCN model executing on `int16` fixed-point data. Therefore, Mage implementation is carried out using `int32` operators, without data packing.

to highlight that this gap could be further reduced using several techniques, such as optimizing DMA-related software calls or, more importantly, exploiting multiple DMA channels for performing multiple concurrent transfers. Nonetheless, significant per-layer speedups are obtained also considering DMA transfers, ranging from 20× to 60× in the `int8` case, while ranging from 10× to 30× in the `int16` case. The only exceptions are represented by the last two layers, in which lower speedups are achieved due to their significantly smaller size compared to other layers.

Second, the *layer* speedup is substantially lower than the *conv-dma* one, mainly because a non-negligible portion of each layer is still executed on the CPU, including floating-point stages and other scalar operators. This is particularly evident in the last layers, where the convolutional workload is no longer dominant and the achievable end-to-end speedup is bounded by the CPU-side fraction of execution time.

The third observed behavior is the difference in speedup across layers with different filter size and dilation. As can be seen in the reported plot, layers with larger dilation and filter size usually achieve lower speedup, as zero-padding has a greater impact when scaling these two parameters. As a matter of fact, using larger filters and dilation requires $dilation \cdot (size_{filter} - 1)$ padding zeros to be added at the beginning and at the end of each temporal sequence, although intermediate zeros can be shared among sequences. Needed zeros also increase when data layout has to be properly adjusted for UF2 and reduced data format in `int8`. Since padding is not needed for CPU execution, the usage of zero-padding on Mage gives software execution an advantage.

Despite the system-level limits imposed by explicit data movement and by the remaining non-offloaded (and often floating-point) computations between layers, Mage enables considerable **12.9× and 9.7× speedups on the overall TCN application on `int8` and `int16` data formats, respectively**.

4.4.6 Comparison with State-of-the-Art CGRAs

In Table 4.19, a comparison of Mage against state-of-the-art CGRAs is provided regarding throughput, area, power and area and energy efficiency. In order to have a fair comparison, whenever possible, performance and efficiency metrics are extracted from the *matmul* implementations across related papers, although they may target different matrix sizes and different data widths. When this is not possible, average values are reported. As regards Mage, results are reported regarding two configurations, namely the *4x4_2x4* variant, which enables a more direct comparison with other 4×4 CGRAs, and the *5x2_2x4_2x1* variant, which represents the best-performing Mage implementation. As a matter of fact, *matmul* is mapped with UF3 on *4x4_2x4* and UF4 on *5x2_2x4_2x1*. For both configurations, results are reported for both scalar and vector implementations. Besides the measured 65 nm implementation, Table 4.19 also includes Mage results scaled at 22, 28, 40, and 45

Table 4.18 Comparison between Mage and other state-of-the-art DAE CGRAs.

Work	Node [nm]	Frequency [MHz]	PEA Area [mm^2]	AGU Area [mm^2]	PE Area [mm^2]	AGE Area [mm^2]
Mage 4x4_2x4_s	65	250	0.1221 ↔ 0.1615[d]	0.0457	0.0073 / 0.0096 / 0.0103[e]	0.0032
	40		0.0449 ↔ 0.0594	0.0168	0.0027 / 0.0035 / 0.0038	0.0012
	22		0.0132 ↔ 0.0174	0.0049	0.0008 / 0.0010 / 0.0011	0.00035
Mage 5x2_2x4_2x1_s	65	250	0.0769 ↔ 0.1031[d]	0.0320	0.0073 / 0.0096 / 0.0103[e]	0.0032
	40		0.0283 ↔ 0.0379	0.0118	0.0027 / 0.0035 / 0.0038	0.0012
	22		0.0083 ↔ 0.0111	0.0035	0.0008 / 0.0010 / 0.0011	0.00035
CASCADE [1]	40	510	∼0.130 [a]	∼0.010 [a]	∼0.006 [a]	∼0.001 [a]
STRELA [26]	65	250	0.217	0.027	0.014	0.003
NX-CGRA [17]	22	200	0.164 [b]	0.164 [b]	-	-

[a] Values are extracted from a chart, hence they may be slightly different.
[b] Values are reported for the AGU-PEA combination only.
[d] Mage PEA areas refer to scalar and vector PEA, respectively.
[e] Mage PE areas refer to scalar PE, vector PE, and vector-accumulating PE, respectively.

nm, obtained using scaling factors retrieved by DeepScaleTool [63]. Since Mage is largely dominated by memory macros, these scaling factors should be taken with care, as they may represent a best-case scenario with respect to standard-cell scaling. Nonetheless, with respect to the SRAM-based scaling factors adopted in the Mage paper [64], the factors used in this thesis result in a less favorable scaling scenario for Mage. Overall, because the considered CGRAs differ in technology node, array size, memory organization, and benchmark dimensions, the comparison should be regarded as an indication of Mage's position in the current CGRA landscape rather than as a strict one-to-one evaluation. Furthermore, it is worth noting that the MOPS figures reported across different CGRA papers may not have been derived using the same methodology. In this comparison, they are assumed to be computed as in Mage and STRELA, hence by accounting only for arithmetic operations, in order to enable the most architecture-independent comparison possible.

As regards **comparison with other DAE CGRAs**, **STRELA** is the most direct comparison reference, as it is also implemented at 65 nm and 250 MHz, and it also adopts a 4×4 organization with 8 AGEs. As regards area, Mage *4x4_2x4* is larger than STRELA, in both scalar and vector variant. However, the breakdown reported in Table 4.18 better clarifies this difference. Mage's PEA in *4x4_4x2* scalar and vector variants is 1.78× and 1.34× smaller than STRELA's. Furthermore, the area of scalar and vector PEs is also smaller than STRELA's PEs, although the PEs in the two CGRAs integrate similar operators. The main difference between the two PEs lies in STRELA's implementation of a ready/valid protocol to support elastic streaming

computation, and the support for *branch/merge* operations. Nonetheless, as also highlighted in section 2.3, although several architectural features are introduced to support dynamic control flow, they are hardly exploited across the evaluated workloads, since most of the mapped benchmarks reported in the paper are SCoP workloads. In addition, while AGEs area is comparable between Mage and STRELA, STRELA does not report which access patterns its AGEs support. In conclusion, the main area overhead of Mage comes from its usage of an internal SpM, which is the major contributor to its area occupation. However, the use of an internal SpM, together with a highly optimized data layout for avoiding access conflicts, allows Mage to achieve about 3× higher throughput than STRELA, even on a smaller *matmul* problem than STRELA, on an equally-sized-PEA implementation and with the same UF. The throughput gap further increases when *5x2_2x4_2x1* or vector variants are considered, although it is fair to observe that STRELA would benefit as well from the introduction of SIMD operators, partially reducing this difference. The same trend is reflected in both area and energy efficiency, with Mage reaching 74.6 MOPS/mW versus STRELA's 58.7 MOPS/mW, and up to 2896.o MOPS/mm^2 versus 1751.2 MOPS/mm^2, despite the larger occupied area and the smaller *matmul* benchmark. If the comparison is extended to Mage *5x2_2x4_2x1* configuration, the throughput gap increases even further, reaching about 4.5× in scalar `int32`.

The comparison with **CASCADE** is also favorable to Mage as regards area occupation. When considering scaled values to 40 nm, Mage's AGEs and AGU area are almost equal to CASCADE's, but they support up to 4D addressing schemes, while CASCADE's AGEs only support 2D incrementing addressing. However, performance comparison with CASCADE is not possible, since it only reports values relative to an unspecified baseline CGRA.

NX-CGRA is another relevant DAE CGRA comparison, since it also targets modern ML-oriented workloads and supports SIMD execution as well as non-linear operators. Moreover, its 4×4 organization with 8 AGEs makes the comparison with Mage *4x4_2x4* meaningful. As regards raw throughput, although NX-CGRA evaluation methodology for MOPS is not reported, Mage *4x4_2x4_v* on `int8` reaches 5237.6 MOPS, compared to 3040 MOPS for NX-CGRA on the same data width, and the gap increases considering the *5x2_2x4_2x1* variant (7776.0 MOPS). Nonetheless, the NX-CGRA's results refer to a 2× larger *matmul* problem. Therefore, the MOPS gap with would be even higher considering the same *matmul* size. In addition, while at 65 nm Mage has a technology disadvantage across metrics, the scaled 22 nm Mage

values suggest that Mage would become highly competitive, despite also including an internal SpM. In particular, for `int8` *matmul*, Mage *4x4_2x4_v* achieves an area efficiency at 65 nm that is slightly more than half of NX-CGRA's, whereas the scaled results reach 90087.2 MOPS/mm^2, corresponding to about 4× the area efficiency of NX-CGRA. As regards power consumption and energy efficiency, NX-CGRA still outperforms scaled Mage, with 2010.0 MOPS/mW versus 562.4 MOPS/mW (719.9 MOPS/mW on *5x2_2x4_2x1*). However, it is important to note that NX-CGRA does not account for system-level contributions such as memory power consumption, which would represent a non-negligible overhead in this context.

As for the **non-DAE CGRAs** in Table 4.19, Mage also usually achieves better throughput, area and energy efficiency, while sometimes providing higher power consumption.

Compared to **IPA**, *4x4_2x4* Mage configuration is clearly superior in throughput, around 20× on an equally-sized PEA (*4x4_2x4_s*), and up to 30× if *5x2_2x4_2x1_s* is considered. While considering results at 65 nm, Mage variants are roughly 2× larger than IPA, both Mage variants are highly superior in area efficiency. In addition, scaled area values suggest Mage would also be significantly smaller than IPA, which further increases the efficiency gap. If scaled values are considered, Mage also surpasses IPA in energy efficiency, while still consuming significantly higher power. Nevertheless, it is useful to highlight that IPA is specifically designed for low-power operations. Nonetheless, Mage balances its higher power consumption with high throughput, thus obtaining comparable energy efficiency to lower-power-targeted designs.

The same analysis holds for **Riptide**. Mage achieves significantly higher area efficiency (2896 MOPS/mm^2 vs. 656 MOPS/mm^2) already at 65 nm and about 8.8× throughput. However, also considering scaled values, Mage loses in terms of power consumption that, in the case of Riptide, takes into account also system contributions. This is again mainly due to Riptide tailoring for low-power operation. Nonetheless, Mage's throughput advantage lowers the energy efficiency gap with Riptide when considering scaled values at 22 nm (167.42 MOPS/mW for Mage vs 180 MOPS/mW for Riptide), and the gap could be in Mage's favor if equally-size *matmul* problems were compared.

In addition, compared to **R-BLOCKS**, Mage *4x4_2x4_v* achieves 2759.3 MOPS on `int16` compared to 678 MOPS for R-BLOCKS, together with substantially higher

Table 4.19 Comparison between Mage and other state-of-the-art CGRAs.

Work	Node [nm]	Frequency [MHz]	Area [mm^2]	PEA [R×C]	AGEs	Power [mW]	MOPS	MOPS/mW	MOPS/mm^2
Mage 4x4_2x4_s `int32`[a]	65	250	0.500	4×4	8 (4-4)	19.4	1446.7	74.6	2896.0
	45		0.245			15.1		95.6	5902.5
	40		0.184			14.2		102.2	7870.0
	28		0.086			10.9		133.2	16865.6
	22		0.054			9.3		155.3	26830.5
Mage 4x4_2x4_v `int32`[b]	65	250	0.539	4×4	8 (4-4)	20.4	1446.5	70.9	2683.0
	45		0.264			15.9		90.9	5474.7
	40		0.198			14.9		97.1	7299.6
	28		0.092			11.4		126.6	15643.1
	22		0.058			9.8		147.7	24885.7
Mage 4x4_2x4_v `int16`[b]	65	250	0.539	4×4	8 (4-4)	19.8	2759.3	139.4	5118.0
	45		0.264			15.4		178.7	10443.4
	40		0.198			14.5		190.9	13924.5
	28		0.092			11.1		248.9	29840.4
	22		0.058			9.5		290.3	47471.2
Mage 4x4_2x4_v `int8`[b]	65	250	0.539	4×4	8 (4-4)	19.4	5237.6	270.0	9715.0
	45		0.264			15.1		346.1	19823.2
	40		0.198			14.2		369.9	26430.9
	28		0.092			10.9		482.2	56641.9
	22		0.058			9.3		562.4	90108.1
Mage 5x2_2x4_2x1_s `int32`[a]	65	250	0.536	5×2	10 (5-5)	24.8	1985.0	80.0	3703.0
	45		0.263			19.3		102.6	7554.9
	40		0.197			18.1		109.7	10073.1
	28		0.092			13.9		143.0	21586.9
	22		0.058			11.9		166.7	34341.2
Mage 5x2_2x4_2x1_v `int32`[b]	65	250	0.562	5×2	10 (5-5)	26.1	1985.0	76.1	3530.0
	45		0.275			20.4		97.5	7205.3
	40		0.207			19.1		104.2	9607.1
	28		0.096			14.6		135.8	20588.2
	22		0.061			12.5		158.4	32752.5
Mage 5x2_2x4_2x1_v `int16`[b]	65	250	0.562	5×2	10 (5-5)	25.1	3935.0	156.8	6997.0
	45		0.275			19.6		201.0	14283.6
	40		0.207			18.3		214.8	19044.8
	28		0.096			14.1		280.0	40813.4
	22		0.061			12.0		326.6	64927.5
Mage 5x2_2x4_2x1_v `int8`[b]	65	250	0.562	5×2	10 (5-5)	22.5	7776.0	345.6	13828.0
	45		0.275			17.6		443.1	28226.0
	40		0.207			16.4		473.5	37634.7
	28		0.096			12.6		617.2	80651.8
	22		0.061			10.8		719.9	128304.0
CASCADE [1]	40	510	∼0.57	4×4	16 (8-8)	73.0[h]	-	-	-
STRELA [26]	65	250	0.25	4×4	8 (4-4)	7.5[c]	437.8[c]	58.7[c]	1751.2[c]
NX-CGRA [17]	22	200	0.178	4×4	8	1.5[d]	3040.0[d]	2010.0[d]	17078.7[d]
IPA [22]	28	100	0.20	4×4	-	0.5[e]	66.0[e]	134.6[e]	329.9[e]
DA-CGRA [57]	45	100	0.20	5×4	-	5.5[f]	2845.0[f]	529.3[f]	14225.0[f]
Riptide [20]	22	50	0.25	6×6	-	0.5[c] (0.9)	164.0[c]	328.0[c] (180.0)	656.0[c]
R-BLOCKS [19]	22	100	0.486	11×7	-	5.8[g]	678.0[g]	116.1[g]	1395.1[g]

[a] Values correspond to a 32×32×32 *matmul* on `int32`.
[b] Values correspond to a 32×32×32 *matmul* on `int32`, `int16`, or `int8`, depending on the row.
[c] Values refer to 64×64×64 *matmul* on `int32` data. Values in parentheses account for system contributions.
[d] Values refer to 32×64×32 *matmul* on `int8` data using packed SIMD operators.
[e] Values refer to 16×16×16 *matmul* on `int32` data.
[f] Values refer to 64×64×64 *matmul* on `int16` data.
The corresponding paper reports technology-scaled values at 22nm.
[g] Values refer to 32×32×32 *matmul* on `int16` data.
[h] Average value reported.

area efficiency and energy efficiency already at 65 nm compared to R-BLOCK's 22 nm implementation. The *5x2_2x4_2x1* configuration further strengthens this advantage.

Among the non-DAE competitors, **DA-CGRA** is the strongest comparison point in terms of normalized metrics. Mage achieves comparable throughput to DA-CGRA in *4x4_2x4_v* on `int16` *matmul*, while surpassing it (almost $2\times$) in its *5x2_2x4_2x1_v* variant. In order to compare technology-dependent metrics to DA-CGRA, 22 nm values should be considered, as DA-CGRA paper reports values scaled at that node. Considering these values, Mage outperforms DA-CGRA in terms of area efficiency on both scalar and vector variants, while still obtaining lower values regarding energy efficiency, although it should be highlighted that DA-CGRA is evaluated on an $8\times$ larger *matmul* problem. In addition, it is important to highlight that Mage still achieves significant area reductions compared to non-DAE CGRA across all variants if scaled values are considered.

In conclusion, although the reported scaled values should be interpreted with caution because they rely on scaling assumptions, they still indicate that **Mage surpasses existing DAE CGRAs in throughput, area efficiency, and energy efficiency by focusing on SCoP kernels and by effectively combining the DAE model with an optimized internal SpM data layout that enables conflict-free memory accesses at $II = 1$.** Moreover, despite this focus on SCoP workloads, **Mage still provides comparable code coverage and significant improvements over non-DAE CGRAs on SCoP kernels, while also achieving energy-efficiency results comparable to designs specifically targeting low-power execution**.

4.5 Conclusions

This chapter presented **Mage**, a template-based DAE CGRA specifically tailored for SCoP workloads. Mage combines an AGU driven by a hardware-loop unit with a GMP-compliant internal multi-bank SpM, enabling conflict-free SpM accesses within a fully-static execution model targeting $II = 1$. In this organization, memory address generation and static-control management are moved out of the PEA, allowing the PEA to focus entirely on arithmetic operations. A design-space exploration was carried out by exploiting Mage's template-based design, showing that this approach enables a wide range of architectural trade-offs.

The evaluation confirmed that Mage is highly effective on SCoP kernels, achieving high throughput together with strong area efficiency as well as substantial energy efficiency. Across the considered workloads, Mage achieved significant speedups and energy savings over the CV32E40P CPU baseline implementing the *XCVPULP* ISA extensions, reaching up to 32.5× speedup and 22× energy savings on scalar kernels. Vector variants further strengthened throughput and efficiency through packed-SIMD execution, also against manually optimized *XCVPULP* codes for kernels that are highly relevant in Transformer and Convolutional Network pipelines. In particular, the *5x2_2x4_2x1_v* reaches up to 7.8 GOPS, 345.6 MOPS/mW, and 13.8 GOPS/mm^2 on `int8` *matmul*. At the same time, the comparison among the different Mage implementations highlighted several design-space trends:

1. the maximum exploitable UF is mainly determined by the amount of PEA, AGU, and SpM resources available. In addition, **access-execute channel crossbars' granularity can improve mapping flexibility and enable higher UFs on selected kernels, while incurring low area cost**.

2. for a fixed UF, smaller architectures generally provide the best area efficiency, since larger variants do not fully exploit their additional hardware. However, **once larger implementations unlock higher UFs, they can recover and often overcome this gap**, especially on compute-dense kernels such as *gemm*, where the increase in parallelism compensates for the larger area. A similar trend is observed for energy efficiency. At fixed UF, MOPS/mW tends to decrease slightly as the architecture grows, but the highest sustainable UF remains the dominant factor, making larger variants more attractive whenever they enable additional unrolling.

3. **kernels with higher arithmetic intensity achieve better MOPS-normalized efficiency**, since they perform more arithmetic work per transferred operand, while SpM activity remains the dominant contributor to power consumption.

4. **vector execution on reduced-precision data provides the largest gains in MOPS-normalized metrics**. Since the area overhead of vector PEs is moderate, packed-SIMD execution translates almost directly into higher throughput, especially for `int16` and `int8`, thus significantly improving both area and energy efficiency.

5. keeping memory-bandwidth capability fixed, most of the considered SCoP workloads achieve the best performance and area/energy efficiency on *rectangular* PEAs rather than on *squared* ones. In contrast, *gemm* and *hydro* not only deliver better results on *squared* PEAs, but cannot be mapped onto the considered *rectangular* implementations. Therefore, for a given memory bandwidth, *squared* PEAs are more suitable for workloads whose loop body contains long chains of dependent operations, while *rectangular* PEAs are more effective for workloads with limited operation depth, where exploiting higher unrolling mainly increases bandwidth requirements.

In addition, the end-to-end TCN case study showed that these kernel-level gains can also translate into application-level acceleration, with overall speedups up to 12.9× on `int16` and 9.7× on `int8`.

The comparison with state-of-the-art CGRAs further highlights Mage's strengths. Against STRELA, Mage achieves about 3× higher throughput on an equally sized PEA, at the same technology node and frequency, while also improving both area and energy efficiency. Compared with NX-CGRA, Mage achieves higher raw throughput on `int8` *matmul*, and scaled results suggest very strong competitiveness also in area efficiency. It is worth noting that the comparison with STRELA, the only DAE CGRA evaluated at the same 65 nm node and 250 MHz frequency, is based entirely on measured results, where Mage already achieves about 3× higher throughput along with higher area and energy efficiency. The comparisons against NX-CGRA and DA-CGRA, by contrast, rely substantially on Mage's results scaled to 22 nm, and should be interpreted with the caution discussed in subsection 4.4.6. Overall, the comparison demonstrates that **the co-design of AGU, tailored SpM data layout, and scheduling strategies enabling conflict-free memory access allows higher throughput while also delivering higher area efficiency and competitive energy efficiency**. In addition, Mage **outperforms non-DAE CGRAs in both throughput and area/energy efficiency**, while sometimes trading these gains for a higher power budget. Overall, the results indicate that explicitly specializing a DAE CGRA for SCoPs, and co-designing AGU capabilities, static scheduling, and SpM data layout, is an effective way to achieve very high performance together with strong area and energy efficiency.

Chapter 5

Rogue: A Streaming DAE CGRA for Integer Activation Approximation

5.1 Introduction & Motivations

In recent years, compute-intensive ML models have increasingly been deployed on resource-constrained, energy-efficient edge platforms. This trend has been enabled by the progressive reduction of model sizes and by the adoption of heterogeneous SoCs that rely on General Matrix Multiplication (GeMM) accelerators. These are specialized units that provide high-throughput and energy-efficient execution of the GeMM kernels, which dominate ML workloads. As a result, accelerator design has historically focused on GeMM, while the remaining parts of the inference pipeline, activation functions, were either simple enough for software or could be partially mapped onto GeMM-oriented units.

However, while ReLU was predominant in earlier deployments, it has been increasingly replaced by more complex nonlinear functions such as Tanh, Gaussian Error Linear Unit (GELU), and Softmax [65]. At the same time, the shift from CNN-centric workloads to attention-based Transformer models, together with the widespread use of GeMM accelerators, has moved the performance bottleneck toward activation layers, as also observed in recent work [66, 67]. In [67], the deployment of *GPT3-XL* on a RISC-V multi-cluster platform shows Softmax accounting for 30% of runtime without GeMM acceleration and 70% after enabling GeMM acceleration on a sequence length of 2048. A similar trend is reported in [66], where authors run

ViT-Base on a PULP cluster integrating tensor processing units of various sizes for GeMM acceleration, while leaving non-linearities (GELU and Softmax) to RISC-V cores. A 12×4 tensor unit yields a 12.3× speedup over 8-core software execution, but further scaling the tensor unit brings diminishing returns due to Softmax and GELU. Hence, a 4× larger tensor unit provides only an additional 2.54× speedup, i.e., 63% of the expected gain. Besides limiting performance, this also reduces energy efficiency.

Therefore, **existing GeMM accelerators alone struggle to meet edge AI's latency and energy targets, as they are not well-suited for nonlinear activation functions**. This has motivated research on integer-friendly approximations of such functions [68–70] and on dedicated hardware support. While many existing accelerators target a single activation or a reduced family of approximations, as discussed in section 5.2, **there is growing interest in more versatile platforms that support a broader set of activations efficiently**.

Interestingly, integer approximations of activation functions exhibit algorithmic and computational properties that make them a good match for CGRA-based acceleration using the DAE execution paradigm:

1. Despite targeting different nonlinear functions and using different approximation strategies, *integer activations tend to reuse a common set of computational patterns*, including multiply-add chains for polynomial evaluation, shifts and clipping for range handling, and division- or exponent-related sequences. These patterns can be captured by a dedicated ISA and implemented efficiently by CGRA PEs equipped with standard arithmetic operators.

2. **all integer activations can be described as SCoP workloads**, as they are mainly characterized by regular loop nests with with compile-time-resolvable bounds and affine indexing of arrays. As a consequence, they are suitable candidates for acceleration on DAE-based CGRAs. Hence, address generation and data movement can be offloaded from the compute array to specialized units, preserving a control-free PEA and enabling fully static schedules whenever the approximation structure allows it.

3. *Integer activation kernels commonly feature limited inter- and intra-iteration data dependencies*, often restricted to reduction patterns for accumulations.

This feature makes them well suited to streaming execution, which can be efficiently implemented in DAE CGRA architectures.

Based on these insights, **Rogue** is proposed as a DAE-based CGRA tailored for streaming execution of integer approximations of activation functions. Rogue combines a dedicated ISA, designed to cover the recurring patterns of these approximations, with a streaming-oriented architecture. Since the ISA includes add, multiply, and shift operations, Rogue is also applicable to GeMM-style kernels and can execute conventional multiply-accumulate workloads in addition to activation functions.

Architecturally, *Rogue does not include an internal AGU or local data memory*. On the other hand, it exposes a FIFO-based interface that enables direct coupling to external AGUs and, in particular, to system DMA engines, which act as AGUs in the DAE setting and stream data directly from main memory into the PEA. The exploitation of direct DMA coupling allows integrating computation within data transfer, while also reducing system-level area overhead and pressure on the system bus. As a matter of fact, this approach does not require adding additional master/slave ports, and avoids increasing interconnect (crossbar) complexity, which is a critical cost driver in scaled implementations.

5.2 Related Works

In this section, the approaches adopted in literature regarding activation functions approximation strategies and hardware acceleration are discussed, highlighting the related advantages and disadvantages.

Early efforts in accelerating activation functions targeted the CNN and Long-Short Term Memory (LSTM) domains. In this context, while *ReLU* presents limited complexity and can be easily implemented in software or onto existing GeMM accelerators, particular focus was put on *Sigmoid* and *Tanh* [71, 72], as they exhibit increased complexity due to the presence of the exponential function. Then, as Transformer models became prevalent, the recent focus has shifted to non-linearities found in these models, specifically *Softmax* [73, 74], which is highly exploited in attention layers, and *GELU* [75, 66], adopted in feed-forward layers, which can be sometimes replaced by *SiLU* [76]. Through the years, several accelerators for

activation functions have been proposed, which can be mainly classified based on the adopted **approximation methods** and the **target functions**.

Regarding approximation techniques, the adopted solutions in literature are mainly three:

1. **Look-Up Tables** (LUTs) are commonly employed [77, 71] to store pre-computed values that approximate a target activation function. In LUT-based approaches, the input domain of the function is divided into a set of intervals, and each interval is associated with a pre-computed output value that is stored in memory elements used as LUTs. Then, an addressing scheme is used to map a given input, or an entire input interval, to a specific LUT entry containing the corresponding approximated function output. This mechanism substantially reduces computation time compared to polynomial approximation methods (discussed below), as it removes the need to explicitly evaluate the activation function for each input. Moreover, LUT-based implementations provide great flexibility since programmable LUTs can be reconfigured to store different sets of output values corresponding to different activation functions. However, achieving high approximation accuracy typically requires substantial area due to high memory footprint. As a matter of fact, the approximation accuracy depends strictly on the number of intervals used to partition the input domain (i.e., the LUT depth) and on the precision of the stored output values (i.e., the LUT width).

2. **Polynomial Approximation** (PA) methods [68, 78, 16] exploit polynomials for direct calculation of activation functions mainly employing multiply-add sequences. This approach can deliver high precision with reduced latency computation, but it can become area-hungry if high-order polynomials are used to minimize accuracy loss. Nonetheless, *high accuracy has been recently achieved with low-order (2^{nd}) polynomials for integer-based GELU and Softmax computation by the authors of i-BERT [68].* In addition to pure PA-based approximation, hybrid approaches have also been proposed where LUT are exploited in combination to polynomial computation. For instance, **Piece-Wise Polynomial Approximation** (PWPA) [79, 72, 65] and **Piece-Wise Linear Approximation** [80, 81] strategies employ a reduced amount of LUT storing coefficients to compute low-order polynomials and linear segments,

respectively. Therefore, they balance memory footprint and combinational area while achieving good approximation results.

3. **Base Conversion** is a less common approach than LUT and PA, but several works [69, 82, 74] leverage it for base conversion from e to 2 in exponential-based activations, thus enabling intensive use of bit-shifting operations that are highly hardware-friendly.

As regards target activations, most accelerators focus on a limited set of functions or, in most cases, a single one. In the LSTM domain, several works [72, 71] address fixed-point acceleration of Tanh and Sigmoid. In the Transformers domain, SoftEx [73] proposes a floating-point accelerator specifically designed for Softmax that is also capable of computing GELU [66]. On the other hand, in the integer computation domain, ITA [74] implements a Softmax acceleration unit tightly connected with a GeMM accelerator to perform efficient execution of attention layers within Transformer models. Nonetheless, *recent research has highlighted the need for reconfigurable accelerators that can be adapted to existing and future non-linear activation functions*. State-of-the-art programmable accelerators employ different approximation strategies to support a wider range of activations. For example, NACU [80] and [81] exploit PWLA to approximate the exponential function, Sigmoid and Tanh, while Flex-SFU [65] and [79] leverage a PWPA strategy that can be applied to a very wide range of activation function. In addition, PICACHU [16] highlights that PA of activations share computational characteristics that perfectly fit acceleration on CGRA.

Building on these observations, an increasingly growing trend towards reconfigurable architectures capable of supporting a wide range of activations is present in literature. In this context, a crucial additional observation is that *integer-based approximations, not only PA-based ones, share similar computational patterns and loop-based algorithmic structures with minimal data dependencies, making them well-suited not only for CGRA-based acceleration, but especially for DAE-based CGRA acceleration.* Therefore, based on these insights, a dedicated ISA that captures recurring patterns, such as multiply-and-add, exponentiation- and division-based sequences, is proposed and implemented by Rogue, which also employs a streaming-capable DAE design that can be easily integrated within existing system DMA.

Overview and Organization

In the remainder of this chapter, an in-detail presentation of Rogue is provided, along with its evaluation based on the results obtained from its implementations. In particular, section 5.4 introduces the proposed ISA, designed to capture recurring patterns in integer approximations of activation functions. Section 5.5 describes Rogue's internal architecture, section 5.6 explains how kernels are mapped and executed on Rogue, and section 5.7 discusses the results obtained on the various investigated Rogue's implementations. Before introducing Rogue itself, section 5.3 describes the interface that enables direct coupling of external streaming accelerators to the X-HEEP DMA subsystem. This interface, which is also part of the X-HEEP's XAIF presented in section 2.4, was developed within this thesis together with its first usage example: a DMA-coupled streaming accelerator, also discussed in section 5.3, representing the first instance of DMA-coupled accelerator integrated in an X-HEEP-based chip.

5.3 System Integration of Streaming Accelerators

The following section mainly addresses two topics: ① the FIFO-based interface adopted by X-HEEP's XAIF to allow connection of external streaming accelerators to the DMA subsystem, and ② the first example of a DMA-coupled streaming accelerator integrated into an X-HEEP-based SoC.

5.3.1 DMA-Coupled Accelerator Interface in X-HEEP

In recent years, the need for low-latency and energy-efficient data transfers along with flexible addressing is increasingly fundamental to meet latency and energy requirement in edge AI pipelines. For this reason, significant effort has been devoted to the development and enhancement of the X-HEEP DMA subsystem [60]. X-HEEP provides a highly configurable DMA subsystem in which designers can select both the *number of channels* and the *number of master ports* exposed to the system bus, with the subsystem automatically handling the required crossbar interconnection.

As regards the internal structure of a single DMA channel, it mainly includes address generators for source and destination address calculation, units that translate

addresses into OBI-compliant transactions, and internal FIFOs used to temporarily store data read from the source before it is written to the destination. Moreover, X-HEEP allows the functionality of each DMA channel to be tailored to the designer's needs, so that only the required features are instantiated. In particular, the available channel features are:

- **1D Transfers**: support for strided linear transfers.
- **2D Transfers**: support for strided transfers on two dimensions, which are especially useful for reading/writing matrices and tensors, as well as sub-portions of them.
- **Zero-Padding**: automatic generation of zero-valued data for padded regions, avoiding software handling of boundary conditions.

Within the scope of this thesis, this set has been extended with an additional feature aimed at **enabling the tight coupling of external streaming accelerators to DMA channels**. Building on the observations reported in [83] on the benefits of embedding computation directly within the DMA data-transfer path, X-HEEP DMA channels have been enhanced with a **FIFO-based interface**, used in the so-called **hardware-fifo mode** (or *hw-fifo* mode). This interface is part of X-HEEP's XAIF and it allows an external streaming accelerator to be directly connected to a DMA channel, so that computation can be performed directly while data is being transferred. The FIFO-based interface is organized into two sub-interfaces, which together implement a FIFO-style communication protocol between DMA and accelerator:

1. **Request FIFO Interface**. This sub-interface is driven by the DMA channel and it is used to send commands and input data to the accelerator. It includes:
   - *push*: 1-bit signal asserted by the DMA channel when a new input data is being offloaded to the accelerator.
   - *pop*: 1-bit signal asserted by the DMA channel when data has to be retrieved from the accelerator.
   - *flush*: 1-bit signal asserted by the DMA channel to issue the flushing of the accelerator's FIFO-like buffers.
   - *data*: data to be offloaded to the accelerator.

2. **Response FIFO Interface**. This sub-interface is mainly driven by the accelerator and it is used to expose its status to the DMA channel. It includes:

    - *full*: 1-bit signal asserted by the accelerator when its input FIFO-like buffer is full, thus preventing the DMA channel from pushing additional data.
    - *empty*: 1-bit signal asserted by the accelerator when its output FIFO-like buffer is empty, thus preventing the DMA channel from popping processed data.
    - *data*: data to be retrieved from the accelerator.

Therefore, in order to support this mode, the accelerator must integrate input and output buffering elements with FIFO-like behavior. However, these storage elements do not necessarily need to be implemented as deep FIFOs and, depending on the accelerator design and throughput requirements, they may also consist of single registers, provided that they support the same rather simple push/pop-style interaction of FIFOs.

When a DMA channel is programmed in *hw-fifo* mode, it performs address computation and OBI-based bus accesses as in a standard data transfer. However, data read from the source do not go through the channel's internal data FIFOs. Instead, it is offloaded to the external FIFO-based interface and offloaded to the integrated streaming accelerator. Results are then retrieved from the accelerator and finally written by the DMA channel to the destination location. In addition, it is important to highlight that *hw-fifo* mode is compatible with 1D transfers, 2D transfers, and zero-padding. This enables flexible and structured data movement toward the integrated accelerator, without sacrificing the addressing capabilities already provided by the DMA subsystem.

Overall, this FIFO-based interaction enables computation to be directly embedded within DMA transfers. Compared to integrating a streaming accelerator through one or more independent master ports, this solution reduces system-level area overhead, since the accelerator does not require dedicated LSU, and it also avoids further increasing the complexity of the system bus crossbar, which becomes particularly beneficial in scaled technologies. Nonetheless, it is important to highlight that this approach causes the streaming accelerator's peak bandwidth to be dependent on the DMA's architectural parameters and bus access. However, X-HEEP's DMA offers

extensive customization of the number of channels, master ports, and addressing features to meet the desired area and throughput requirements.

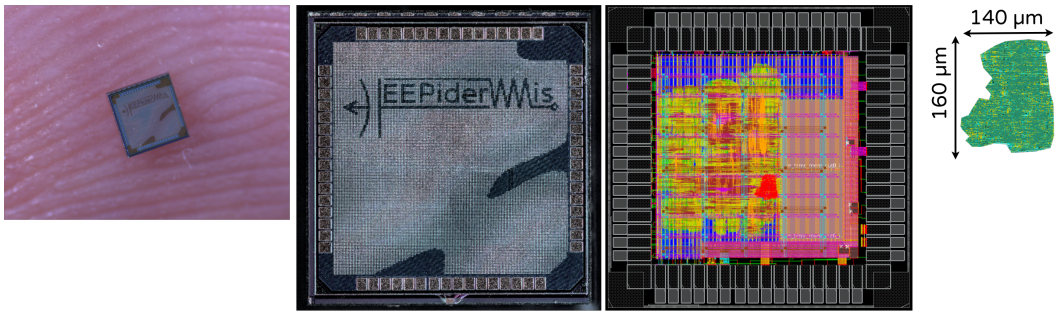


Fig. 5.1 (a) Heepidermis SoC picture on a finger. (b) Heepidermis SoC picture taken using a microscope. (c) Heepidermis SoC layout view in which the portion highlighted in red in the dLC module. (d) dlC area occupation within the Heepidermis layout.

5.3.2 HEEPidermis & Digital Level-Crossing Module

The feature presented in subsection 5.3.1, which allows external streaming accelerators to be tightly coupled with X-HEEP's DMA, has been exploited for the first time in the **HEEPidermis** [84] mixed-signal SoC. HEEPidermis is a versatile System-on-Chip for *biological impedance* (*BioZ*) recording, built upon an X-HEEP-based digital back-end and a custom analog front-end, and taped out on the *TSMC* 65nm *Low-Power High-Voltage Threshold* process. The digital subsystem encompasses the low-power CV32E20 RISC-V core, a fully connected system bus, two 16-KiB SRAM banks, standard communication peripherals (SPI and JTAG), and two DMA channels capable of implementing 1D transfers and integrating external streaming accelerators. On the other hand, the analog front-end integrates the main blocks required for impedance sensing, including a dedicated LDO, an on-chip voltage reference, a current reference, two 8-bit current DACs (*iDAC*) for programmable current injection, and two VCO-based ADCs for voltage acquisition. Based on this heterogeneous structure, HEEPidermis can autonomously acquire, transfer, and process sensed data on chip, enabling flexible and low-power biological impedance measurement applications.

Building on the DMA FIFO-based interface described in subsection 5.3.1, HEEPidermis digital back-end integrates a **Digital Level-Crossing** (*dLC*) module as a

streaming peripheral attached to a DMA channel operated in *hw-fifo* mode. In this context, the dLC acts as an *event-based sub-sampler* for data coming from the ADCs inserted directly within the DMA data transfer path between the sample source and system memory. Rather than storing every acquired sample, it analyzes the input stream of data and it only provides the DMA with output data when the signal undergoes a sufficiently large variation. This event-based sub-sampling mechanism allows the system to reduce memory traffic and storage, while retaining useful characteristics of the initial signal. In addition, the dLC can also monitor the amplitude of the incoming data and subsequently wake up the CPU in case of significant events, thus enabling an event-based operation of the SoC. The dLC design is provided as open-source and integrated within the X-HEEP's github repository as a peripheral, with the in-detail documentation available at https://x-heep.readthedocs.io/en/latest/ExternalPeripherals/dLC.html.

dLC Internal Structure. The dLC has been designed as a streaming peripheral coupled with the X-HEEP's DMA through the FIFO-based interface described in subsection 5.3.1. Therefore, the dLC receives (up to) 16-bit input samples from the DMA and it compares them against pre-defined levels, thus replicating the behavior of a *Level-Crossing ADC* [85] in the digital domain.

Internally, the dLC keeps track of the last crossed level, and it compares each new sample against a programmable level width that can be set in software through a configuration register. If the new sample remains within the same level, it is considered as *redundant* and it is discarded while, at the same time, an internal counter tracks how many input samples have been skipped since the last output. On the other hand, if the input sample crosses one or more levels, the dLC produces an output word containing the **level difference**, denoted as `dlvl`, together with the **temporal distance** from the previous output, denoted as `dt`. At the same time, the dLC raises a **crossing event signal**, referred to as `xing`, which can also be used to generate an interrupt to wake up the CPU. In addition, a **direction** output, denoted as `dir`, can be generated and, within a SoC, exposed to an I/O pad in order to provide information about the direction of the crossing to an off-chip SNN or for data transmission.

Furthermore, the output format of the dLC is configurable from software. In particular, the designer can select the number of bits allocated to `dlvl` and `dt`, re-

spectively, and the representation to be used for the signed level difference, selecting between *two's complement* or *sign-plus-magnitude*. Before being pushed back to the DMA through the FIFO-based interface, the generated event is packed into a 16-bit word using the selected format.

In addition, the dLC handles **overflow** conditions. If the level difference exceeds the representable range, the dLC produces and pushes back to DMA multiple outputs, setting $\texttt{dt} = 0$ for all words, so that they are all associated with the same time instant. Similarly, if no crossing occurs for a long time interval and the `dt` field overflows, the dLC emits a word with null level difference and saturated `dt`. Beyond this standard event-compression mode, the dLC also supports a **bypass mode**, in which the original input samples are preserved while crossing events are still generated, and a **differentiation** mode, in which the module effectively outputs sample-to-sample differences.

Within HEEPidermis, the dLC is integrated as an external streaming peripheral coupled to one of the two DMA channels, enabling on-the-fly event-based sub-sampling through the system DMA. A significant advantage of this integration methodology is that it preserves the flexibility of the analog sensing subsystem. Data can be acquired from either on-chip VCO-based ADC, from both ADCs, from external ADCs accessed through SPI, or from $\Delta\Sigma$ ADCs and then followed by the same digital decimation strategy implemented by the dLC. All these acquisition paths can be operated autonomously through the DMA and can be routed through the dLC for event-based sub-sampling. Therefore, the dLC is not tied to a single front-end block, but rather acts as a reusable streaming post-processing stage that can be inserted after multiple data sources within the SoC.

Application-level use of the dLC in HEEPidermis. At application level, the dLC is used in HEEPidermis alongside iDACs and VCO-based ADCs to support various use cases while retaining the possibility to be reconfigured in real-time to tradeoff sensitivity, range, data-rate and power to make up for long-term variations. Two possible uses cases involving the dLC are presented in the following.

A first example is **Galvanic Skin Response** (*GSR*). In this configuration, a single VCO-based ADC is used to sense the voltage associated with a DC current injected through the skin. Since the operating point of the VCO affects both distortion and power, the sensed voltage should remain within a desired range. In this case, the

dLC is configured as a range-monitoring stage. Therefore, the output of the ADC is passed through the dLC, which compares the incoming signal value against the programmed thresholds. When the signal exits the range, the dLC raises an interrupt that wakes up the CPU, which can then adjust the current injected by the iDAC to restore a suitable operating point. As a result, the digital back-end can remain quiescent, with only sporadic DMA transfers and occasional CPU interventions.

A second example is the **impedance measurement** mode. In this case, one DMA channel transfers pairs of de-phased sine samples to the two iDACs, while the two VCO-based ADCs sense the resulting voltages and operate in pseudo-differential mode. The second DMA channel periodically acquires the digital output difference and streams it through the dLC. The dLC compares the data against pre-determined thresholds discarding small variations, and sporadically emitting compressed output words which are stored into memory. After a fixed number of output words, the DMA wakes up the CPU so that the phase difference can be changed and the process relaunched, or the final impedance can be computed. In this way, the dLC reduces the amount of data that must be stored and processed, while preserving the information required by the higher-level impedance-estimation algorithm.

Overall, the dLC shows how the DMA-coupled streaming interface can be exploited not only to move data efficiently, but also to embed lightweight processing directly within the transfer path. In HEEPidermis, this capability is used to transform dense streams of raw samples into sparse event-driven representations, thus reducing memory activity, limiting CPU intervention, and enabling long-term autonomous operation of the SoC.

5.4 Proposed Instruction Set Architecture

As previously discussed, various integer approximation strategies for activation functions share common computational patterns. For example, PAs generally rely on multiply-and-add sequences, where polynomial terms are multiplied by their coefficients and then summed together. In addition, they often use other operators, such as shifts, exponentiations, clipping, absolute value, and division or remainder operations. Interestingly, approximations that exploit the base conversion from e to 2 also exhibit similar operational patterns. Therefore, a dedicated ISA is proposed to expose recurring operations and patterns that can be found in the approximations

Table 5.1 Description of proposed ISA instructions. The latency of Rogue's implementation is also reported.

Type	Name	Function	Latency
	ABS	$abs(a)$	1
	ADD, SUB, MUL	$a[+,-,*]b$	1
	LSH, ARSH, LRSH[a]	$a[<<,>>]b$	1
	MAX, MIN	$max(a,b), min(a,b)$	1
	DIV, REM	$a[/,\%]b$	L_{div}[b]
S	SGNSEL	$a > 0\ ?\ a : b$	1
	ABSDIV	$abs(a)\ /\ b$	$L_{div}+1$[b]
	ABSMIN	$abs(a) > b\ ?\ b : a$	2
	ABSREM	$abs(a)\ \%\ b$	$L_{div}+1$[b]
	ADDPOW	$(a+b)*(a+b)$	2
	SUBPOW	$(a-b)*(a-b)$	2
	SGNCSUB	$a > 0\ ?\ (C-b) : (b-C)$	2
	CADDMUL	$(a+C)*b$	2
	ADDCMUL	$(a+b)*C$	2
C	CMULADD	$(C*a)+b$	2
	CADDDIV	$(a+C)\ /\ b$	$L_{div}+1$[b]
	MULCARSH	$(a*b) >>> C$	2
	CLSHSUB	$(a << C)-b$	2

[a] ARSH uses signed right shifting, while LRSH unsigned.
[b] Divider's latency is $L_{div} = \frac{32}{log_2(radix)}$, and it is the same for serial and pipelined dividers, but pipelined ones have higher throughput.

of integer activations. Although implementation-independent by definition, the ISA has been primarily designed to efficiently execute integer approximations of activation functions on CGRAs. Therefore, the main aim of the proposed ISA is to combine multiple operators into single instructions that can be applied across different approximations, while limiting the number of variable operands. The main reason behind this choice is that combining multiple operations within a PE can improve utilization and efficiency, as prior CGRA studies have also highlighted [31]. Multi-operator instructions allow activation kernels to be mapped onto fewer PEs, while restricting the number of variable inputs per instruction reduces both interconnection overhead and PE internal complexity.

The proposed ISA employs 13-bit-wide instructions, in which 5 bits are devoted to instructions' opcode, and two 4-bit fields contain operand selectors. Instructions

are divided into two groups that differ regarding the number of input operands, and are discussed in the following:

- `S-type` instructions encompass one- and two-operand operations. Among them, more common instructions can be found, such as integer addition, subtraction, multiplication, left and right logic and arithmetic shifting, maximum and minimum and comparison operations. In addition, instructions based on division and remainder are also present, as they are a crucial computation element in integer approximation of activation functions. Furthermore, instructions are also present that can be useful mostly in PA-based approximations, such as `ADDPOW` and `SUBPOW`.

- `C-type` instructions feature three-operand instructions in which one operand is fixed to a constant value *C*. This choice has been enacted since many computations in integer approximations of activations involve constant coefficients used as multipliers, divisors, or shift amounts. `C-type` instructions mainly group single operators to compose more complex operational patterns that can be reused across different approximation strategies. In addition, some of these instructions can also be profitable when accelerating GeMM workloads, such as `CADDMUL`, `CMULADD` or `MULCARSH`. As for the nomenclature for these instructions, the *C* is posed before the operator that uses the constant *C*. As regards the CGRA implementation side, the choice of having instruction employing a constant operand reduces PEA complexity, as a simple register can be used to store the constant compared to increased routing complexity needed in case the third operand would be retrieved by neighbor PEs.

A complete list of instructions is provided in Table 5.1. The table also reports the latency specific to Rogue's implementation, although the ISA does not impose any latency constraints, and other designs can implement it differently.

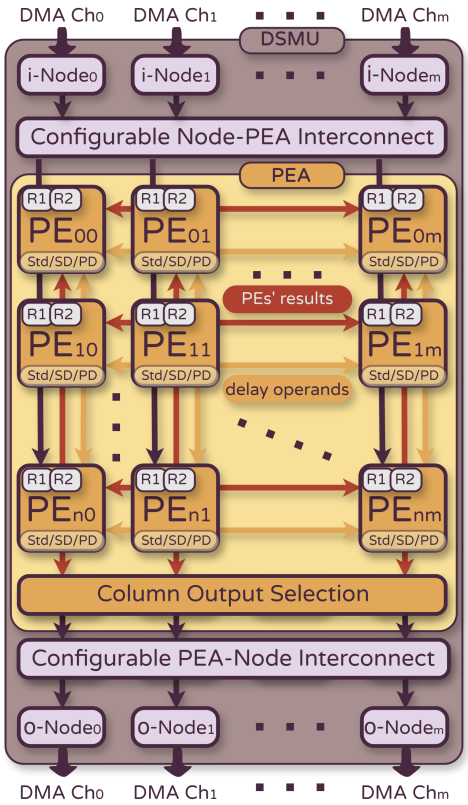


Fig. 5.2 Overall template-based architecture of Rogue encompassing PEA and DSMU.

5.5 Rogue Architecture

In this section, Rogue's architecture is discussed in detail. Rogue supports streaming execution of integer approximations of activation functions through two main blocks, depicted in Figure 5.2. The **PEA** performs computation, using a validity-based protocol between PEs and a global PEA *ready/stall* status to sustain streaming execution. Also, each PE implements the ISA (or part of if) introduced in section 5.4. Moreover, the **Data Stream Management Unit** (DSMU) provides the FIFO-based interface that connects the PEA to external AGUs or to system DMAs. Through this interface, Rogue can be integrated with existing DMA subsystems [61, 60] without adding extra bus masters, thus avoiding additional pressure on the system interconnect. Rogue also includes a set of configuration registers that program both DSMU and PEA. These registers are accessed through a single OBI-like slave port

exposed for configuration purposes. When integrated in a microcontroller SoC, the registers are memory-mapped and can be read and written by the CPU.

Rogue's architecture is configurable in terms of PEA/DSMU sizing, DSMU-PEA crossbar complexity, and PE placement. These configurations are realized through the same template-based RTL flow used for Mage, based on `SystemVerilog` and `Python mako`, where `.sv.tpl` sources are processed by a `Python` script to generate synthesizable `.sv` files.

The following sections detail the DSMU and PEA organization, together with the template parameters used for RTL generation and the configuration registers used to program Rogue.

5.5.1 Data Stream Management Unit

The DSMU is responsible for interfacing Rogue with external AGUs or DMA channels. It buffers input data destined for the PEA, and collects the results computed by the PEA before transferring them back to the external AGUs or DMA channels. Although Rogue can be connected to any external AGU or DMA, only the DMA case will be considered in the following discussion, since the implemented version of Rogue is integrated into the DMA subsystem of X-HEEP. Nevertheless, the same considerations are applicable to any AGU that is compliant with the FIFO-based interface employed by Rogue's DSMU. In Figure 5.3, Rogue's DSMU is depicted along with its template parameters and configuration registers, which are described in detail in Figure 5.5 and Figure 5.4, respectively.

At a high level, the DSMU provides ① input and output data buffering through FIFOs for each DMA channel, ② programmable data stream routing between DMA channels and PEA columns, ③ optional and flexible synchronization among input data streams, and ④ finite-length streaming through programmable down-counters. The DSMU is primarily composed of the three building blocks discussed in the following.

I/O Nodes Input/Output (I/O) nodes are FIFO storage elements used to decouple the DMA channels from the PEA. Each channel connected to Rogue is associated with one *input node* (DMA $\rightarrow$ PEA) and one *output node* (PEA $\rightarrow$ DMA). The total number of nodes is a template parameter (`n_nodes`) and it has to match the number

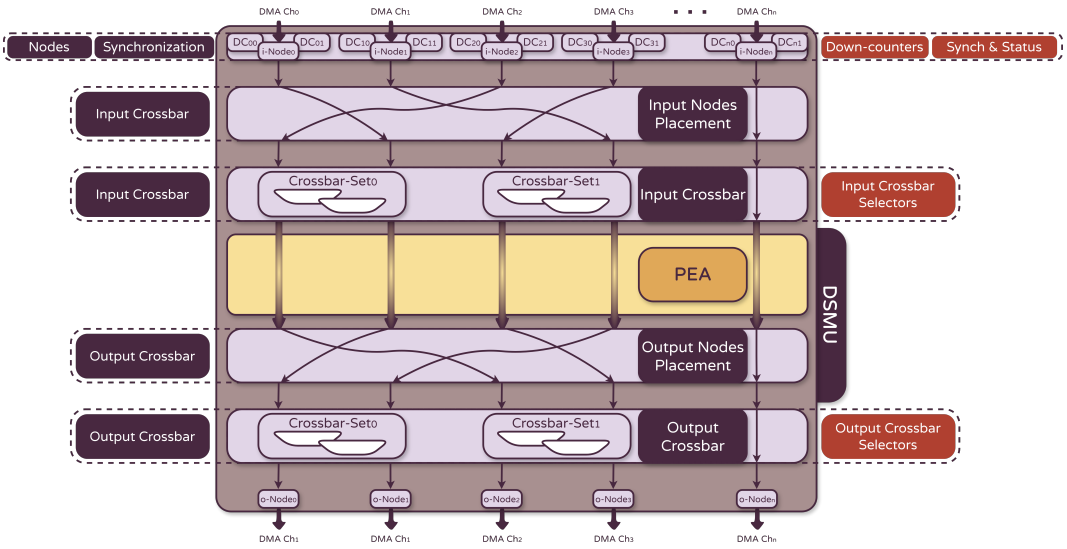


Fig. 5.3 Rogue's DSMU template-based structure, along with template parameters and configuration registers identified with the same "section" titles used in Figure 5.5 and Figure 5.4, respectively.

of connected DMA channels. Input nodes receive data from the associated DMA channels, and they feed PEA columns through a set of template-configurable *input crossbar sets*. On the other hand, PEA columns feed output nodes through *output crossbar sets*, thus providing DMA channels with data to be written back to memory. A DMA channel may be used as a *load channel* for pushing values into an input FIFO and, in turn, into Rogue's PEA, or as a *store channel* for draining values from an output FIFO back to memory, or also both.

In addition, the DSMU can perform **stream synchronization** across selected input nodes when the PEA requires aligned consumption of multiple input data streams. At design time, template parameters (`is_sync_plc_size`, `is_sync_plc_g2`, `is_sync_plc_g3`, `is_sync_plc_g4`) define which nodes may participate in synchronization groups, i.e., which dependency patterns are allowed. At run time, desired synchronization groups are enabled and sized by the related configuration registers (`is_synch_groups`, `sync_dma_ch_trans`). When enabled, the DSMU coordinates input FIFO pop operations so that the PEA receives a consistent relative ordering across the synchronized streams. It is important to highlight that the synchronization enacted at the DSMU level simplifies the ready/valid protocol adopted among PEs, as input data is guaranteed to be properly aligned before reaching the

PEA. The synchronization of input nodes is further discussed in the next paragraph, as it is handled by down-counters.

Down-Counters Down-counters are employed to implement bounded transfers and support the synchronization mechanism of input data streams. Each I/O node couple associated to a DMA channel is equipped with a 32-bit down-counter that bounds the transaction length. The counter is initialized through a configuration register (`trans_size_dma_ch`) and decremented at run time based on the channel role, i.e. load or store. Therefore, it can be configured (`dma_rnw`) to decrement either when the DMA pushes an element into the corresponding input FIFO, or when the PEA pushes an element into the corresponding output FIFO. When a counter reaches zero, the DSMU signals the associated DMA channel to stop the transaction. Rogue execution terminates once all 32-bit counters reach zero.

Moreover, each input node is also equipped with a 16-bit down-counter to implement the synchronization strategy discussed in the previous paragraph, which will be explained using a practical example. Suppose input node i and j have to be synchronized in such as way that the DSMU pops an element from input FIFO i once every n element are popped from FIFO j. In this case, the configuration register (`sync_dma_ch_trans`) of node i must be set in order to let the DSMU know that it has to be synchronized with node j. Then, the 16-bit down-counter related to input node j has to be set to n through the related configuration register (`trans_s_size_dma_ch`). Only when it reaches zero, the DSMU pops an element from input node i at the same time as a pop operation is performed on node j. If n is set to zero, it results in synchronous popping from input FIFOs i and j. As discussed in the previous paragraph, template parameters can be used to define, at design time, the complexity of this synchronization mechanism, meaning that they can be used to decide how many input nodes can be configured as dependent on each other. Once decided, a configuration register (`is_synch_groups`) is used to define the size of synchronization groups.

Crossbars Crossbars route data streams between I/O nodes and PEA columns, and vice versa. On the input side, nodes can be partitioned into *input streams* through template parameters (`n_is`), each feeding one input crossbar set. Symmetrically, on the output side, PEA columns can be gathered into *output streams*

(`n_os`), each feeding one output crossbar set. Template parameters can be exploited to define the fan-in (`n_dma_ch_per_is`, `n_pea_dout_per_is`) and fan-out size (`n_pea_din_per_is`, `n_dma_ch_per_os`) of input (`dma_ch_plc_is`) and output (`pea_dout_plc_os`) crossbar sets, as well as the composition of input and output streams, thus defining crossbars' complexity.

More precisely, each input stream is defined as a subset of input nodes, and is connected to an input crossbar set with a template-configurable number of outputs. These outputs form the set of inputs delivered to the PEA columns. If an input stream contains a single node, the corresponding connection can be realized as a direct node-to-column link without generating a crossbar. Symmetrically, PEA column outputs can be grouped into output streams. Each output stream feeds an output crossbar set with a template-configurable number of outputs, which then connect to the output nodes and ultimately to the DMA channels. As on the input side, crossbars can be omitted to obtain direct column-to-node connections. In conclusion, template-based input and output crossbars may provide full connectivity, restricted connectivity within smaller groups, or direct wiring, depending on template settings. The actual mux selectors of the instantiated crossbars are programmed at run time through DSMU configuration registers (`is_sel`, `os_sel`).

Ready/Valid Interface towards PEA and DMA To comply with the ready/valid protocol at the DSMU-PEA boundary, the DSMU provides each input delivered to the PEA with a valid bit, and it exposes ready ports that inform about its ability to accept new PEA results. Since the DSMU pops an input node FIFO only when ① the FIFO is non-empty and ② the PEA is ready to consume values, the input valid can be directly identified with the DSMU pop strobe for that FIFO. On the output side, the DSMU identifies as *ready* whenever it can accept results into the output FIFO nodes. On the other hand, it becomes *stall* if output FIFOs are full and cannot accept additional data. In particular, the DSMU exposes one ready port for each output node, which can either be used individually or combined (e.g., AND-reduced) to represent an aggregate readiness condition.

Regarding the interaction with DMA channels, the DSMU exposes an interface based on standard FIFO-style handshake signals. This interface complies with the one already described in section 5.3, since it matches the one adopted by the X-HEEP DMA. Nonetheless, since it follows a simple FIFO-based handshake protocol, it is

not tied to X-HEEP, and it can be easily exploited for interfacing with other DMA subsystems [61] or AGUs.

DSMU Configuration Registers

Down-Counters

dma_rnw	For each channel i, selects whether the 32-bit down-counter decrements on pushes into the input node (DMA $\rightarrow$ DSMU) or on pushes into the output node (PEA $\rightarrow$ DSMU).
trans_size_dma_ch	Initial value of the 32-bit down-counter.
trans_s_size_dma_ch	Initial value of the 16-bit synchronization down-counter for input node i.

Crossbars Selectors

is_sel	Selectors for input crossbar sets.
os_sel	Selectors for output crossbar sets.

Synchronization & Status

is_synch_groups	Sets the size of the synchronization group.
sync_dma_ch_trans	For each input node i, enables synchronization with the other nodes in its group.
cols_grouping	Controls how output nodes ready flags are combined to generate the DSMU ready signals toward the PEA.

Fig. 5.4 Configuration registers of Rogue used by the DSMU.

DSMU Template Parameters

Nodes

n_nodes	number of I/O nodes.
n_is	number of input streams, i.e. groups of input nodes sharing the same input crossbar set.
n_os	number of output streams, i.e. groups of PEA columns sharing the same output crossbar set.

Input & Output Crossbars

n_dma_ch_per_is	number of input nodes in each input stream.
dma_ch_plc_is	list of **n_is** elements, in which element *i* lists the nodes connected to stream *i*.
n_pea_din_per_is	number of outputs of each input crossbar set (PEA input lanes driven by that input stream).
n_pea_dout_per_os	number of columns in each output stream.
pea_dout_plc_os	list of **n_os** elements, in which element *i* lists the columns connected to stream *i*.
n_dma_ch_per_os	number of outputs of each output crossbar set (output nodes driven by that output stream).

Synchronization

is_sync_plc_size	maximum size of synchronization groups.
is_sync_plc_g2	for each input node *i*, the input node synchronizable with *i* when selecting a 2-node group.
is_sync_plc_g3	for each input node *i*, the two input nodes synchronizable with *i* when selecting a 3-node group.
is_sync_plc_g4	for each input node *i*, the three input nodes synchronizable with *i* when selecting a 4-node group.

Fig. 5.5 Template parameters to define the internal structure of the DSMU.

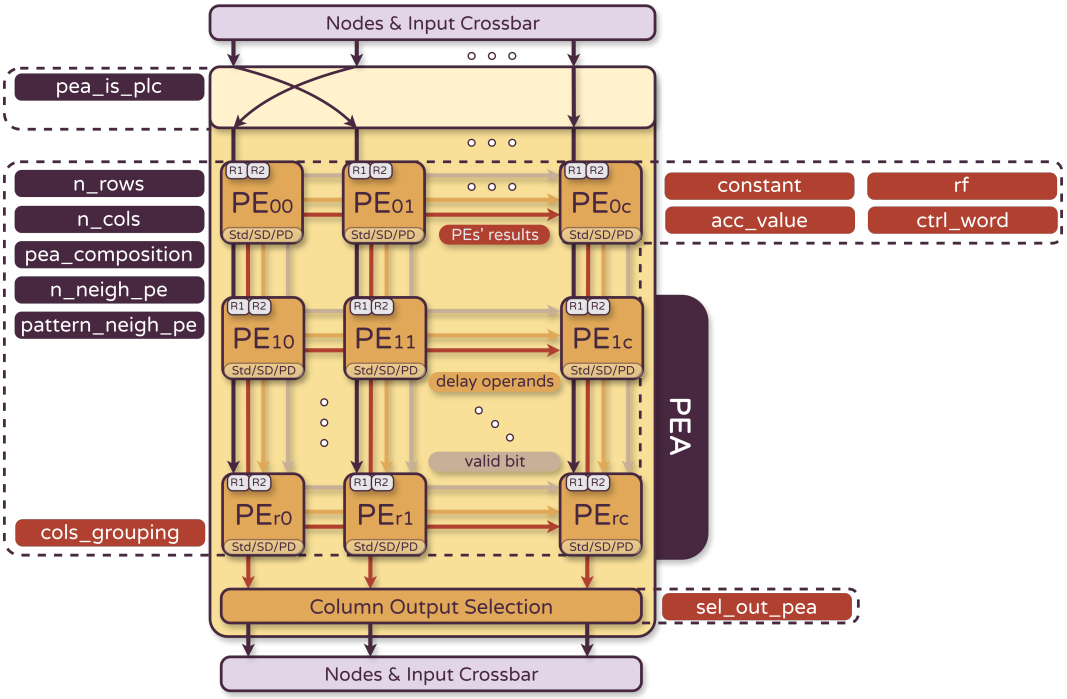


Fig. 5.6 Rogue's PEA template-based structure, along with template parameters and configuration registers identified with the same "section" titles used in Figure 5.8 and Figure 5.7, respectively.

5.5.2 Processing Element Array

The PEA is Rogue's computational core. Differently from standard GeMM-oriented CGRAs, Rogue's PEA is tailored for streaming computation of integer approximations of activation functions by implementing, within its PEs, the dedicated ISA described in section 5.4. Nevertheless, since the ISA also includes instructions involving multiply, accumulate and shifting operations, the same PEA can also execute more conventional GeMM workloads.

This section presents the PEA with a top-down approach. First, a description of array-level aspects is provided, including template-based generation of array sizes, PE placement and PEs connectivity schemes. After that, the discussion deals with how streaming execution is sustained, focusing on the ready/valid protocol used between PEs and at the DSMU boundary. Finally, going deeper in the hierarchy, the PE microarchitecture is presented, detailing how the proposed ISA is implemented, and PEs compliance with the ready/valid protocol.

Template-Based PEA Structure & Connectivity

Rogue uses template-based generation for the PEA hardware description. Template parameters can be used to define ① the array dimensions (i.e. the number of rows and columns), ② the interconnect topology, and ③ the type of PE instantiated at each position. As a rule of thumb, *setting the number of columns equal to the number of DSMU I/O nodes, and in turn to the number of DMA channels, is convenient*. As a matter of fact, it aligns PEA columns with load/store streams, simplifies mapping, and facilitates loop unrolling across multiple streams. Although convenient, this is not a hard constraint on the design template parameters used to size the PEA (`n_rows` and `n_cols`).

As regards connectivity, the interconnect template supports any topology that is a subset of nearest-neighbor links along the horizontal, vertical, and diagonal directions. Therefore, this includes the common *N2N* topology, which is also used in the Rogue implementations discussed in later sections. Template parameters can be used to set the number of neighbors for each PE (`n_neigh_pe`) and the connectivity pattern (`pattern_neigh_pe`). Besides neighbor connections, PEs also receive operands from the DSMU. For each PEA column, a template parameter (`pea_is_plc`) selects which outputs of the DSMU *input crossbar* are exposed to the PEs in that column. In other words, the parameter selects which crossbar outputs to feed to each column's PEs, it does *not* select directly among the input nodes. Although no upper limit is set to the number of inputs PEs can receive from DSMU, increasing the number of DSMU-fed inputs increases the size of the input multiplexers inside PEs and the routing complexity within the PEA. Therefore, this choice has to be carefully taken, as it trades mapping flexibility for area. Furthermore, for each column, the PEA selects the values that form the output streams toward the DSMU output crossbar by multiplexing among the results produced by all PEs in that column. The selectors of these muxes are set through configuration registers (`sel_out_pea`).

Finally, inter-PE connectivity also includes an additional **operand-forwarding mechanism**, whose introduction has been motivated by the algorithmic structure of integer activation kernels, although its exploitation can be useful more broadly. Integer approximation of activation functions may exhibit, within their loop body, *intra-iteration dependencies* that would otherwise reduce the applicability and efficacy of streaming computation. Therefore, to overcome this possible issue, each PE receives operands *delayed* (also referred to as **delay-operands**) by their neighbor

PEs, in addition to their outputs. This forwarding mechanism supports operand reuse across instructions while preserving the streaming model by solving data dependencies.

In addition, template parameters can be exploited to build heterogeneous arrays. As described in section 5.5.2, three PE variants are available. Exploiting the template-based PEA generation, a placement parameter (`pea_composition`) can select which PE variant is instantiated at each row i and column j.

Ready/Valid Synchronization Protocol

To sustain streaming execution, the PEA relies on validity bits to be exchanged together with data between neighboring PEs, and global status signals that control whether computation and data movement can proceed.

As in the DSMU, PEA can assume a *ready* or a *stall* status. Furthermore, symmetrically to DSMU, the PEA *ready*/*stall* status can be *partitioned* across columns through a configuration register. This means that multiple partitions of the PEA can be identified, each with its own status. In this context, three cases are supported: ① a single global status is computed by AND-reducing all DSMU ready signals and all PE stall signals; ② the array is split in half column-wise, identifying two partitions each with its own status computed by AND-reduction over the corresponding DSMU ready and PE stall signals; ③ one status per column is computed, i.e., N_{columns} independent AND-reductions. This partitioning mirrors the DSMU mechanism and is useful when independent computations are mapped onto different column groups, as happens in loop-unrolled kernels, and it can be configured using a configuration register (`cols_grouping`).

As regards execution, when a partition is in the *ready* status, PEs execute, operands propagate through the array, and data is exchanged with the DSMU. When in *stall* status, both execution and data flow stop and remain frozen until the stall condition is cleared. The PEA enters *stall* status either because a PE is executing a multi-cycle operation (e.g., division-related instructions in the proposed ISA) or because the DSMU is in the *stall* status. It is important to point out that DSMU and PEA are tightly coupled as regards status. They always share the same status, and execution resumes when the multi-cycle operation completes or when the DSMU returns to *ready*.

As regards synchronization among PEs and at the DSMU boundary, validity bits are employed. Each output of DSMU's input crossbars is accompanied with a valid bit, as discussed in subsection 5.5.1. Similarly, each PE's output has its own valid bit. Therefore, PEs know if their inputs are valid, and it performs computation whenever they are both valid and the PEA partition in which they reside is in the *ready* status.

Processing Element Architecture

Rogue's PEs enable integer approximations of activation functions by implementing the proposed ISA. In this section, the main features of the internal structure of PEs are discussed, focusing on ① the implementation of the ISA within the different types of FUs, ② local storage, input operand selection and delay operand selection, and ③ compliance with the ready/valid synchronization protocol.

As mentioned in previous paragraphs, heterogeneous PEAs can be generated having different area performance trade-offs. As a matter of fact, Rogue's PEA can exploit *three PE variants*, which mainly differ in the set of supported ISA instructions and in the implementation of division. The variants are described in the following.

1. **Standard PEs** implement `S` and `C` instructions that *do not include* division. Their FU includes the following operators: one 32-bit adder/subtractor, one 32-bit multiplier, one 32-bit right barrel shifter, and a dedicated 32-bit incrementer that only performs $+1$. The adder/subtractor is used for instructions involving `ADD`, `SUB`, `ABS`, and comparisons. The multiplier is employed for `MUL`-based operations. As for the shifter, it is used for all shift-based instructions, including the ones requiring a left shift, which is implemented through a proper bit-reversal strategy. The incrementer is exploited for the absolute value calculation in `ABSMIN` and, when needed, in `SGNCSUB`. The `ADD` instruction can also be used to implement accumulations. In this case, the incrementer is used to count how many elements have been accumulated, incrementing on each addition. The target accumulation length is programmed through a configuration register (`acc_value`).

2. **Serial-Division PEs** implement the full ISA by extending the Standard PE with a *serial divider*. The divider follows a *non-restoring* division algorithm and is radix-configurable. Hence, a radix r divider produces $\log_2(r)$ quotient bits per

cycle using $\log_2(r)$ subtractors, resulting in a latency of $\frac{32}{\log_2(r)}$ cycles. In the Rogue instances discussed later, $r = 4$, hence the divider uses 4 subtractors and completes in 8 cycles. Additionally, the divider produces both quotient and remainder. As for Standard PEs, the incrementer is also used for absolute-value handling in `ABSDIV` and `ABSREM`.

3. **Pipelined-Division PEs** also implement the full ISA, but use a *pipelined divider*. The divider uses 32 subtractors and, once filled, delivers one result per cycle. Pipeline registers are inserted every 4 subtractors to limit the critical path. Aside from the divider implementation, these PEs match the *Serial-Division* variant in terms of instruction support and FU usage.

Table 5.1 reports instruction latencies for the implemented PEs. Single-operation instructions complete in one cycle. On the other hand, most multi-operation instructions complete in two cycles by splitting the operations across two cycles, so that the critical path contains only one operator. Despite the two-cycle latency, these instructions do not require global stalls. As a matter of fact, since the involved operations use distinct operators, the PE can accept new inputs every cycle and sustain pipelined execution. As regards division-based instructions, they inherit the divider latency. For *Serial-Division* PEs, this latency is $\frac{32}{\log_2(r)}$, and a new result is valid each $\frac{32}{\log_2(r)}$ cycles. On the contrary, *Pipelined-Division* PEs expose the same per-result latency, but achieve 1 Output-per-Cycle (OpC) after the pipeline is filled.

Each PE is programmed through a 32-bit configuration register. It includes the 5-bit opcode field defined by the ISA and additional fields used for operand selection. In particular, the register contains the two 4-bit fields used by the ISA to select the FU source operands, denoted as a and b in Table 5.1. Each source operand can be selected from:

1. **Neighbor results**: up to 8 inputs, depending on the chosen PEA connectivity.
2. **DSMU inputs**: selected outputs of the DSMU input crossbar delivered to the PE.
3. **Constant operand**: it is used as the ISA operand C in Table 5.1, and it is implemented as a 32-bit configuration register accessible from the external world. Using C reduces routing pressure because it avoids distributing an

additional operand through the PEA. Other than acting as the constant in `C` instructions, it can also be used as operand in `S` ones.

4. **One-entry register file**: a 32-bit register (`rf`) that can be written by software and can optionally capture one DSMU input for later reuse. A dedicated configuration bit (in (`ctrl_word`)) enables capturing of the input coming from the DSMU. In case the PE receives more than one input from the DSMU, the first is the one that can be sampled.

5. **Local feedback**: the PE's own result, used for accumulation and short local recurrences.

6. **Delay-operand**: one value selected among the delayed values forwarded by neighbors (up to 8). The selection is controlled by a field (up to 3 bits) in a PE configuration register (`ctrl_word`).

Since neighbor connectivity is a template parameter, the width of the selection fields depends on the instantiated topology (i.e., fewer neighbor inputs reduce the required selector bits).

On the output side, the PE produces both the FU result and one delay-operand, each with an associated valid bit. The forwarded operand is selected via a configuration register among four options: the previously selected delayed operand, either of the two FU input operands, or the FU result. For division-based instructions, when forwarding the FU result, the PE can forward either quotient or remainder and keep the other as the main result, depending on the programmed choice. This is particularly useful in integer approximations of activations to overcome long data dependency paths. In addition, it is useful to specify that Rogue's PEs apply the same latency of the implemented instruction to its delay-operand, if it is configured to delay an operand.

To support the ready/valid mechanism described at array level, PEs also expose a *stall* signal to the PEA. A stall is required only for multi-cycle operations that cannot be internally pipelined without freezing inputs. In the implemented PE designs, this only applies to serial-division instructions in *Serial-Division* PEs. Whenever such an instruction is issued, the PE asserts *stall*, bringing the PEA to *stall* and stopping data flow while the divider runs. After $\frac{32}{\log_2(r)}$ cycles, the division completes, *stall* is deasserted, and the PEA returns to *ready*.

PEA and PE Configuration Registers

PEA

sel_out_pea	Per-column selector of the PE result forwarded to the DSMU output crossbar.
cols_grouping	Selects the partitioning scheme for the PEA for *ready/stall* status assignment.

PE

acc_value	Contains the accumulation length if an accumulation must be performed
constant	Contains constant operand C, not writable by the PE.
rf	Single-entry register file to locally retain one value, writable by the PE.
ctrl_word	PE control word, including the instruction opcode, FU input selectors, the delay operand selection and DSMU sampling in RF.

Fig. 5.7 Configuration registers of Rogue used by the PEA and internally to PEs.

PEA Template Parameters

Sizing

n_rows	Number of PEA rows.
n_cols	Number of PEA columns.

Composition

pea_composition	list where entry i selects the PE type instantiated at position $row = \lfloor i/C \rfloor$ and $column = i \bmod C$, with C the number of PEA columns

Connectivity

n_neigh_pe	Number of neighbor PEs considered by each PE for inter-PE communication.
pattern_neigh_pe	Neighborhood pattern: list of neighbor positions defining, for each PE, which surrounding PEs are reachable.
pea_is_plc	Input-stream for each PEA column, specifies which DSMU input-crossbar outputs are provided to the PEs in that column.

Fig. 5.8 Template parameters to define the structure of the PEA.

5.6 Workloads Mapping & Execution Model

Although some characteristics have been mentioned throughout previous sections, in this section, Rogue's streaming execution model is discussed, as well as the constraints it imposes on the kernels mapping process. In addition, the mapping process is also described. Rogue's streaming execution model is mainly based on two principles, which translate into **two main mapping constraints**, both discussed in the following.

1. **Synchronization of input streams in the DSMU**. Input streams produced by DMA *load channels* (or by external AGUs) must be synchronized at the DSMU level using the down-counter mechanism described in subsection 5.5.1. If Rogue operates on a single input stream, no synchronization is required. Synchronization is also unnecessary when multiple streams are *independent*, i.e., they do not participate in the same DFG. This is the case, for example, when different loop-unrolled instances are mapped onto different PEA partitions, as in the case of unrolled activation workloads in section 5.7. Instead, when two or more streams feed the same computation node on a PE and, therefore, are *data-dependent*, they must be synchronized by the DSMU so that the PEA observes correctly aligned values at its inputs. As a consequence, the PEA always operates on streams that are already aligned.

2. **Wait-free PE execution**. As discussed in section 5.5.2, a PE starts computations only when its required input operands are asserted as valid. As a consequence, it produces an output tagged with a valid bit. However, if only one operand is valid, the PE *does not wait* for the missing one. This mechanism is safe because operand alignment is enforced upstream by the DSMU: when a PE consumes a pair of operands, they are expected to become valid in the intended cycle. This makes the intra-PE synchronization lightweight and limits PE-induced stalls, which are only needed for multi-cycle instructions (e.g., serial division).

   Nonetheless, this choice imposes a mapping constraint. *Considering a DFG of a workload to be mapped on Rogue, producer and consumer operations must be placed such that, considering instruction latencies, operands arrive in the same cycle at the consumer PE input*. In other words, mapping must be **latency-consistent**. Therefore, along each path from the DFG sources to any node, the cumulative delay must match across *re-converging paths*. If two branches feeding the same operation have different latency, the mapper must compensate the difference by inserting additional delay on the faster branch (e.g., through explicit forwarding mechanisms supported by the PEA) or by choosing a placement/routing that naturally matches timing. Otherwise, the consumer PE would see one operand valid and the other invalid in a given cycle, thus skipping the computation for that cycle, effectively dropping data and breaking correctness.

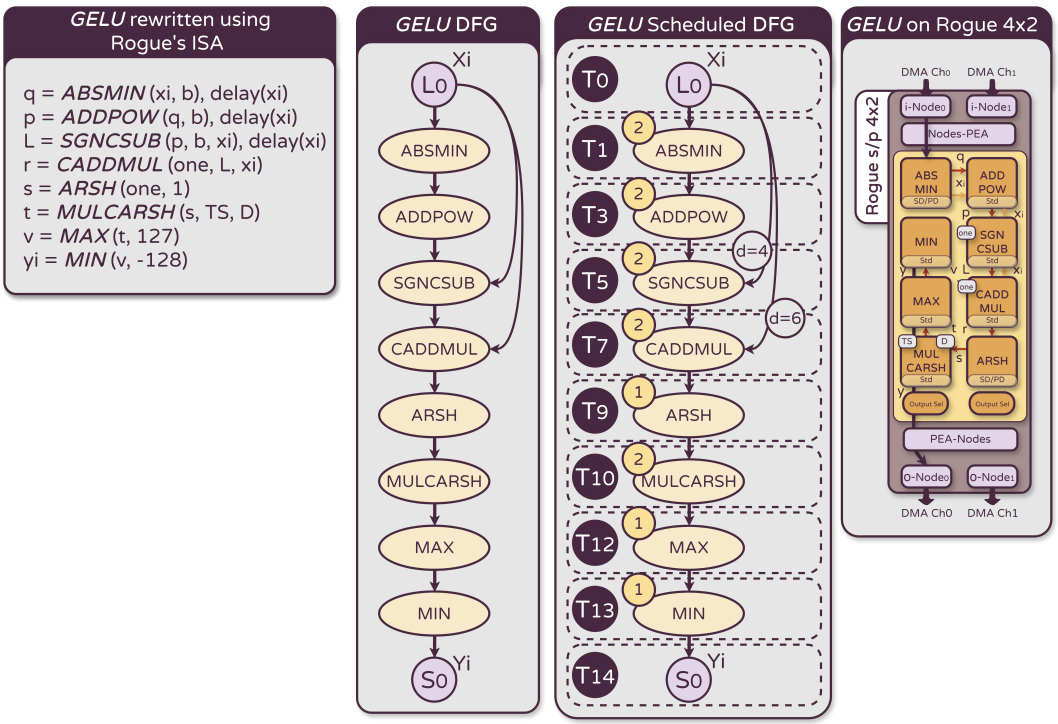


Fig. 5.9 Rewriting of *GELU* using Rogue's ISA, its DFG and scheduled DFG, along with the relative mapping onto a 4×2-PEA Rogue.

Considering the described characteristics, the main constraints the drive the workload mapping process onto Rogue are the following:

1. Access **Resource limits:** the number of load/store operations cannot exceed the total number of available DMA channels (or LSUs).
2. Access **Load stream synchronization:** data-dependent load streams, i.e. load streams that feed re-convergent DFG paths, must be synchronized at the DSMU level using the related configuration registers.
3. Execute **Resource limits:** the number of data-related operations per cycle cannot exceed the number of available total PEs.
4. Execute **PE connectivity:** compute nodes must be mapped onto PEs such that operands can be routed through the selected topology and through the DSMU→PEA inputs exposed to each column.
5. Execute **Latency-consistency:** the cumulative delay must match across *re-converging paths*.

Algorithm 3 i-BERT's GELU algorithm considered for mapping onto Rogue.

Require: x: input `int8` tensor
Require: y: output `int8` tensor
Require: N: number of input sequence elements
Require: b: quantized negative offset/clipping constant of the i-GELU polynomial
Require: one: integer representation of the constant 1 in the polynomial scale
Require: TS: output requantization multiplier
Require: D: output requantization right shift
1: **for** $i \leftarrow 0$ **to** $N-1$ **do**
2: $s \leftarrow \text{sign}(x[i])$
3: $a \leftarrow s \cdot x[i]$
4: $q \leftarrow \min(a, -b)$
5: $d \leftarrow q + b$
6: $L \leftarrow s \cdot (one - d^2)$
7: $z \leftarrow (x[i] \cdot (one + L)) >> 1$
8: $y[i] \leftarrow \max(\min((z \cdot TS) >> D,\ 127),\ -128)$
9: **end for**

Taking into account the discussed constraints, the kernel mapping methodology for Rogue is described in the following, and it is mainly composed of three steps, discussed in the following. For better understanding, an example is also provided, following the methodology employed for mapping i-BERT's *GELU* kernel onto a Rogue implementation having a 4×2 PEA and integrated within 2 DMA channels.

① **C code implementation & ISA-based rewriting.** This step consists in **implementing a `C` version of the workload to be deployed on Rogue**, along with the selected Unroll Factor (UF) on the target loop. As it happens for Mage, this operation allows to verify if the kernel can be effectively mapped onto Rogue by checking that:

1. the kernel is a **SCoP workload**, hence array indices and any static-control predicates are affine functions of loop IVs.
2. the **operators are available** based on the employed set of Rogue's PEs.
3. the **array access patterns are supported** by the used LSUs or DMA channels.

Furthermore, *this step also involves rewriting the workloads using the ISA implemented by Rogue*, in order to fully benefit from it and employ as few PEs as possible for mapping the kernel. This process is also useful to verify constraint 1 and 3 by

counting the number of arithmetic and load/store operations, as they must all concurrently fit both available DMA channels and PEA, respectively. The `C` implementation of *GELU* kernel is reported in Algorithm 3, and its rewriting according to Rogue's ISA is illustrated in Figure 5.9. The integer-only i-BERT's GELU kernel is taken from the `C` implementation that can be found at https://github.com/pulp-platform/pulp-transformer/blob/main/Kernel/src/gelu.c.

② **DFG extraction.** Based on the `C` implementation and its rewriting considering Rogue's ISA, **the workload is translated into a DFG representation**. This operation is crucial to explicitly identify the **data dependencies** among load/store operations and arithmetic ones. Therefore, it is important to identify re-convergent path on which constraint 2 and 5 must be applied.

③ **Scheduling.** This step mainly consists in **assigning each operation in the DFG with a time instant in which it executes.** In this context, it is important to highlight that, due to the intrinsic characteristics of Rogue's computational model and by Rogue's construction, neither load/store operations nor arithmetic one are "fired" at specific time instants on DMA channels and PEs, respectively. However, performing the scheduling process is mainly useful to explicitly identify the timing of re-convergent paths, and inserting, if necessary, delays to equalize the different timing of converging branches. As for the scheduling process, it follows an **As Soon As Possible** (*ASAP*) methodology. Therefore, load operations must all be assigned $T_{load} = 0$, which is compliant with the fact that DMA channels are started all together. Then, arithmetic operations are assigned a time slot based on the latencies of previous operations on which they depend on, while also matching latencies on re-convergent paths. Lastly, store operations are scheduled.

④ **Mapping.** This step mainly involves **assigning each load/store operation with an DMA channel, and each arithmetic operation with a PE**. Therefore, constraint 4 can be checked, ensuring that operations can be mapped onto PEs such that inputs, intermediate results and outputs can flow in the PEA and between PEA and DSMU. In order to perform the mapping onto Rogue, data-dependent operation must be placed on neighbor PEs, as well as load/store operations placed such that they can provide inputs of retrieve outputs from accessible PEs. In addition, the mapping process must also take that into account delay-operands, in case they are exploited. The mapping of *GELU* onto a 4×2 Rogue implementation (*s4x2_2* or *p4x2_2* discussed in subsection 5.7.2) is reported in Figure 5.9

In conclusion, in order to be executed on Rogue, workloads have to be mapped onto the CGRA considering the mapping constraints imposed by Rogue's streaming execution model. However, no automatic mapper is currently available for Rogue, therefore all kernels discussed in section 5.7 have been manually mapped. Once a kernel is mapped, configuration registers have to be set accordingly to the found mapping. After that, DMA channels or AGUs can be started and kernel execution takes place. Once the DSMU receives and synchronizes the input streams, each PEA column receives data from an input node. Then, values propagate through the array and results are pushed into output FIFOs, before being written back to memory by store channels.

5.7 Results and Discussion

In this section, Rogue is ① implemented in multiple homogeneous and heterogeneous variants, exploiting its template-based design, ② integrated into the X-HEEP SoC as an external DMA-coupled streaming accelerator, and ③ evaluated on several integer approximations of activation functions as well as on a GeMM workload.

5.7.1 Rogue Implementations

Leveraging Rogue's template-based design, **seven Rogue variants** have been generated and analyzed. These implementations differ in PEA size, in the PE variants instantiated across the array, and in the interconnections complexity between the DSMU input crossbar and the PEA. This set of Rogue implementations is used to explore area-performance trade-offs and to highlight how different configurations affect kernel mapping. The main features of the studied implementations are summarized below.

The variant names includes three fields. The first field is a letter that specifies the PEA composition, with*p* and *s* indicating heterogeneous arrays that include *Pipelined-Division* or *Serial-Division* PEs, respectively, while *g* indicating a homogeneous array with only *Standard* PEs. The second field is a $R \times C$ sequence that indicates the PEA size, where C is the number of columns and R is the number of rows. In the DSMU, the number of I/O nodes is also set to C. The final field is a number that captures two template choices, i.e., the maximum size of synchronizable input-

stream groups in the DSMU, and the number of inputs from the DSMU side exposed to the PEs of each column.

As for the DSMU↔PEA connection, none of the discussed variants instantiates an input crossbar, so input nodes are directly connected to the PEA. On the output side, all variants use 2×2 crossbars that connect columns pairs $(0,1)$ and $(2,3)$ to output-node pairs $(0,1)$ and $(2,3)$, respectively. The main characteristics of the analyzed Rogue variant are reported in the following.

- **p4x2_2** has a 4×2 PEA with *Pipelined-Division* PEs at positions $(0,0)$ and $(3,1)$, while all other PEs are *Standard*. At the DSMU level, both nodes (0 and 1) can be synchronized with each other. Also, PEs in columns 0 and 1 can select their inputs from nodes 0 and 1
- **p4x4_2** has a 4×4 PEA with *Pipelined-Division* PEs at positions $(0,0)$, $(0,3)$, $(3,1)$, $(3,3)$ while all other PEs are *Standard*. At the DSMU level, synchronizable input nodes groups are limited to pairs $(0,1)$ and $(2,3)$. PEs in columns 0 and 1 can select their inputs from nodes 0 and 1, and PEs in columns 2 and 3 can select their inputs from nodes 2 and 3.
- **p4x2_2** has a 4×2 PEA with *Serial-Division* PEs at positions $(0,0)$ and $(3,1)$, while all other PEs are *Standard*. At the DSMU level, both nodes (0 and 1) can be synchronized with each other. Also, PEs in columns 0 and 1 can select their inputs from nodes 0 and 1
- **p4x4_2** has a 4×4 PEA with *Serial-Division* PEs at positions $(0,0)$, $(0,3)$, $(3,1)$, $(3,3)$ while all other PEs are *Standard*. At the DSMU level, synchronizable input nodes groups are limited to pairs $(0,1)$ and $(2,3)$. PEs in columns 0 and 1 can select their inputs from nodes 0 and 1, and PEs in columns 2 and 3 can select their inputs from nodes 2 and 3.
- **g2x2_2** has a 2×2 PEA with only *Standard* PEs. The DSMU supports synchronization between input nodes 0 and 1. Each column can select up to 2 DSMU-fed inputs.
- **g2x4_2** has a 2×4 PEA with only *Standard* PEs. The DSMU supports pairwise synchronization. PEs in columns 0 and 1 can select their inputs from nodes 0 and 1, and PEs in columns 2 and 3 can select their inputs from nodes 2 and 3

- **g2x4_4** has a 2×4 PEA with only *Standard* PEs. At the DSMU level, synchronizable input nodes groups can include up to four nodes $(0,1,2,3)$. PEs in each column can select any of the inputs coming from the 4 input nodes.

In this design-space exploration, heterogeneous Rogue variants are used to study area-energy-performance trade-offs on integer activation kernels, while homogeneous variants are mainly used for GeMM kernels. As for heterogeneous implementations, the choice behind PEA sizes are the following. The 4×2 configuration with the selected division-PE placement is the smallest array that supports all evaluated integer-approximation kernels, while the 4×4 configurations (*s4x4* and *p4x4*) are introduced to increase parallelism. For the homogeneous designs, the explored design knobs mainly target input handling, i.e. the maximum size of synchronizable input node groups, as well as the number of DSMU inputs exposed to the PEs. Together with the different PEA sizes, these parameters enable different degrees of loop unrolling for the GeMM workload evaluated in the following sections.

Although the variants considered here cover only a small subset of the possible configurations, they are a good set of Rogue implementations to highlight the main architectural trade-offs and to validate Rogue's mapping and streaming execution model across both integer activation kernels and GeMM workloads.

5.7.2 Experimental Setup

In order to evaluate Rogue template-generated implementations, they have been integrated into an X-HEEP SoC as external DMA-coupled streaming accelerators. In this environment, the Rogue variant is attached, through the DMA extension interface, to X-HEEP's DMA subsystem, and it operates directly on the DMA data stream. The employed X-HEEP-based host system encompasses the OpenHW Foundation CV32E40P RISC-V core implementing the *XCVPULP* ISA extensions, 8 32-KiB memory banks supporting interleaved layout of data across 4 banks, and a DMA subsystem. The DMA subsystem features two channels sharing two master ports on the bus in the 4×2 PEA variants. On the other hand, in the 4×4 PEA configurations, the DMA has four channels sharing four master ports. The explored design space includes heterogeneous *s* and *p* variants, used to study integer approximations of activation kernels, and homogeneous *g* variants, mainly considered for *GeMM*-like workloads.

Table 5.2 Workloads used to evaluate Rogue on integer approximations of activations and GeMM kernels.

Workload	Description
ReLU	computes the Rectified Linear Unit activation ($y_i = \max(x_i, 0)$) with *a single loop*.
GELU	computes the i-BERT approximation of the Gaussian Linear Error Unit through *a single loop*.
Tanh	computes an integer approximation of the hyperbolic tangent activation based on i-BERT's exponential using *a single loop*.
Sigmoid	computes an integer approximation of the Sigmoid activation based on i-BERT's exponential using *a single loop*.
SiLU	computes the Sigmoid Linear Error activation ($y_i = x_i \cdot \sigma(x_i)$), based on i-BERT's exponential through *a single loop*.
Softmax	computes the i-BERT approximation of Softmax, combining max search, accumulation of the normalization term, and final normalization, using *three subsequent loops*.
ITA-Max	computes an 8-bit integer approximation of Softmax proposed in [74], through *two successive loops*.
Matmul	computes matrix multiplication $C = AB$ using *three perfectly nested loops*.

s and *p* Rogue implementations and related X-HEEP systems have been synthesized with Synopsys Design Compiler targeting a commercial 16 nm technology node and operating at 330MHz under worst-case conditions. In addition, all variants (including *g* ones) have also been synthesized with Synopsys Design Compiler targeting a commercial 65 nm technology node and operating at 250MHz under worst-case conditions. Functional validation has been performed through post-synthesis simulations, while power and energy results have been obtained with Synopsys PrimeTime using switching activity annotation extracted from the waveforms gathered in *.vcd* files. While energy metrics are reported considering the 16 for nm technology *s* and *p* implementations, *g* Rogue variants have been analyzed on the 65 nm technology. Since Rogue does not currently rely on an automatic compilation flow, all the considered workloads have been manually mapped onto the considered variants, and the corresponding Rogue configurations and DMA transactions have been derived accordingly.

Table 5.3 For each **kernel** and related **UF**, one or multiple **DMA Transfer** may be needed. For each Rogue variant, the number of DMA channels used to implement the related workload on the specified UF in the indicated DMA transfer is indicated (in the related cell within the variant column). As for *matmul*, *N* is the number of rows of the first input matrix.

Kernel	UF	DMA Transfer	s4x2_2	s4x4_2	p4x2_2	p4x4_2	g2x2_2	g2x4_2	g2x4_4
ReLU	**2**	1	2	2	2	2	2^b	2^b	2^b
	4	1	-	-	4	4	-	4^b	4^b
GELU	**1**	1	1	1	1	1	-	-	-
	2	1	-	-	2	2	-	-	-
Sigmoid	**1**	1	1	1	1	1	-	-	-
	2	1	-	-	2	2	-	-	-
Tanh	**1**	1	1	1	1	1	-	-	-
	2	1	-	-	2	2	-	-	-
Softmax	**1**	1	2	2	2	2	-	-	-
		2	2	2	2	2	-	-	-
		3	2	2	2	2	-	-	-
	2	1	4	4	4	4	-	-	-
		4	4	4	4	4	-	-	-
		3	4	4	4	4	-	-	-
ITA-Max	**1**	1	2	2	2	2	-	-	-
		2	2	2	2	2	-	-	-
	2	1	4	4	4	4	-	-	-
		2	4	4	4	4	-	-	-
SiLU	**1**	1	1	1	1	1	-	-	-
		2	2	2	2	2	-	-	-
	2	1	2	2	2	2	-	-	-
		2	4	4	4	4	-	-	-
Matmul	**1**	N	2^a	2^a	2^a	2^a	2	2	2
	2	$\frac{N}{2}$	-	3^a	-	3^a	-	4	3
	3	$\frac{N}{3}$	-	-	-	-	-	-	4

matmul is not evaluated on *s* and *p* Rogue variant in this thesis.
ReLU is not evaluated on *g* variants in this thesis.

The set of evaluated workloads, whose description is deepened in Table 5.2 includes seven integer approximations of activation functions, namely **Softmax**, **ITA-Max**, **GELU**, **ReLU**, **Sigmoid**, **Tanh**, and **SiLU**, together with **matmul** (matrix multiplication) as representative for GeMM kernel. As regards activations, the Rogue-accelerated *GELU* and *Softmax* are based on the integer approximations proposed in *i-BERT* [68], *ITA-Max* follows the 8-bit approximation introduced in ITA [74]. In addition, *Tanh*, *Sigmoid*, and *SiLU* rely on the same integer exponential formulation used in i-BERT. As these last three approximations have been proposed within the scope of this thesis, they have been also evaluated in terms of obtained

errors. As regards *Tanh*, *Sigmoid* and *SiLU*, they achieve MAE (Mean Absolute Error) and RMSE (Root Mean Squared Error) below 6.0×10^{-3} with respect to the exact functions over the interval $[-4,4]$. Activation experiments employ 8-bit input data, consistently with ITA-Max and with the i-BERT-based approximations, while intermediate computations are carried out on 32 bits. Nonetheless, Rogue also supports `int16` and `int32` computation.

As for the mapping on Rogue for the different integer activation kernels, they can be divided in two groups. **Single-transfer kernels** (*GeLU*, *ReLU*, *Sigmoid* and *Tanh*) have an algorithmic description that allows executing them on Rogue exploiting a single DMA transfer using one channel, with possible unrolling using two channels. These kernels are the most streaming-friendly and, therefore, more favorable for Rogue-based acceleration. On the other hand, **multi-transfer kernels** (*Softmax*, *ITA-Max* and *SiLU*) need multiple successive DMA transfers to be performed, as either their structure rely on multiple successive loops (*Softmax* and *ITA-Max*), or data dependencies in the single loop they exhibit are too stringent (*SiLU*). Although more execution steps are needed, the single steps are still streaming-friendly. In addition, they also can benefit from unrolling on the various needed transfers. Rogue variants are reported along with supported workloads and the relative UFs in Table 5.3, as well as the number of DMA channels employed for every transfer needed for the workload.

Furthermore, the homogeneous *g* variants are evaluated on *matmul* to analyze how PEA size and DSMU input-handling parameters affect the exploitable degree of loop unrolling on a *GeMM* workload. In this case, since *matmul* exploits three nested loops but X-HEEP DMA only supports 2D transfers, N successive transfers are needed to compute results, where N is the number of rows of matrix A considering the $A \times B$ matrix multiplication. Due to the absent support of X-HEEP's DMA for 3D and to its intrinsic algorithmic structure, this kernel is not directly favorable for Rogue, but still achieves good performance and gains compared to software execution. It is also worth noting that *s* and *p* Rogue implementations can execute *matmul* as well. However, in this analysis, *g* variants are considered as the target implementations for this kernel, since Rogue is being evaluated as a GeMM accelerator and *g* variants are expected to provide the best area and energy efficiency.

In the following sections, Rogue implementations are analyzed regarding ① area occupation and efficiency, ② achieved performance (i.e. throughput and speedup

compared to the CV32E40P CPU), and ③ energy efficiency (energy savings with respect to the CV32E40P CPU).

5.7.3 Area Evaluation - *s*, *p* & *g* Variants

Table 5.4 shows the area breakdown for all *s* and *p* Rogue implementations and related X-HEEP systems implemented in 16 nm technology at 330 MHz. In both 4×2 and 4×4 PEA scenarios, the Rogue *p* implementations occupy approximately 1.37× more area than their *s* counterparts, mainly due to the *pipelined-division* PE being 2.4× larger than the *serial-division* one. As regards 4×4 configurations, they occupy twice the area compared to 4×2 in both *s* and *p* configurations. However, they compensate the increased area occupation by enabling enhanced parallelization capabilities for all kernels. Furthermore, it is important to highlight that the area occupation of the DSMU and the configuration registers remains marginal, reaching up to 6% and 11%, respectively, in *s4x2*, and their contributions become less significant on 4×4 arrays in both *p* and *s* variants.

Table 5.4 Area breakdown of *S* and *P* Rogue variants and related X-HEEP systems ($\times 10^3$ μm^2) in 16 nm technology at 330 MHz.

	s4x2_2	*s4x2_2*	*s4x4_2*	*s4x4_2*	X-HEEP	
Conf. Regs.	1.9	1.9	3.7	3.7	15.4	CV32E40P
DSMU	1.1	1.1	2.2	2.2	259.7	Memory
PEA	14.0	20.4	28.6	40.9	5.0 (2 Ch.) 10.0 (4 Ch.)	DMA
Standard PEs	9.2	9.4	18.9	18.0	31.6	Other Periph.
Division PEs	4.8	11.0	9.7	22.8	70.1	Pad Ring
Total Area	17.0 (+4.5%)	23.3 (+6.2%)	34.5 (+9.0%)	46.7 (+12.2%)	376.8 (2 Ch.) 381.8 (4 Ch.)	Total Area

At the system level, Rogue's smallest variant, *s4x2*, occupies just 1.1× the CPU area, incurring only a 4.5% increase in the overall system area. With *p* variants and larger arrays, the overhead increases, and Rogue is 3× the size of the CPU in the *p4x4* configuration, reaching up to 12.2% system area increment. In addition to the contained system-level area overheads, Rogue *p* and *s* variant achieve significant throughput and energy improvements compared to the CV32E40P CPU, as discussed in subsection 5.7.4.

As regards Rogue *g* variants, they have been analyzed in 65 nm technology at 250 MHz. For completeness, also *s* and *p* variants have been analyzed in the same conditions as regards area occupation, and trends evaluated at 16 nm are confirmed. Results for both are reported in Table 5.5. Comparing equally-sized *s*-*p* and *g* PEAs (i.e., *s4x2_2* and *g2x4_2*), homogeneous *g* arrays are approximately 1.2× smaller compared to the *s* PEA, and up to 1.6× smaller compared to the *p* PEA. Among *g* Rogue variants, PEA, DSMU and configuration registers sizes naturally increases when considering 2×4 implementations. In addition, it is important to highlight that *supporting more flexible synchronization schemes incurs negligible area overheads in the DSMU* between *g2x4_2* and *g2x4_4*. Also, supporting more flexible interconnection schemes among PEA and DSMU incurs in slightly increased PEA area in the *g2x4_4* variant. Nonetheless, the system-level area overhead of *g* variants are even lower than *s* and *p* ones, ranging from just 2.8% up to 5.5%.

Table 5.5 Area breakdown of Rogue *s*, *p* and *g* variants and related X-HEEP systems ($\times 10^3$ μm^2) in 65 nm technology at 250 MHz.

g variants

	g2x2_2	*g2x4_2*	*g2x4_4*	X-HEEP	
Conf. Regs.	7.7	15.3	15.3	78.4	CV32E40P
DSMU	7.9	15.9	16.0	1864.5	Memory
PEA	54.5	107.4	111.5	36.4-72.6	DMA
				351.7	Other Periph.
Standard PEs	53.7	105.9	110.0	198.2	Pad Ring
Total Area	70.1 (+2.8%)	138.6 (+5.4%)	142.9 (+5.5%)	2529.3–2582.1	Total Area

s and p variants

	s4x2_2	*p4x2_2*	*s4x4_2*	*p4x4_2*	X-HEEP	
Conf. Regs.	13.7	13.8	27.4	27.4	78.4	CV32E40P
DSMU	7.9	8.0	15.8	16.0	1864.5	Memory
PEA	129.0	176.2	255.2	351.5	36.4-72.6	DMA
Division PEs	49.2	95.2	95.3	189.5	351.7	Other Periph.
Standard PEs	78.7	80.0	157.6	160.0	198.2	Pad Ring
Total Area	150.6 (+6.0%)	198.0 (+7.8%)	298.3 (+11.6%)	394.9 (+15.3%)	2524.5-2582.5	Total Area

5.7.4 Performance Evaluation - *s* & *p* Rogue Variants

In this section, *s* and *p* Rogue implementations are evaluated regarding throughput and speedup compared to execution on the CV32E40P implementing the *XCVPULP* ISA extension running compiler-generated code aware of the mentioned ISA.

① **Throughput.** For the *s* and *p* Rogue variants, throughput is defined as the number of results produced per cycle, i.e., *Outputs-per-Cycle* (OpC). This metric is chosen as it allows easier comparison with other accelerators for activation functions. Figure 5.10 reports the throughput scaling of all considered integer activation kernels as the input sequence length increases. For each kernel and UF, a horizontal line is also shown to indicate the corresponding *ideal throughput*. For single-transfer kernels, the ideal throughput directly matches the UF of the kernel (e.g., UF1 $\rightarrow$ 1 OpC). On the other hand, for multi-transfer kernels, the ideal throughput is computed as the average of the ideal throughput values of the individual transfers, since the unrolling factor applied on Rogue may differ across them.

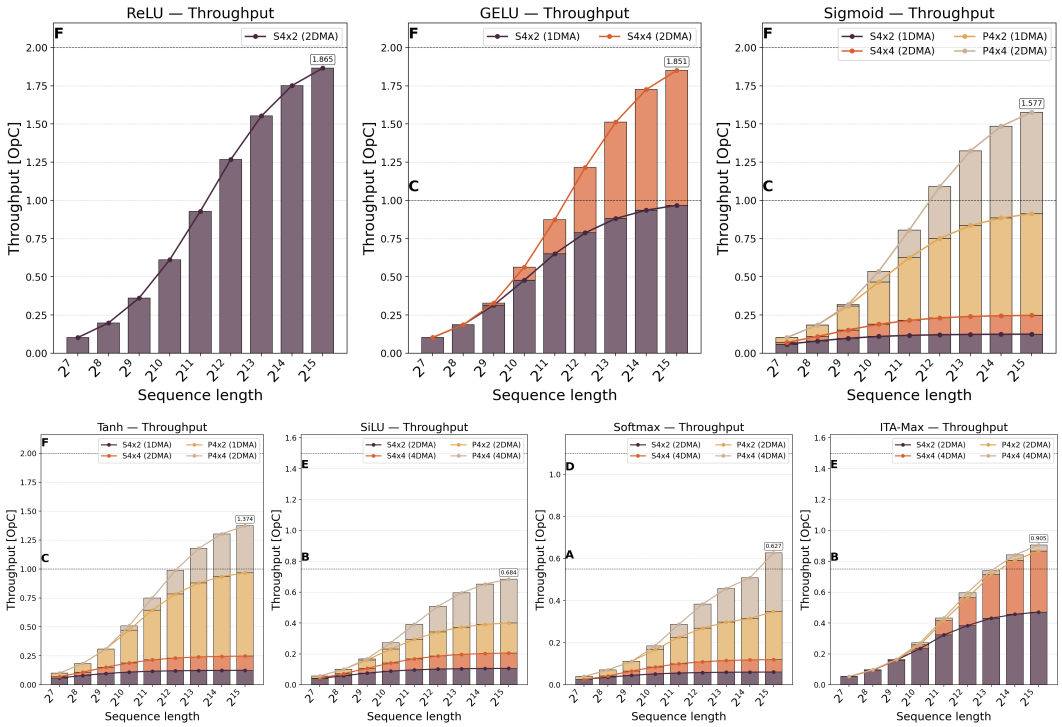


Fig. 5.10 Throughput scaling for *S* and *P* Rogue implementations over growing input sequence length.

All activation kernels exhibit the same overall trend, with **throughput increasing along with the input sequence length, and then gradually approaching a**

saturation plateau. Unlike the methodology adopted for Mage, Rogue and DMA configuration times are included in the throughput calculation. While such overhead can be effectively hidden in an end-to-end application for single-transfer workloads, multi-transfer kernels, namely *Softmax*, *ITA-Max*, and *SiLU*, necessarily require intermediate reconfiguration steps, although they can be limited. Therefore, for consistency, configuration cycles are accounted for in all activations. As a result, Rogue and DMA setup introduce a constant offset whose impact decreases as the input sequence becomes longer. In addition, pipeline filling contributes an additional fixed latency component, thus contributing to the same trend.

Although some kernels approach their ideal throughput on long input sequences, a residual gap is always present. This gap is mainly caused by two factors: **fixed configuration overhead** of Rogue and the DMA, and **memory access conflicts**. As already observed, the former is especially relevant for short sequences and becomes even more penalizing for multi-transfer activations, which require repeated reconfiguration between successive transactions. The latter, on the other hand, prevents the architectures from reaching the theoretical throughput even for long sequences, due to unfavorable alignments between DMA reads and writes targeting the same memory banks. In the following, the relative impact of these two effects is discussed for each kernel class.

For the division-free activations, namely *ReLU* and *GELU*, the throughput of *s* and *p* variants fully overlaps, hence only the *s* results are reported. *GELU* achieves nearly ideal performance, reaching 1.85 OpC on the 4×4 configuration in UF2 and about 0.97 OpC on the 4×2 one in UF1. In this case, configuration overhead is the main limiting factor, although its impact remains modest. By contrast, *ReLU* is the clearest example of degradation caused by memory access conflicts. Despite a theoretical throughput of 4 OpC on the 4×4 architecture in UF4, the measured value saturates around 2 OpC, showing that memory access conflicts become highly impactful when multiple DMA transfers are carried out concurrently.

For single-transfer division-based activations, the benefit of the pipelined divider is substantial. *p4x2_2* approaches ideal throughput on both *Sigmoid* and *Tanh*, reaching about 0.92 OpC and 0.97 OpC, respectively, in UF1. At the same time, *p4x4_2* delivers the highest absolute throughput, with 1.8 OpC on *Sigmoid* and 1.37 OpC on *Tanh* in UF2. By contrast, the *s* variants show much lower throughput,

around 0.13 OpC on 4×2 in UF1 and 0.25 OpC on 4×4 in UF2, confirming that the *non-pipelined divider is a major throughput bottleneck for these kernel*s.

As regards multi-transfer activations, throughput is naturally reduced because the available bandwidth must be shared across successive transactions. Since these kernels are also division-based, *p* variants again outperform *s* ones. The highest throughput is achieved by *p4x4*, which reaches 0.68 OpC on *SiLU*, 0.63 OpC on *Softmax*, and 0.91 OpC on *ITA-Max* at the corresponding maximum UFs. An important exception is *ITA-Max*, for which *p4x4_2* only slightly outperforms its *s* counterpart. This behavior is mainly due to the algorithmic structure of *ITA-Max*, whose approximation does not stress the divider as heavily as the i-BERT *Softmax* implementation.

Overall, the throughput results show that **Rogue achieves its highest efficiency on long and computation-dense workloads**. Moreover, 4×4 **variants consistently deliver higher throughput than** 4×2, as they enable enhanced loop unrolling capabilities. At the same time, ***p* variants significantly outperform *s* ones on division-based kernels**. Nevertheless, **the main limiting factors remain memory access conflicts**, especially when several DMA transfers are active concurrently, and, **reconfiguration overhead in multi-transfer workloads**. As regards conflicts, they can be reduced by manually arranging data in system memory to favor a more conflict-free layout or, at design time, by increasing the number of banks across which data can be interleaved, which is four in the considered X-HEEP-based systems. As regards reconfiguration overhead, its impact can be mitigated by cleverly "hiding" it within the execution flow of an end-to-end application.

① **Speedup.** Figure 5.11 reports the speedup achieved by the *s* and *p* Rogue variants over the execution on the CV32E40P CPU for all considered integer activation kernels. In this case, the CPU execute compiler-generated code aware of *XCVPULP* extensions, but no manual optimization is performed on the code. Across all workloads, speedup follows the same trend observed for throughput. As a result, it increases with the input sequence length and then gradually approaches a saturation plateau. For short sequences, the fixed overhead due to Rogue and DMA configuration affects execution time, thus limiting the advantage of hardware acceleration. As the sequence length increases, this constant cost is amortized, and the performance benefit of Rogue becomes more pronounced.

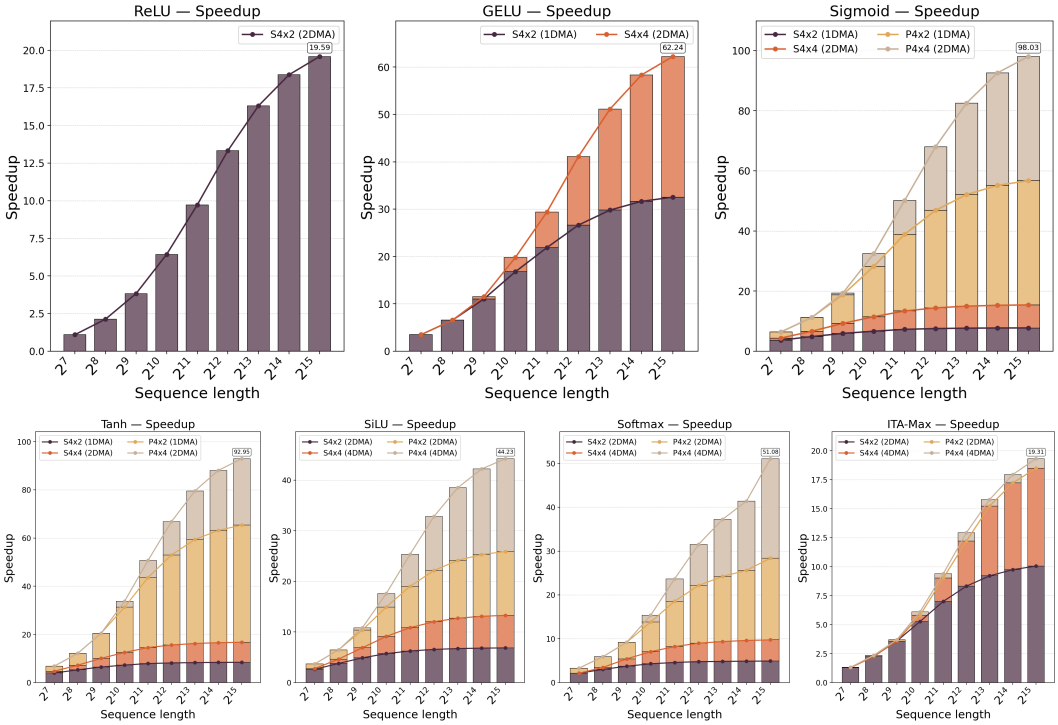


Fig. 5.11 Scaling of speedup over execution on the CV32E40P implementing the *XCVPULP* ISA extensions for *S* and *P* Rogue implementations over growing input sequence length.

In all cases, 4×4 configurations outperform their 4×2 counterparts, since the larger PEA allows a higher UFs to be applied and, therefore, a greater amount of exploitable parallelism. At the same time, **the gain over software execution also depends strongly on the computational complexity of the activation**. Therefore, kernels with a heavier arithmetic structure usually benefit more from acceleration, while lighter kernels are more affected by configuration overhead effects.

As regards division-free activations, namely *ReLU* and *GELU*, the speedup of *s* and *p* variants overlaps. In particular, *ReLU* reaches a maximum speedup of about 19.59×, while *GELU* scales up to 62.24× on the 4×4 configuration. This confirms that the speedup gain of Rogue compared to the CV32E40P CPU increases along with the complexity of the workload, with more compute-intensive kernels deriving a much larger benefit from Rogue acceleration once the fixed overhead is amortized.

A much clearer distinction between *s* and *p* variants can be noted for division-based activations. For *Sigmoid* and *Tanh*, the pipelined divider of *p* Rogue provides a greater advantage, enabling *P4x4_2* to achieve peak speedups of 98.03× and 92.95×, respectively, whereas the corresponding *s* configurations remain below 17×.

The obtained speedups on division-based kernels are mainly due to the superior throughput of Rogue's pipelined divider compared to the CV32E40P CPU's Count Leading Zero (CLZ) divider, whose latency can be up to 32 cycles. This confirms that division is the dominant bottleneck for these kernels, both in software and in the *s* variants, and that replacing the serial divider with a pipelined one substantially improves the acceleration that can be achieved.

For multi-transfer activations, significant speedups are achieved, although they are attenuated by the additional DMA transactions required to complete the computation. Also in this case, *p* variants obtain higher speedups than *s* counterparts on division-based workloads. In particular, *p4x4_2* reaches 51.08× on *Softmax* and 44.23× on *SiLU*. A notable exception is again represented by *ITA-Max*. Although it is also division-based, its formulation involves far fewer divisions than standard *Softmax*. As a result, *s* and *p* variants achieve very similar speedups, with a peak around 19×. In this case, the advantage of the pipelined divider is mitigated by the algorithmic structure of the kernel itself, as discussed above.

Overall, the speedup results confirm the ones obtained regarding throughput, showing that **Rogue is particularly effective on long and computationally intensive workloads**, with 4×4 variants providing higher gains than 4×2 due to larger UFs, and *p* variants being largely superior to *s* ones on division-based kernels.

5.7.5 Energy Efficiency Evaluation - *s* & *p* Rogue Variants

Figure 5.12 reports the energy gains achieved by the *s* and *p* Rogue variants over execution on the CV32E40P CPU for all considered integer activation kernels. Energy gain follows the same general trend already observed for throughput and speedup, it increases with the input sequence length and then gradually approaches a plateau.

However, unlike speedup, energy gain depends not only on latency reduction, but also on the overall power consumption of the considered system. In this analysis, the energy contribution of all system components is considered, including Rogue, the DMA subsystem, system memory, and the CPU. Across the considered activations, the average power consumption is 0.26 mW for *S4x2* and 0.47 mW for *P4x2*, increasing to 0.51 mW for *S4x4* and 0.88 mW for *P4x4*. Despite this increase, 4×4 variants consistently achieve higher energy gains than 4×2 ones, since their larger

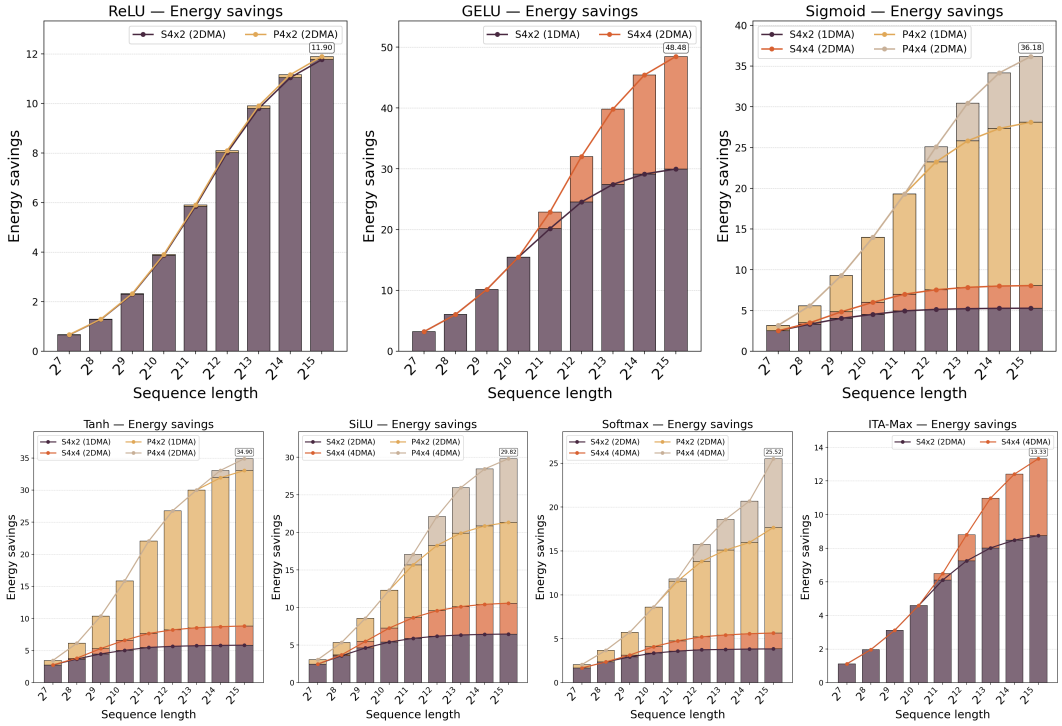


Fig. 5.12 Scaling of energy savings over execution on the CV32E40P implementing the *XCVPULP* ISA extensions for *S* and *P* Rogue implementations over growing input sequence length.

throughput and shorter execution time more than compensate for the higher power. Moreover, this also applies to the comparison between *p* and *s* Rogue variants.

For division-free activations (*ReLU* and *GELU*) the energy behavior of *s* and *p* variants is very similar. In this case, the obtained energy gains are mainly dictated by the computational complexity of the kernel and the corresponding speedup. As a result, *GELU* achieves significantly higher gains than *ReLU*, reaching a peak of about 48×, whereas the latter remains substantially lower because of its limited arithmetic complexity and stronger influence of memory-access conflicts.

For division-based single-transfer activations, the advantage of the *p* variants becomes evident. Although the pipelined divider increases power consumption, its much higher throughput leads to shorter execution times and therefore to larger overall energy gains. This is particularly visible for *Sigmoid* and *Tanh*, for which *p* configurations outperform *s* ones, reaching up to 36.2× and 34.9× respectively, thus showing that the speedup improvement more than compensates for the higher power cost of the pipelined divider.

For multi-transfer activations, energy gain is attenuated by the repeated DMA transactions and intermediate reconfiguration steps required to complete the computation. Nevertheless, these kernels still achieve substantial improvements over software execution. In particular, *SiLU* reaches peak energy gains of up to 30×, confirming that, even in the presence of additional transfer overheads, the acceleration provided by Rogue remains highly beneficial. *Softmax* also achieves significant gains, although lower than those of *Sigmoid* and *Tanh* due to the larger contribution of DMA and memory overhead. As already observed for throughput and speedup, *ITA-Max* represents a special case. Since its formulation requires far fewer divisions than standard *Softmax*, the advantage of the *p* variants over the *s* ones is much less pronounced.

Overall, the energy results show that **Rogue provides substantial energy savings over software execution, especially on long and computationally intensive workloads**. Moreover, 4×4 configurations consistently achieve higher gains than 4×2, despite their larger power consumption, thanks to their superior throughput. Finally, for division-based activations, the *p* variants prove more energy-efficient than the corresponding *s* ones, since the increased power of the pipelined divider is offset by the much larger reduction in execution time. The only relevant exceptions are division-free workloads, for which *s* and *p* variants overlap, and *ITA-Max*, where division has a reduced impact on the overall computation.

5.7.6 Performance Evaluation - *g* Rogue Variants

In the following sections, *g* Rogue variants are evaluated on a classical GeMM workload, matrix multiplication (*matmul*), in terms of throughput (MOPS) and speedup with respect to the CV32E40P executing *XCVPULP*-aware compiler-generated code. Therefore, this analysis is useful to assess Rogue not only as an accelerator for integer activation approximation, but also as a GeMM accelerator.

① **Throughput.** As for throughput for *matmul*, it is reported at different UFs in MOPS (Mega Operations per Second), a metric that also enables comparison against other State-of-the-Art CGRAs. Figure 5.13 reports MOPS scaling over increasing matrix sizes. As also shown in Table 5.3, *g2x2_2* supports up to UF1 using two concurrent DMA channels, *g2x4_2* supports up to UF2 using four concurrent DMA

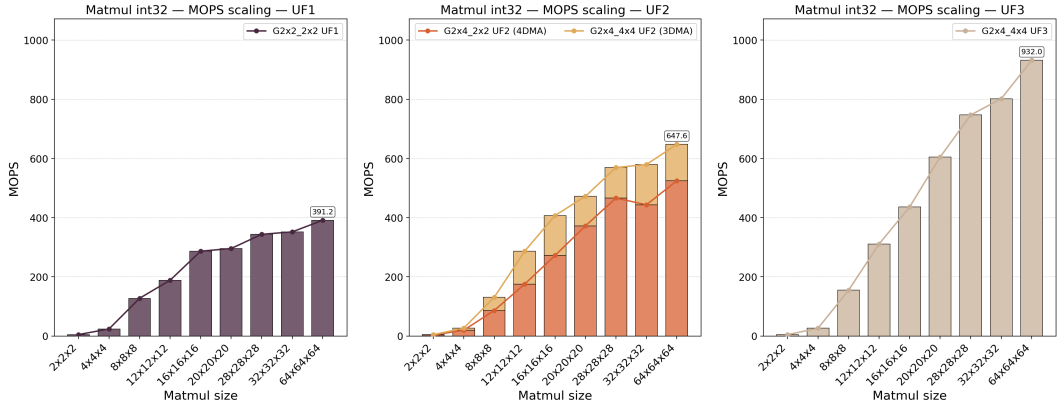


Fig. 5.13 Scaling of MOPS on *matmul* for *g* Rogue implementations over growing input matrices. The sizes are $N \times K \times M$, where N and K are rows and columns of the first input matrix, and K and M rows and columns of the second.

channels, and *g2x4_4* supports up to UF3 with four concurrent DMA channels, while implementing UF2 with only three channels.

A first clear trend is that **throughput increases with the available UF**. While UF1 peaks at 391.2 MOPS, UF2 exceeds 500 MOPS on *g2x4_2* and reaches 647.6 MOPS on *g2x4_4*, whereas UF3 achieves the highest value, **932.0 MOPS**. Hence, larger UFs effectively translate into higher computational throughput. However, MOPS scaling across increasing UFs is not proportional to the nominal increase in UF. Indeed, the gain from UF1 to UF2, and from UF2 to UF3, is smaller than what an ideal trend would suggest (e.g., MOPS doubling from UF1 to UF2). This indicates that, although higher UFs expose more parallelism, performance is still limited by data movement and system-level effects, which prevent perfectly linear throughput scaling.

A second relevant trend emerges from the comparison between the two UF2 implementations. For the same UF, *g2x4_4* outperforms *g2x4_2* across all matrix sizes. Since the former implements *matmul* at UF2 using only 3 DMA channels, whereas the latter requires 4 channels, the results indicate that **reducing the number of concurrent DMA transfers for the same UF improves effective throughput**. This can be explained by the fact that increasing the number of concurrent transfers for the same UF also increases the occurrence of memory access conflicts, which significantly reduce the attainable throughput.

A third trend is the steady **growth of throughput with matrix size for all supported UFs**. For very small matrices, throughput remains limited because fixed costs, such as DMA and Rogue configuration and pipeline filling, represent a significant fraction of execution time. As the problem size increases, these overheads are progressively amortized and the architecture operates for longer in steady-state conditions, thus yielding much higher MOPS. This effect is especially evident when moving from the smallest matrices to intermediate sizes, where all curves exhibit a steep increase.

At the same time, the curves start to grow less quickly for larger matrix sizes. In particular, UF1 increases rapidly up to medium-sized matrices and then more slowly, while UF2 also shows a smaller growth rate for larger workloads. Also, in *g2x4_2* UF2, a slight drop is also visible between the $28 \times 28 \times 28$ and $32 \times 32 \times 32$ cases, likely due to size-dependent sudden increase of memory access conflicts. On the other hand, UF3 still keeps growing strongly even at the largest tested size. Therefore, differently from activation kernels that tend to approach a clearer throughput plateau in the presented charts, *matmul* does not seem to reach a fully saturated regime up to $64 \times 64 \times 64$, especially for the most parallel configuration. This suggests that testing larger matrices would likely better show the maximum throughput reachable by the *g* variants.

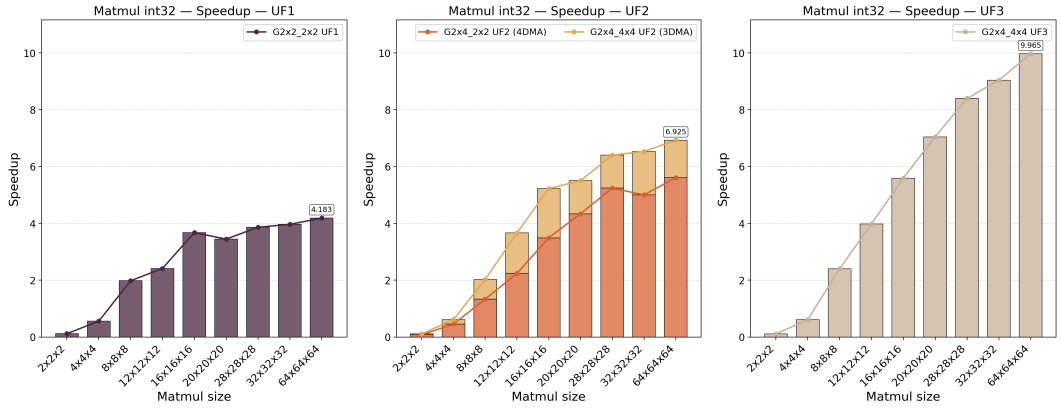


Fig. 5.14 Scaling of speedup compared to *CV32E40P* on *matmul* for *G* Rogue implementations over growing input matrices.

② **Speedup.** Figure 5.14 reports the speedup achieved by *g* Rogue variants on *matmul* with respect to the CV32E40P executing *XCVPULP*-aware compiler-generated code. Overall, the results confirm that Rogue provides a substantial

advantage over the CPU baseline, and that this advantage becomes more evident as matrix size increases. For the smallest inputs, speedup is limited because Rogue and DMA configuration overheads account for a significant fraction of total execution time. As matrix sizes grow, these fixed costs are amortized, and speedup increases accordingly.

The same overall behavior already observed for throughput across increasing UFs is also confirmed for speedup. In particular, *g2x2_2* at UF1 reaches up to 4.18×, UF2 exceeds 5× on *g2x4_2* and reaches 6.93× on *g2x4_4*, while UF3 on *g2x4_4* achieves the highest result, namely **9.97**×. Therefore, as expected, increasing the supported UF also translates into a progressively larger acceleration over software execution. Moreover, the comparison between the two UF2 implementations shows that *g2x4_4* consistently outperforms *g2x4_2*, confirming that realizing the same UF with fewer DMAs leads to higher acceleration.

Although a few local fluctuations are visible for specific matrix sizes across all scaling plots, the overall trend remains increasing for all supported UFs. Therefore, consistently with the throughput analysis, the speedup results show that *g* Rogue variants can effectively accelerate the *matmul* workload, with benefits that become increasingly relevant for larger matrices.

5.7.7 Energy & Area Efficiency Evaluation - *g* Rogue Variants

In this section, energy and area efficiency of *g* Rogue implementations is evaluated on the *matmul* workload across various UFs.

① **Energy Efficiency.** As for energy, savings are reported with respect to the execution on the CV32E40P and they are calculated considering all involved system components, as in

$$\frac{E_{CPU} + E_{Memory} + E_{Bus}}{E_{Rogue} + E_{Memory} + E_{Bus} + E_{DMA}} \tag{5.1}$$

On the other hand, energy efficiency is evaluated as MOPS/mW. As shown in Figure 5.15 and Figure 5.16, both metrics follow the same overall trend across the different *g* Rogue variants and UFs. They are both limited on the smallest matrices, and then they progressively improve as matrix size increases. This behavior can be

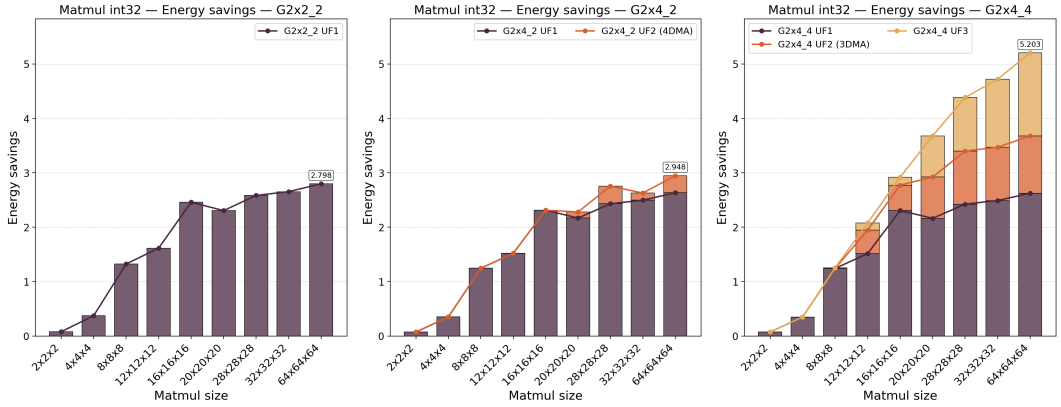


Fig. 5.15 Scaling of energy savings compared to *CV32E40P* on *matmul* for *G* Rogue implementations over growing input matrices.

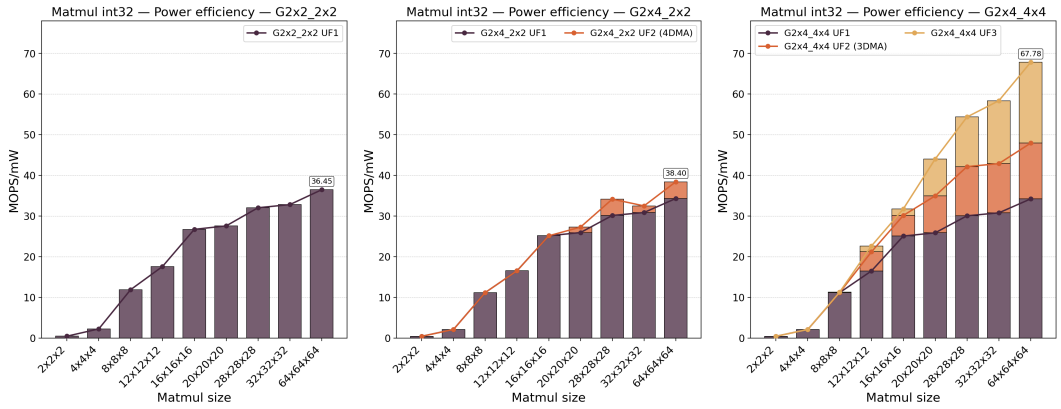


Fig. 5.16 Scaling of energy efficiency (MOPS/mW) on *matmul* for *G* Rogue implementations over growing input matrices.

mainly explained again considering the fixed costs of DMA and Rogue configuration times.

As expected, **increasing the supported UF improves both energy savings over software execution and energy efficiency**. As for energy savings, they increase from a peak of 2.80× on *g2x2_2* at UF1 to 2.95× on *g2x4_2* at UF2, up to the best saving of **5.20**× on *g2x4_4* at UF3. energy efficiency follows the same ordering, with UF1 on *g2x2_2* reaching **36.45 MOPS/mW**, UF2 on *g2x4_2* reaching 38.40 MOPS/mW, while the best overall value is again obtained by *g2x4_4* at UF3, with **67.78 MOPS/mW**. Therefore, the increased power consumption caused by larger

UFs do not translate into lower energy savings or poor energy efficiency, rather it is compensated by the performance gains enabled by larger UFs.

Furthermore, another trend can be observed when comparing architectures at the same UF. At UF1, the smallest variant, *g2x2_2*, is the most convenient solution from both points of view, since the larger *g2x4_2* and *g2x4_4* configurations do not yet exploit enough parallelism to compensate their higher hardware cost. However, once higher UFs are enabled, the larger variants recover this gap and become increasingly more attractive, eventually providing the best overall results. This confirms that **the additional hardware of the larger *g* variants becomes beneficial when it is effectively translated into higher sustained parallelism**.

In addition, as already analyzed for throughput and speedup, *g2x4_4* at UF2 consistently outperforms *g2x4_2* at the same UF in both energy savings and in MOPS/mW. Since the former implements UF2 with 3 DMAs, whereas the latter requires 4, these results confirm that reducing the number of concurrent DMA transfers improves not only throughput and speedup, but also the overall efficiency of the execution. In other words, a more efficient data-movement scheme benefits both performance and energy behavior.

Although a few local fluctuations are visible for specific matrix sizes, the overall trend remains clearly increasing in both energy savings and energy efficiency. Overall, the analysis leads to the conclusion that, if only limited parallelism is exploited, the smallest *g* Rogue variant is the most efficient one. However, when the larger architectures are fully exploited through higher UFs, *g2x4_4* becomes the best configuration also from an efficiency perspective, providing both the highest energy savings and the highest energy efficiency on *matmul*.

② **Area efficiency.** Area efficiency for *matmul* is reported across all *g* Rogue implementations in MOPS/mm^2, thus normalizing throughput to the post-synthesis area of each *g* Rogue variant. Since area is fixed for each architecture, the scaling along with matrix size presented in Figure 5.17 follows the same behavior already observed for throughput. As a result, area efficiency remains low when operating on small matrices, and then sharply increases for medium-sized ones as configuration overheads are amortized, and finally grows more gradually at larger sizes, with only minor local fluctuations.

The more relevant insight emerges from the comparison across architectures and UFs. As expected, the smallest variant, *g2x2_2*, is clearly the most area efficient

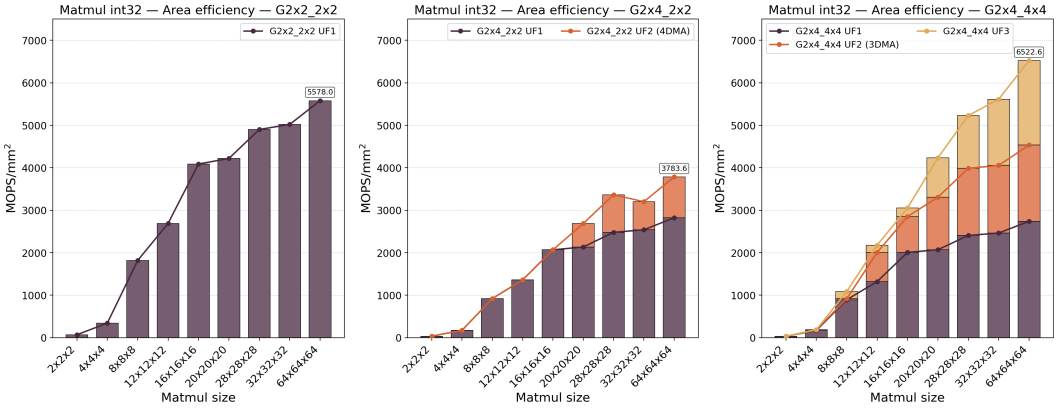


Fig. 5.17 Scaling of area efficiency (MOPS/mm^2) on *matmul* for *G* Rogue implementations over growing input matrices.

at UF1, reaching up to **5578.0 MOPS/**mm^2, while UF1 on *g2x4_2* and *g2x4_4* remains around 2800 MOPS/mm^2. This indicates that increasing the architecture size without also increasing the exploited parallelism penalizes area efficiency, since the additional hardware does not translate into a proportional throughput increase.

However, when higher UFs are enabled, this penalty is progressively recovered. On *g2x4_2*, UF2 raises area efficiency up to **3783.6 MOPS/**mm^2, while on *g2x4_4* UF2 reaches about 4500 MOPS/mm^2, again confirming the benefit of realizing the same UF with fewer DMAs. The best result is obtained by *g2x4_4* at UF3, which reaches **6522.6 MOPS/**mm^2. Therefore, the largest *g* Rogue variant becomes the most area efficient only when its additional hardware is effectively exploited through a higher UF.

5.7.8 Comparison with State-of-the-Art Accelerators & CGRAs

In this section, *s* and *p* implementations of Rogue are compared to other State-of-the-Art accelerators for activation functions, while *g* Rogue variants are evaluated against Mage and State-of-the-Art CGRAs mainly targeting GeMM workloads.

s & *p* Rogue vs. Accelerators for Activation Functions

In Table 5.6, results achieved by Rogue *s* and *p* implementations are compared against state-of-the-art accelerators for activation functions.

Table 5.6 Comparison with State-of-the-Art Accelerators.

Work	Node [nm]	Frequency [MHz]	Area [$\times 10^3 \mu m^2$]	Power [mW]	Throughput [OpC]	Format/ Bitwidth	Supported [a] Activations	Approximation Method
ITA [74]	22	500	6	0.85	1[b]	int/8	IM	Base substitution
[71]	180	471	11	-	1	Fixed Point/16	T	LUT
NACU [80]	28	267	1	2.16 (S), 1.95 (T)	1 (S, T) [b]	Fixed Point/16	T, S, Sm	PWLA
Flex-SFU [65]	28	500	4 - 22	0.4 - 1.6 (all its functions)	4 (all functions)	FP-int/ 8-16-32	Sm, G, T, S, SL, HS	1^{st}, 2^{nd} order PPA
Rogue [this work]	16	330	16 (*S4x2*) - 46 (*P4x4*)	(S) 1.5[c]- 3.2[d] (T) 1.6[c]- 3.2[d] (IM) 0.5[e]- 2.4[f]	(G) 1.85[h]- (S) 1.58[g] (IM) 0.9[h]- (T) 1.37[g] (Sm) 0.63[g]	int/8-16-32	Sm, G, T, S, SL, RL, IM	any integer approximation

[a] Tanh (T), Sigmoid (S), Softmax (Sm), GELU (G), ReLU (R), SiLU (SL), HardSwish (HS), ITA-Max (IM).
[b] Although not explicitly reported, it can be reasonably inferred.
[c, d, e, f] Power consumption of Rogue *P4x4* (c) and *S4x4* (e), power consumption of X-HEEP with Rogue *P4x4* (d) and with *S4x4* (f).
[g, h] Peak throughput on Rogue *P4x4* (g), peak throughput on Rogue *P4x4* and *S4x4* (h).

A first comparison can be made considering **fixed-function accelerators**. Fixed-point and integer accelerators targeting a specific activation function, such as [71] and ITA [74], can be more compact than Rogue. Indeed, their designs can be tightly co-optimized with the adopted activation approximation strategy. Nonetheless, Rogue provides substantial advantages by supporting a wider range of activation functions, while also delivering competitive throughput with [71] on *Tanh* using the *p4x2_2* variant, and surpassing it by 37% with *p4x4_2*. Regarding throughput, it is important to highlight that the selected accelerators, apart from ITA, do not consider system-integration effects, such as memory-access conflicts, as Rogue does. In addition, Rogue *S4x4* nearly matches ITA's throughput on its own approximation (*ITA-Max*), while using 41% less power.

Furthermore, Table 5.6 also reports comparisons with **reconfigurable accelerators for activation functions**. For example, NACU [80] exploits *PWLA* to approximate the exponential function and, in turn, *Tanh* and *Sigmoid*. At 28 nm, it is more compact than the smallest Rogue variant at 16 nm. Although *PWLA* enables low-area, high-throughput implementations, it requires careful tuning of coefficients and intervals to minimize errors for specific functions. Therefore, without considering system-level effects, NACU achieves an ideal throughput of 1 OpC with less area than Rogue, but it remains specifically tailored to a restricted set of functions, namely *Tanh* and *Sigmoid*, whereas Rogue supports a broader set of activations. In addition, Rogue *p4x2_2* nearly matches NACU's peak ideal throughput on *Tanh* and *Sigmoid*, while *p4x4_2* exceeds it by 37% and 58%, respectively. Moreover, Rogue reduces power on *Tanh* and *Sigmoid* compared to NACU, although NACU is implemented on an older node.

As regards Flex-SFU [65], it supports a wide range of activations by implementing a flexible *PWPA*-based approximation strategy. It supports nearly the same set of activations as Rogue, and it also implements *HardSwish* ($\frac{ReLU6(x+3)}{6}$), but it lacks support for *ITA-Max*. However, while Rogue and its *ISA* can be easily extended to simple *PWPA*-based functions such as HardSwish, as well as to future more complex approximations, Flex-SFU cannot accommodate *ITA-Max*, since it only supports *PWPA*. Furthermore, Flex-SFU also reports lower area and power than Rogue, but it is important to note that the reported figures exclude the floating-point multiply-add units shared with the vector processor it is *ideally* integrated into. Nonetheless, its vector-processing capabilities allow higher peak throughput than Rogue on `int8` data, although throughput is halved when parallelism is doubled, thus decreasing to 1 OpC in `int32`. On the other hand, Rogue's current implementation achieves the same throughput on `int8`, `int16`, and `int32` data, and it could be further enhanced by introducing *SIMD* support in order to reach throughput comparable to Flex-SFU.

Furthermore, NX-CGRA [17] and PICACHU [16] would also be interesting comparison points for Rogue, as they are **CGRAs supporting activation functions**, but a fair comparison against them is very difficult. For PICACHU, the authors only report relative results, e.g., against an unknown baseline CGRA in terms of speedup on non-linear functions, which makes a direct comparison impossible. For NX-CGRA, the comparison is difficult because throughput (GOPS), area efficiency (GOPS/mm^2), and energy efficiency (GOPS/mW) are reported, but neither the methodology used to count operations nor the approximation strategy adopted for *GELU* and *Softmax* is specified.

Overall, Rogue does not aim to outperform activation-specific accelerators in absolute area or power. Rather, **its contribution lies in providing a more versatile activation accelerator that still achieves competitive throughput and efficiency across several functions, while also remaining extensible to future activations**. In this sense, Rogue occupies an intermediate design point between narrowly specialized units and more flexible activation accelerators, retaining enough specialization to achieve substantial gains while supporting a broader and potentially evolving set of integer activation approximations.

g Rogue vs. Mage

In this section, Rogue *g* variants are compared against scalar and vector Mage configurations. In particular, the comparison considers two design points. Rogue *g2x2_2* is compared against Mage *2x2_2x2_s* and *2x2_2x2_v*, while Rogue *g1x4_4* is compared against Mage *4x2_2x4_s* and *4x2_2x4_v*, as they are equally sized in terms of PEA. This comparison is especially relevant, as *it highlights the trade-offs between the two DAE-based CGRA design approaches proposed in this thesis*. Since Rogue is evaluated only on the *matmul* workload, the reported Mage results consider the same kernel on the same input matrices size ($32\times32\times32$) to ensure a consistent comparison.

Table 5.7 reports the main implementation characteristics of the considered Rogue and Mage variants, including PEA size, number of AGEs, area, and power. As expected, **Rogue is substantially more compact than Mage regarding system-level area overhead** at both comparison points. For the smaller configurations, *g2x2_2* occupies 0.070 mm^2, whereas Mage *2x2_2x2_s* and *2x2_2x2_v* occupy 0.220 mm^2 and 0.230 mm^2, respectively. Also, for the larger configurations, *g2x4_4* occupies 0.143 mm^2, while Mage *4x2_2x4_s* and *4x2_2x4_v* reach 0.433 mm^2 and 0.454 mm^2. Therefore, for similarly sized PEAs, Rogue incurs about $3\times$ less system-level area overhead than Mage, independently of the considered scale.

This gap is mainly explained by Mage's usage of an internal SpM, which is the dominant contributor to its total area, while Rogue relies on a lighter streaming architecture that leaves data storage to the main system memory and address generation to the system's DMA. Nevertheless, when only logic is considered, the difference becomes much smaller. Excluding the internal scratchpad, *4x2_2x4_s* occupies 0.129 mm^2 and *4x2_2x4_v* 0.150 mm^2, values much closer to *g2x4_4*. A similar trend emerges when focusing only on the PEA. The PEA of Mage *4x2_2x4_s* and *4x2_2x4_v* occupies 0.061 mm^2 and 0.083 mm^2, respectively, compared to 0.111 mm^2 for Rogue *g2x4_4*. This confirms that Rogue's area penalty is concentrated in the compute fabric, as its PEs implement a wider ISA than Mage's ones and they must also support delay operands, which increase both routing complexity and internal steering logic.

A similar trend can be observed for *accelerator-only* power consumption. For the smaller configurations, Rogue *g2x2_2x2* consumes about $4.3\times$ less power than

Table 5.7 Technology, area, and power comparison between Rogue and Mage.

Work	Node [nm]	Frequency [MHz]	Area [mm^2]	PEA [R×C]	AGEs	Power [mW]
Rogue	65	250	0.070 ↔ 0.143	2×2 ↔ 2×4	8 (4-4)	1.63 (10.73)[c] ↔ 2.96 (13.75)
Mage scalar `int32`[a]	65	250	0.220 ↔ 0.433	2×2 ↔ 4×2	4 (2-2) ↔ 8 (4-4)	7.09 ↔ 17.90
Mage vector `int32`[b]	65	250	0.230 ↔ 0.454	2×2 ↔ 4×2	4 (2-2) ↔ 8 (4-4)	7.30 ↔ 18.60
Mage vector `int16`[b]	65	250	0.230 ↔ 0.454	2×2 ↔ 4×2	4 (2-2) ↔ 8 (4-4)	7.23 ↔ 18.20
Mage vector `int8`[b]	65	250	0.230 ↔ 0.454	2×2 ↔ 4×2	4 (2-2) ↔ 8 (4-4)	7.36 ↔ 17.64

[a] Values correspond to peak *matmul* results on *2x2_2x2_s* and *4x2_2x4_s*.
[b] Values correspond to peak *matmul* results on *2x2_2x2_v* and *4x2_2x4_v* for `int32`, `int16` and `int8`.
[c] Values in parenthesis include system contributions, while others only Rogue.

Mage *2x2_2x2*, while for the larger configurations *g2x4_4x4* consumes about 6× less power than Mage *4x2_2x4*. As happens for the area comparison, this difference is largely explained by the different memory organization of the two architectures. Mage integrates a multi-bank internal SpM together with AGU logic for affine address generation, while Rogue relies on a streaming organization and offloads data storage and load-store operations to the system memory and DMA. As a result, Mage internalizes a larger fraction of the memory- and access-side cost, both in area and in power.

However, the comparison becomes more intricate when system-related contributions are also included for Rogue. In the smaller case, Rogue *g2x2_2* power exceeds Mage *2x2_2x2*, indicating that, at this scale, DMA and system-memory overhead dominate the total cost of Rogue. However, when considering the larger case, Rogue still achieves a power advantage of about 1.3× over Mage *4x2_2x4*. This suggests that, although part of Rogue's cost is shifted to the surrounding system, Mage still pays a higher intrinsic power cost to sustain execution at the larger scale.

This gap can be attributed to multiple factors. First, the *matmul* mapping differs between Rogue and Mage. At UF3, Rogue multiplies the same column element by three different row elements, resulting in four loads and three stores. Mage, instead, multiplies three row-column pairs in parallel and accumulates the partial results, thus requiring six loads and one store. Second, Mage exploits a richer access side, with more AGEs active in parallel and more complex address-generation logic than

Table 5.8 Performance and efficiency comparison between Rogue and Mage.

Work	Node [nm]	MOPS	MOPS/mW	MOPS/mm^2
Rogue	65	351.9 ↔ 801.9	215.89 (32.79) [c] ↔ 270.91 (58.32)	5018.2 ↔ 5612.4
Mage scalar `int32`[a]	65	499.1 ↔ 1446.7	70.40 ↔ 80.82	2266.0 ↔ 3342.0
Mage vector `int32`[b]	65	499.0 ↔ 1446.9	68.36 ↔ 77.78	2166.0 ↔ 3189.0
Mage vector `int16`[b]	65	996.8 ↔ 2759.0	137.9 ↔ 151.6	4327.0 ↔ 6081.0
Mage vector `int8`[b]	65	1984.6 ↔ 5237.6	269.6 ↔ 296.9	8615.0 ↔ 11544.0

[a] Values correspond to peak *matmul* results on *2x2_2x2_s* and *4x2_2x4_s*.
[b] Values correspond to peak *matmul* results on *2x2_2x2_v* and *4x2_2x4_v* for `int32`, `int16` and `int8`.
[c] Values in parenthesis include system contributions, while others only Rogue.

X-HEEP's DMA, which mainly relies on counters. Third, Mage sustains a wider memory-access front, with more banks and more address streams active in parallel. Taken together, these factors increase switching activity in the memory subsystem, in the access logic, and in the interconnect, thereby contributing to Mage's higher power.

Although Mage is disadvantaged compared to Rogue in terms of power consumption and system-level area overhead, it compensates for these costs by delivering higher throughput. As reported in Table 5.8, Mage consistently outperforms Rogue in terms of raw *matmul* throughput. For instance, at the smaller comparison point, Mage scalar *2x2_2x2* reaches 499.1 MOPS against 391 MOPS for Rogue *g2x2_2x2*. At the larger point, Mage scalar *4x2_2x4* reaches 1446.7 MOPS, whereas Rogue *g2x4_4x4* reaches 932 MOPS. The gap becomes even larger for Mage vector variants, which reach up to 1984.6 MOPS on *2x2_2x2* and up to 5237.6 MOPS on *4x2_2x4* in the int8 case. Therefore, Mage is clearly more effective at extracting raw *matmul* throughput, especially when reduced-precision vector execution is exploited.

Nonetheless, Rogue throughput can still be improved by acting on several design knobs. A first option is increasing the number of memory banks across which data is interleaved, since this reduces bank conflicts and slightly improves the sustained throughput. However, unlike Mage, *Rogue cannot exploit a fully conflict-free data*

layout, and this limits the maximum achievable performance. A second option is increasing the number of available DMA channels, so as to provide more concurrent data streams to the accelerator. However, this would also increase the associated area cost, thereby reducing the resulting area efficiency. Finally, part of the gap with respect to Mage vector variants could be bridged by extending Rogue with SIMD operators and by transferring packed data through the DMA. Such an enhancement would be relatively simple to integrate, as it would mainly require replacing the current addition, multiplication, and shift units with SIMD counterparts, without the need of modifying Rogue's control logic.

Nevertheless, Rogue remains highly competitive in normalized metrics. In particular, Rogue achieves higher MOPS/mm^2 than Mage scalar and Mage vector `int32` at both comparison points. This advantage also extends to Mage vector `int16`, while Mage overtakes Rogue only in the vector `int8` case, where packed-SIMD execution provides a much larger arithmetic density. Hence, despite its lower absolute throughput, Rogue often uses area more efficiently than Mage.

A similar observation applies to energy efficiency. When only accelerator energy is considered, Rogue clearly outperforms Mage scalar and Mage vector `int32` at both sizes, and it also remains ahead of Mage vector `int16`. Only Mage vector int8 on the smaller configuration slightly surpasses Rogue in MOPS/mW, while for the larger array Rogue still remains competitive. However, in this context, a fairer comparison between the two CGRAs would also consider Rogue together with its system-related contributions. In this setup, Rogue's energy efficiency gap with Mage becomes much smaller than what accelerator-only figures suggest, and it shifts in Mage's favor. This effect is especially evident on smaller arrays, where the system overhead associated with data movement has a stronger impact on Rogue. On the larger variants, Rogue remains competitive, but its advantage is no longer as pronounced as in the accelerator-only view. Therefore, the energy efficiency comparison highlights that Rogue is intrinsically very efficient at the accelerator level, but that part of this benefit is offset at the platform level by the cost of the external memory and DMA infrastructure. At the same time, if those contributions are taken into account, **Mage proves to be slightly more energy efficient compared to Rogue, compensating for its larger power consumption with its higher throughput**.

Overall, the comparison highlights a clear trade-off. Mage is the preferable solution when maximizing *matmul* throughput is the primary objective, especially

Table 5.9 Comparison between Rogue and other state-of-the-art DAE CGRAs.

Work	Node [nm]	Frequency [MHz]	PEA Area [mm^2]	AGU Area [mm^2]	PE Area [mm^2]	AGE Area [mm^2]
Rogue	65	250	0.054 ↔ 0.111	0.008 ↔ 0.016	0.013	-
	40		0.020 ↔ 0.041	0.003 ↔ 0.006	0.005	
	22		0.006 ↔ 0.012	0.001 ↔ 0.002	0.001	
CASCADE [1]	40	510	∼0.130 [a]	∼0.010 [a]	∼0.006 [a]	∼0.001 [a]
STRELA [26]	65	250	0.217	0.027	0.014	0.003
NX-CGRA [17]	22	200	0.164 [b]	0.164 [b]	-	-

[a] Values are extracted from a chart, hence they may be slightly different.
[b] Values are reported for the AGU-PEA combination only.

when vector reduced-precision execution can be exploited. Rogue, instead, offers a much more compact design and, in many cases, higher area efficiency, while retaining competitive energy efficiency at the accelerator level. Therefore, **Rogue is better suited to designs in which hardware cost and compute density are the primary constraints, whereas Mage is more suitable when peak matrix-computation throughput is the dominant target**.

g Rogue vs. State-of-the-Art CGRAs

Table 5.10 and Table 5.11 compare Rogue against a subset of state-of-the-art CGRAs. It is important to note that the considered architectures differ in technology node, operating frequency, array size, and benchmark setup. Therefore, this comparison should be seen as an attempt to Rogue's positioning within the current CGRA landscape, rather than as a strict one-to-one evaluation. To make the comparison as meaningful as possible, in addition to the retrieved 65 nm results, scaled Rogue values at 45, 40, 28, and 22 nm are also reported, obtained through DeepScaleTool [63]. Moreover, whenever possible, the reported figures for the other CGRAs refer to the same workload considered for Rogue, namely *matmul*. When this is not possible, the average values reported in the corresponding papers are used instead. Finally, it is worth pointing out that, among the considered architectures, CASCADE, STRELA, and NX-CGRA are also DAE-based CGRAs.

With respect to area, Rogue proves to be very compact. At 65 nm, the largest Rogue configuration occupies only 0.143 mm^2, compared to 0.25 mm^2 for **STRELA**, which is the only other design implemented at the same node and frequency. It

Table 5.10 Technology, area, and power comparison between Rogue and other state-of-the-art CGRAs.

Work	Node [nm]	Frequency [MHz]	Area [mm^2]	PEA [R×C]	AGEs	Power [mW]
Rogue[a]	65	250	0.070 ↔ 0.143	2×2 ↔ 2×4	8 (4-4)	1.63 (10.73) ↔ 2.96 (13.75)
	45		0.034 ↔ 0.070			1.27 (8.37) ↔ 2.31 (10.73)
	40		0.026 ↔ 0.053			1.19 (7.83) ↔ 2.16 (10.04)
	28		0.012 ↔ 0.025			0.91 (6.01) ↔ 1.66 (7.70)
	22		0.008 ↔ 0.015			0.78 (5.15) ↔ 1.42 (6.60)
CASCADE [1]	40	510	∼0.57	4×4	16 (8-8)	73[f]
STRELA [26]	65	250	0.25	4×4	8 (4-4)	7.46[a]
NX-CGRA [17]	22	200	0.178	4×4	8	1.51[b]
IPA [22]	28	100	0.20	4×4	-	0.49[c]
DA-CGRA [57]	45	100	0.20	5×4	-	5.47[e]
Riptide [20]	22	50	0.25	6×6	-	0.50[g]
R-BLOCKS [19]	22	100	0.486	11×7	-	5.84[d]

[a] Values refer to 64×64×64 *matmul* on `int32` data (values in parentheses account for system-related contributions).
[b] Values refer to 32×64×32 *matmul* on `int8` data using packed SIMD operators.
[c] Values refer to 16×16×16 *matmul* on `int32` data.
[d] Values refer to 32×32×32 *matmul* on `int16` data.
[e] Values refer to 64×64×64 *matmul* on `int16` data, and values in the related paper are scaled to 22nm.
[f] Average value reported.

should be noted that STRELA employs a 4×4 PEA, whereas Rogue uses a 4×2 one. As a result, a 4×4 Rogue PEA would likely be slightly larger than STRELA. Nevertheless, Rogue still delivers more than 2× higher throughput than STRELA at 250 MHz on the same *matmul*, which results in higher area efficiency for the considered 4×2 Rogue PEA. Even if a 4×4 Rogue PEA were considered, the improvement in throughput would still not be offset by the increase in area cost, so area efficiency would remain higher. More generally, Rogue also remains very compact with respect to the other considered CGRAs, since even when scaled to more advanced nodes, its area stays well below that of both DAE-based and conventional state-of-the-art CGRAs. In addition, Table 5.9 provides comparison on PEA and AGU with other DAE CGRAs. The area occupation of Rogue's PE is comparable to STRELA's, suggesting that the increased area in a 4×4 configuration would be induced by the additional interconnection complexity. Rogue's PE is also comparable to CASCADE's when scaled to 40 nm. As regards AGU, values reported for Rogue refer to the DSMU, which is not a real AGU, as Rogue relies on external LSUs. As a

Table 5.11 Performance and efficiency comparison between Rogue and other state-of-the-art CGRAs.

Work	Node [nm]	MOPS	MOPS/mW	MOPS/mm^2
Rogue[a]	65	391 ↔ 932	239.88 (36.44) ↔ 314.86 (67.78)	5575.4 ↔ 6522.8
	45	391 ↔ 932	307.52 (46.72) ↔ 403.66 (86.90)	11373.8 ↔ 13306.5
	40	391 ↔ 932	328.63 (49.92) ↔ 431.36 (92.86)	15165.0 ↔ 17741.9
	28	391 ↔ 932	428.42 (65.08) ↔ 562.35 (121.06)	32498.9 ↔ 38021.2
	22	391 ↔ 932	499.66 (75.90) ↔ 655.86 (141.19)	51700.5 ↔ 60485.7
CASCADE [1]	40	-	-	-
STRELA [26]	65	437.8[a]	58.66[a]	1751.2[a]
NX-CGRA [17]	22	3040[b]	2010.0[b]	17078.7[b]
IPA [22]	28	65.98[c]	134.65[c]	329.9[c]
DA-CGRA [57]	45	2845[e]	529.29[e]	14225[e]
Riptide [20]	22	164[a]	328.00[a]	656.0[a]
R-BLOCKS [19]	22	678[d]	116.10[d]	1395.1[d]

[a] Values refer to 64×64×64 *matmul* on `int32` data (values in parentheses account for system-related contributions).
[b] Values refer to 32×64×32 *matmul* on `int8` data using packed SIMD operators.
[c] Values refer to 16×16×16 *matmul* on `int32` data.
[d] Values refer to 32×32×32 *matmul* on `int16` data.
[e] Values refer to 64×64×64 *matmul* on `int16` data, and values in the related paper are scaled to 22nm.

result, the DSMU is much smaller than the AGUs of other CGRAs. Therefore, *this strengthens Rogue's area efficiency and its low system-level area overhead.*

A similar observation can be done for power. Rogue again outperforms STRELA, consuming 2.96 mW in *g2x4_4* compared to the 7.46 mW reported for STRELA. Moreover, Rogue would likely still consume less power even with a 4×4 PEA, as power does not scale linearly with array size and unused PEs are fully clock-gated. Compared with the other state-of-the-art CGRAs, Rogue also exhibits lower power consumption than most designs, with the exception of Riptide and IPA, which were explicitly developed with low-power operation as a main target. It should also be emphasized that the considered CGRAs differ in operating frequency, which further complicates a fair comparison. Even so, Rogue still compares favorably, providing power figures in the same order of magnitude despite the older technology node and the relatively high operating frequency.

As for throughput, it must be discussed more carefully, since the definition of MOPS is not consistent across all CGRA implementations. Rogue and STRELA explicitly count only arithmetic operations, thus providing a metric that is as architecture-agnostic as possible. By contrast, other works often do not specify the adopted counting methodology. A specific case is the one of DA-CGRA, which also includes address generation and memory access operations, thereby significantly inflating the reported MOPS with respect to Rogue and STRELA. This makes a direct comparison of absolute throughput more difficult. Nonetheless, some general trends can be identified. Rogue surpasses STRELA at the same frequency, input size, and UF, achieving about 2× higher throughput on *matmul*. NX-CGRA reports a much higher throughput, namely 3040 MOPS, but on `int8` data and for a smaller 32×64×32 *matmul*. Although NX-CGRA does not explicitly state how MOPS are computed, its implementation is conceptually close to Rogue and STRELA, as it relies on MOBs to generate memory accesses toward the X-HEEP main memory. Therefore, the higher reported throughput is due to the exploitation of SIMD execution and packed data on NX-CGRA, which would also greatly benefit Rogue if similar support were introduced. By contrast, R-BLOCKS, Riptide, and IPA report substantially lower raw throughput than Rogue, although the benchmark conditions are not always identical.

When area and energy efficiency are also taken into account, Rogue's position becomes even clearer. In terms of area efficiency, at 65 nm Rogue already reaches 5575.4 MOPS/mm^2 in its smaller variant at UF1, and up to 6522.8 MOPS/mm^2 in its larger variant at UF3. Both values are significantly higher than STRELA's 1751.2 MOPS/mm^2 at the same frequency, even though STRELA employs a larger array. Moreover, a 4×4 Rogue PEA would still retain better area efficiency. Rogue also surpasses R-BLOCKS (1395.1), Riptide (656.0), and IPA (329.9). Although these designs operate at lower frequencies than Rogue, they are also implemented in more advanced technology nodes. Naturally, Rogue's area efficiency figures become even more favorable when scaled nodes are considered. Only DA-CGRA and NX-CGRA report higher area efficiency values, namely 14225 and 17078.7 MOPS/mm^2, respectively. However, when scaled Rogue figures are taken into account, the comparison shifts in Rogue's favor. In particular, the larger Rogue variant reaches 60485.7 MOPS/mm^2 at 22 nm, thus largely surpassing both NX-CGRA and DA-CGRA, whose reported values are also scaled to 22 nm in the corresponding paper. Even if a 4×4 PEA were assumed for Rogue, its area efficiency would still remain

higher. This confirms that Rogue is especially strong in terms of compute density, which represents one of its main architectural advantages.

As for energy efficiency, when only accelerator power is considered, Rogue reaches 314.86 MOPS/mW in UF3 at 65 nm, outperforming STRELA at the same frequency and UF, whose values reported in Table 5.11 refer only to the accelerator itself. In addition, the comparison at 22 nm against Riptide and R-BLOCKS remains favorable for Rogue even when the overall system is considered. With respect to DA-CGRA, Rogue achieves lower energy efficiency when the full system is taken into account, although it should again be recalled that DA-CGRA reports inflated MOPS due to its broader operation count. NX-CGRA is the only design that reports substantially higher values, although these results are obtained for `int8` computation. Overall, Rogue proves to be comparable to, and in some cases better than, other state-of-the-art CGRAs also in terms of energy efficiency, while at the same time maintaining very high area efficiency.

Overall, the comparison shows that Rogue occupies a very competitive position within the current CGRA landscape. **While Rogue does not always achieve the highest absolute throughput, it combines compact area, high compute density, and competitive energy efficiency, making it particularly attractive for area- and efficiency-constrained designs**.

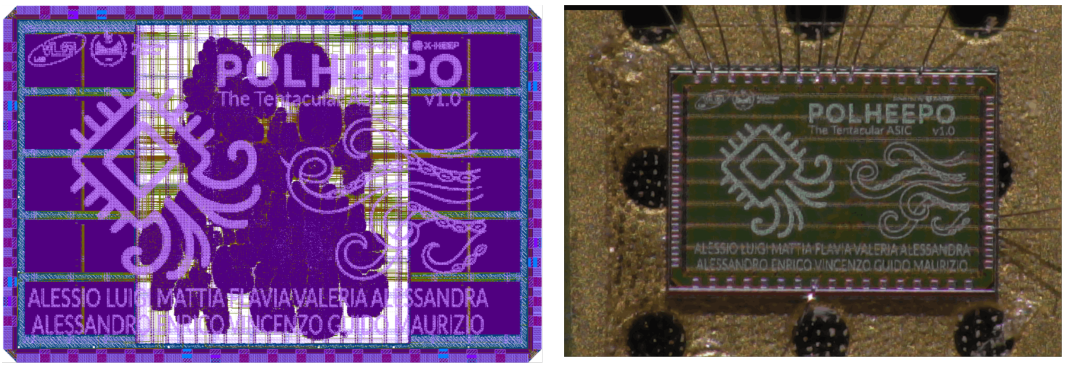


Fig. 5.18 On the left, polHEEPo's layout is shown, while, on the right, the polHEEPo chip is shown mounted on a temporary board for testing its functionalities, waiting for the custom boards to be shipped.

5.8 Rogue Integration in the polHEEPo SoC

The Rogue *p4x4_2* variant has been integrated into **polHEEPo**, an X-HEEP-based SoC taped out in 65 nm *TSMC Low-Voltage Threshold* technology and operating at 100 MHz. polHEEPo includes an X-HEEP-based system encompassing a CV32E40P CPU with *XCVPULP* extensions, 320 KiB of system memory organized as ten 32-KiB interleaved SRAM banks, an *NtoM*-topology system bus, and a peripheral subsystem. The latter includes four DMA channels, all configured with the full set of available features and with access to four bus masters, together with SPI interfaces, an I2C interface, GPIOs, a UART, and JTAG.

In addition to the base platform, polHEEPo integrates several hardware accelerators targeting post-quantum cryptography and ML. Among them, Rogue *p4x4_2* is included as a streaming DMA-coupled accelerator, with particular focus on integer activation kernels. By exploiting the FIFO-based interface available on each of the four DMA channels, Rogue is tightly coupled to the DMA subsystem and can execute integer activation workloads on the fly as data is streamed through the accelerator.

Layout and SoC images of polHEEPo are shown in Figure 5.18. At the time of writing, the chip has been successfully received and validated across all major components, including the integrated accelerators. However, the measurement setup is still under development. Nevertheless, in parallel, a tiny-Transformer application for embedded *Human Activity Recognition* (HAR) is being implemented, using Rogue to accelerate the *Softmax*, *GELU*, and *SiLU* kernels.

5.9 Conclusions

This chapter presented Rogue, a template-based streaming DAE CGRA specifically tailored for integer approximations of activation functions. Rogue combines a PEA implementing a dedicated ISA for recurrent activation-related computational patterns with a DSMU providing a FIFO-based interface toward external AGUs or directly toward the system DMA, thus enabling DMA-coupled streaming acceleration within X-HEEP. Although being tailored to integer activations, Rogue can also be exploited as a standard CGRA to accelerate GeMM workloads. A design-space exploration was carried out by exploiting Rogue's template-based design, showing that this approach

enables multiple architectural trade-offs in terms of PEA size, PE composition, synchronization flexibility, and DMA-coupled data streaming.

The evaluation confirmed that the analyzed *s* and *p* Rogue implementations are highly effective on integer activation workloads, achieving high speedups together with substantial energy savings compared to software execution on the CV32E40P CPU, while also retaining a very compact hardware footprint. Across the considered activation kernels, Rogue achieved up to 98.03× speedup on Sigmoid, 92.95× on Tanh, and about 48× energy savings on GELU over execution on the CV32E40P CPU in its *p4x4_2* variant. In addition, the homogeneous *g* variants showed that Rogue can also act as a compact GeMM accelerator, reaching up to 9.97× speedup and 5.20× energy savings compared to CV32E40P CPU running compiler-generate *XCVPULP*-aware code, as well as 932.0 MOPS, 67.78 MOPS/mW when system contributions are included, and 6522.6 MOPS/mm^2 on *matmul*. At the same time, Rogue incurs only limited system-level area overhead, ranging from 2.8% for the smallest *g* variant up to 12.2% for the largest *p* one.

At the same time, the comparison among the different Rogue implementations highlighted several design-space trends:

1. 4×4 *p* variants generally provide the highest throughput and energy gains on activation kernels, since the larger PEA enables higher UFs and therefore a greater amount of exploitable parallelism. Despite the higher power consumption, the resulting reduction in execution time is large enough to also improve energy savings.

2. *p* variants are clearly preferable on division-based activations, since the pipelined divider removes the main bottleneck affecting both software execution and *s* variants. On the other hand, on division-free workloads such as ReLU and GELU, *s* and *p* variants show almost the same behavior, while ITA-Max represents an intermediate case in which the divider has a more limited impact.

3. For *matmul*, if only limited parallelism is exploited, the smallest *g* variant is the most convenient one in terms of both area and energy efficiency. However, once larger implementations enable higher UFs, they progressively recover this gap and *g2x4_4* becomes the preferable design point, providing the highest throughput, energy efficiency, and area efficiency.

4. On streaming workloads, **the efficiency of data movement remains a key factor**. In particular, reducing the number of concurrent DMA transfers improves not only throughput and speedup, but also energy efficiency, showing that Rogue performance depends not only on the PEA itself, but also on how effectively the surrounding DMA-coupled streaming scheme is organized.

Furthermore, Rogue *g* variants have been compared with Mage, enabling a thorough analysis of both benefits and disadvantages of the different DAE approaches adopted in this thesis for CGRA implementations. **Rogue proves to be substantially more compact than Mage**, incurring about 3× lower system-level area overhead than Mage for similarly sized PEAs, while also achieving lower power consumption. At the same time, **Mage remains the best solution when maximizing raw throughput is the main objective**, especially when reduced-precision vector execution can be exploited. Nonetheless, **Rogue remains highly competitive in throughput-normalized metrics**, hence often delivering higher area efficiency and strong accelerator-level energy efficiency.

Compared to state-of-the-art CGRAs, Rogue further confirms this positioning by combining compact area, low power, high compute density, and competitive energy efficiency. In particular, it already surpasses STRELA in throughput and area efficiency at the same frequency, while scaled results further strengthen its position against other CGRAs. Overall, the comparison highlights a clear trade-off: Mage is preferable when peak matrix-computation throughput is the dominant target, whereas Rogue is better suited to designs in which hardware cost, compactness, and compute density are the primary constraints.

Furthermore, the comparison between Rogue *g* variants and Mage helps better clarify the trade-offs between the two DAE approaches explored in this thesis. For similarly-sized PEAs, **Rogue proves to be substantially more compact than Mage**, incurring about 3× lower system-level area overhead while also achieving lower power consumption. This advantage is mainly due to Rogue's lighter streaming organization, while Mage pays the area cost of its GMP-compliant internal multi-bank SpM. At the same time, **Mage remains the best solution when maximizing raw *matmul* throughput is the main goal**, especially when reduced-precision vector execution can be exploited. Nonetheless, **Rogue remains highly competitive in normalized metrics**, often achieving higher area efficiency and strong accelerator-level energy efficiency.

Compared with state-of-the-art designs, Rogue *s* and *p* variants have been evaluated against accelerators for activation functions, whereas *g* implementations have been compared with both standard and DAE CGRAs. On integer approximations of activation functions, the comparison shows that Rogue does not aim to outperform fixed-function accelerators in absolute area or power. Rather, **Rogue's main strength lies in supporting a broader set of activations while still retaining competitive throughput and efficiency with respect to both fixed-function and reconfigurable accelerators for activation**. For instance, the best-performing Rogue variant, *p4x4_2*, exceeds NACU on both *Tanh* and *Sigmoid* throughput, while also remaining significantly more flexible. At the same time, fixed-function accelerators remain more area-efficient and lower-power than Rogue, since they are optimized for a single workload. Nevertheless, Rogue proves to be capable of obtaining throughput comparable to activation-specific accelerators, while preserving substantially higher flexibility and adaptability toward new activation kernels. Overall, the comparison highlights that Rogue occupies an intermediate design point, combining flexibility, competitive performance, and enough specialization to remain efficient across a broad set of integer activation approximations. Although it has been primarily designed for integer activation workloads, results obtained for *g* variants also show that **Rogue is competitive and, in some cases, preferable with respect to state-of-the-art DAE and standard CGRAs**. In particular, on the *matmul* kernel, Rogue delivers more than $2\times$ STRELA's throughput at the same frequency and technology node, while also achieving substantially higher area efficiency and better accelerator-level energy efficiency. Overall, the comparison shows that Rogue occupies a very competitive position within the current CGRA landscape, making it particularly attractive for designs in which hardware cost, compactness, and efficiency are primary constraints.

Chapter 6

Conclusions & Future Works

This thesis investigated how **Coarse-Grain Reconfigurable Arrays** (CGRAs) can be effectively employed to achieve energy-efficient acceleration for edge applications by exploiting the **Decoupled Access-Execute** (*DAE*) computational model and by explicitly targeting **Static Control Programs** (SCoPs) as the primary workload class. The motivations behind this work stem from the observation that, despite the growing trend in CGRA research toward broader code coverage and support for increasingly complex workloads, the large majority of kernels mapped onto and used to evaluate modern CGRAs still falls within the SCoP domain. At the same time, SCoPs encompass a large portion of the crucial workloads found in many of today's relevant application fields, such as Machine Learning, Digital Signal/Image Processing, Hardware Security, etc. Based on this insight, this thesis explored a complementary point in the design space: rather than prioritizing generality within the compute fabric, the proposed CGRAs were designed to maximize throughput, area efficiency, and energy efficiency on structured workloads through architectural specialization. Although this choice slightly restricts the application space, it allows the full potential of CGRAs to be unlocked for this highly relevant class of workloads.

The first proposed architecture, **Mage**, addressed this objective by combining a dedicated AGU driven by hardware loops, a GMP-compliant internal multi-bank Scratchpad Memory (SpM), and a fully static execution model targeting $II = 1$. In this organization, memory-address generation and static-control handling are removed from the Processing Element Array (PEA), while the internal data layout enables conflict-free accesses to the SpM and maximum-bandwidth data delivery

to the compute fabric. The results obtained through the design-space exploration confirm that this design direction is highly effective. In particular, Mage vector variants reach up to **7.8 GOPS**, **13.8 GOPS/mm^2**, and **345.6 MOPS/mW** on int8 `matmul`, outperforming the considered state-of-the-art standard and DAE CGRAs in throughput, area efficiency, and energy efficiency. Overall, Mage shows that a DAE CGRA tightly co-designed with its internal memory organization can fully exploit the regularity of SCoP workloads, achieving very high throughput together with strong area and energy efficiency. While the explicit focus on SCoPs allows Mage to significantly outperform recent standard CGRAs targeting broader code coverage, the tailored data layout for conflict-free memory accesses also enables it to surpass other recent DAE CGRAs.

The second proposed architecture, **Rogue**, explored a complementary point in the DAE CGRA design space. Rather than integrating a full internal memory subsystem, Rogue adopts a lighter streaming-oriented DAE organization based on a FIFO interface toward external Load-Store Units and, more importantly, the system DMA. In addition, Rogue implements a dedicated ISA that captures recurrent computational patterns found in integer approximations of activation functions, while still supporting conventional GeMM-style computation. The design-space exploration confirmed that this lighter DAE implementation remains highly effective in terms of throughput, while also incurring much lower system-level area overheads. For activation workloads, Rogue achieves throughput comparable to, or higher than, dedicated and reconfigurable activation accelerators, while retaining a higher degree of flexibility. In particular, it reaches up to about **93× speedup** on *Tanh* and about **48× energy savings** on *GELU* over software execution on the RISC-V CV32E40P CPU implementing the *XCVPULP* extensions, while incurring only limited system-level area overhead, ranging from **2.8%** to **12.2%**. At the same time, Rogue also proves effective on GeMM workloads, reaching up to **932 MOPS**, **67.8 MOPS/mW**, and **6522.6 MOPS/mm^2** on `matmul`, thus achieving comparable or superior results with respect to the considered state-of-the-art standard and DAE CGRAs. Overall, Rogue shows that a lightweight streaming-oriented DAE CGRA can still deliver substantial performance and efficiency gains on relevant edge workloads. Although not fully eliminating memory access conflicts, Rogue achieves substantial throughput while also incurring low system-level area overhead and reduced pressure on the system bus.

Overall, Mage and Rogue represent two complementary specializations of the DAE paradigm applied to CGRA design. Mage demonstrates that the DAE model can be exploited very aggressively when the main aim is prioritizing throughput while still delivering very high area/energy efficiency on SCoP kernels by exploiting an internal SpM and a conflict-free data layout. On the other hand, Rogue shows that the same DAE principle can also be employed for a lighter streaming-oriented architecture, capable of delivering significant acceleration benefits while substantially reducing system-level area overhead and pressure on the system bus.

More broadly, the results of this thesis support the overall conclusion that **the DAE model is particularly effective when CGRAs explicitly target SCoPs as their primary workload class**. Although this may appear to narrow the scope of CGRA applicability, it is in fact a highly relevant design direction from an application perspective. As a matter of fact, SCoPs represent a large portion of the kernels that are most commonly accelerated in many crucial edge domains, including Machine Learning, Signal and Image Processing, and Hardware Security. Therefore, explicitly designing CGRAs around this workload class should not be regarded as a limitation, but rather as a way to align the architecture with the computation patterns that most consistently and effectively benefit from hardware acceleration on a CGRA. In this sense, the main contribution of this thesis is not only the presentation of two concrete CGRA implementations, but also the demonstration that **SCoP-centered DAE specialization remains a highly relevant and promising direction for future edge CGRA design**.

Although the results achieved by Mage and Rogue are encouraging, several directions can still be explored as future work.

A particularly important direction concerns the development of a **compilation and mapping toolchain** for both architectures. This aspect is especially critical for Mage, whose execution model requires the scheduling phase to satisfy the GMP-compliant data layout constraints needed for the AGU to access memory without conflicts. Although all the SCoP workloads analyzed in this thesis were mapped manually without major difficulties, the availability of an automatic strategy capable of solving the scheduling problem together with the associated data-layout constraints would highly facilitate the mapping of SCoP kernels, especially when aggressive loop unrolling is applied. A similar effort would also be beneficial for Rogue, although its mapping constraints are not as stringent as Mage's. Overall, a

compiler toolchain would make both architectures significantly more practical, while also enabling broader architectural exploration through software support.

A second future research direction concerns **architectural enhancements** for both Rogue and Mage, as well as possible *combinations* between the two designs.

One possible enhancement for Rogue would be to import **vector computation capabilities from Mage**. In particular, extending Rogue with packed-SIMD execution similar to that of Mage's vector PEs could significantly improve performance on GeMM workloads, while also benefiting reduced-precision activation workloads based on `int8` and `int16` data that do not require full `int32` computation. Several operators already developed for Mage could be integrated directly into Rogue, since they are fully pipelined, while more compact alternatives could also be explored to better investigate the latency-area trade-off space.

At the same time, another promising extension would be to **replace Mage's PEA with Rogue's PEA organization and activation-oriented ISA**. This would make it possible to evaluate a memory-equipped DAE CGRA capable of supporting packed-SIMD computation for both GeMM and integer activation kernels, while still benefiting from tailored data placement and conflict-free memory accesses. A further direction, closely related to this idea, would be to **evolve Mage toward multi-kernel execution in an application-driven perspective**. In its current implementation, Mage is designed to execute a single statically configured kernel. A possible evolution would be to integrate an internal configuration memory capable of storing multiple kernels, together with a control unit able to track their locations, orchestrate their execution meaningfully, and dispatch the appropriate configuration to the different Mage subsystems. This would enable the execution of concatenated kernels without host intervention between phases. Such an extension would be particularly valuable for ML pipelines, enabling combinations such as *matmul* + *softmax*, *matmul* + *GELU*, and similar operator sequences that frequently occur in Transformers and other neural-network models.

Finally, an important system-level future research direction is to strengthen the evaluation of Mage and Rogue, and more generally of the proposed DAE design approach, on **complete ML pipelines**. As for Mage, an initial evaluation has already been carried out on a TCN pipeline, but further investigations could consider variants equipped with larger SpMs in order to reduce the need for tiling. As for Rogue, it could be evaluated as a self-contained streaming accelerator in which both GeMM

and activation kernels are mapped onto the same architecture, so as to assess the benefits and limitations of an all-in-one solution. Alternatively, a heterogeneous setup could be explored, where a memory-equipped accelerator (such as Mage) executes GeMM stages, while Rogue applies activation functions directly during the DMA transfers that are already required for tiling and data movement. This second approach is particularly appealing because it would exploit Rogue exactly where its streaming nature is most advantageous, namely by embedding computation into data transfers that would already take place within the system. In addition, in order to fully evaluate the enabled benefits, energy improvements should be analyzed. As for Mage, power consumption can be evaluated on post-synthesis or post-layout netlists as well as through measures on FPGA implementations. On the other hand, Rogue can also be evaluated on the polHEEPo SoC.

Overall, these future directions suggest that the work presented in this thesis can be extended along both software and hardware axes. On the software side, automated compilation and mapping are essential to unlock the full potential of the proposed architectures. On the hardware side, the template-based nature of Mage and Rogue opens the way to richer hybridizations, wider ISA support, vector extensions, and multi-kernel orchestration mechanisms. Taken together, these developments could further strengthen the role of SCoP-oriented DAE CGRAs as practical and high-efficiency accelerators for edge systems.

References

[1] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. CASCADE: High Throughput Data Streaming via Decoupled Access-Execute CGRA. *ACM Trans. Embed. Comput. Syst.*, 18(5s):50:1–50:26, October 2019.

[2] Pasquale Davide Schiavone, Simone Machetti, Miguel Peón-Quirós, Jose Miranda, Benoît Denkinger, Thomas Christoph Müller, Ruben Rodríguez, Saverio Nasturzio, and David Atienza Alonso. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, pages 379–380, New York, NY, USA, August 2023. Association for Computing Machinery.

[3] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators, March 2024. arXiv:2401.05548 [cs].

[4] Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, FPGA '14, pages 199–208, New York, NY, USA, February 2014. Association for Computing Machinery.

[5] J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 12–21, April 1997.

[6] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143, Monterey California USA, February 1999. ACM.

[7] Gerard J. M. Smit, André B. J. Kokkeler, Pascal T. Wolkotte, Philip K. F. Hölzenspies, Marcel D. Van De Burgwal, and Paul M. Heysters. The Chameleon Architecture for Streaming DSP Applications. *EURASIP Journal on Embedded Systems*, 2007:1–10, 2007.

[8] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Proceedings 20th Anniversary Conference on Advanced Research in VLSI*, pages 23–40, March 1999. ISSN: 1522-869X.

[9] Mirsky and DeHon. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, April 1996.

[10] S.C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. PipeRench: a reconfigurable architecture and compiler. *Computer*, 33(4):70–77, April 2000.

[11] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[12] Cheng Tan, Deepak Patil, Antonino Tumeo, Gabriel Weisz, Steve Reinhardt, and Jeff Zhang. VecPAC: A Vectorizable and Precision-Aware CGRA. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, October 2023. ISSN: 1558-2434.

[13] Omid Akbari, Mehdi Kamal, Ali Afzali-Kusha, Massoud Pedram, and Muhammad Shafique. X-CGRA: An Energy-Efficient Approximate Coarse-Grained Reconfigurable Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2558–2571, October 2020.

[14] Marcelo Brandalero, Antonio Carlos S. Beck, Luigi Carro, and Muhammad Shafique. Approximate On-The-Fly Coarse-Grained Reconfigurable Acceleration for General-Purpose Applications. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018.

[15] Zahra Ebrahimi and Akash Kumar. BioCare: An Energy-Efficient CGRA for Bio-Signal Processing at the Edge. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2021. ISSN: 2158-1525.

[16] Jiajun Qin, Tianhua Xia, Cheng Tan, Jeff Zhang, and Sai Qian Zhang. PICACHU: Plug-In CGRA Handling Upcoming Nonlinear Operations in LLMs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 845–861, Rotterdam Netherlands, March 2025. ACM.

[17] Rohit Prasad. NX-CGRA: A Programmable Hardware Accelerator for Core Transformer Algorithms on Edge Devices, November 2025. arXiv:2511.17235 [cs].

[18] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. Ultra-Elastic CGRAs for Irregular Loop Specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 412–425, February 2021. ISSN: 2378-203X.

[19] Barry de Bruin, Kanishkan Vadivel, Mark Wijtvliet, Pekka Jääskeläinen, and Henk Corporaal. R-Blocks: an Energy-Efficient, Flexible, and Programmable CGRA. *ACM Trans. Reconfigurable Technol. Syst.*, 17(2):34:1–34:34, May 2024.

[20] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 546–564, October 2022.

[21] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1027–1040, June 2021. ISSN: 2575-713X.

[22] Satyajit Das, Kevin J. M. Martin, Davide Rossi, Philippe Coussy, and Luca Benini. An Energy-Efficient Integrated Programmable Array Accelerator and Compilation Flow for Near-Sensor Ultralow Power Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(6):1095–1108, June 2019.

[23] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 1–6, New York, NY, USA, June 2017. Association for Computing Machinery.

[24] Kyuseung Han, Seongsik Park, and Kiyoung Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1367–1372, March 2012. ISSN: 1558-1101.

[25] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. Branch-aware loop mapping on CGRAs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014. ISSN: 0738-100X.

[26] Daniel Vazquez, Jose Miranda, Alfonso Rodriguez, Andres Otero, Pascuale Davide Schiavone, and David Atienza. STRELA: STReaming ELAstic CGRA Accelerator for Embedded Systems, April 2024. arXiv:2404.12503 [cs].

[27] Jinyi Deng, Xinru Tang, Jiahao Zhang, Yuxuan Li, Linyun Zhang, Boxiao Han, Hongjun He, Fengbin Tu, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin.

Towards Efficient Control Flow Handling in Spatial Architecture via Architecting the Control Flow Plane. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1395–1408, October 2023. arXiv:2307.02847 [cs].

[28] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–429, June 2017.

[29] Boma Adhi, Carlos Cortes, Yiyu Tan, Takuya Kojima, Artur Podobas, and Kentaro Sano. Exploration Framework for Synthesizable CGRAs Targeting HPC: Initial Design and Evaluation. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 639–646, May 2022.

[30] Boma Adhi, Carlos Cortes, Emanuele Del Sozzo, Tomohiro Ueno, Yiyu Tan, Takuya Kojima, Artur Podobas, and Kentaro Sano. Less for More: Reducing Intra-CGRA Connectivity for Higher Performance and Efficiency in HPC. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 452–459, May 2023.

[31] Emanuele Del Sozzo, Xinyuan Wang, Boma Adhi, Carlos Cortes, Jason Anderson, and Kentaro Sano. Exploration of Trade-offs Between General-Purpose and Specialized Processing Elements in HPC-Oriented CGRA. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 668–680, May 2024. ISSN: 1530-2075.

[32] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. REVAMP: a systematic framework for heterogeneous CGRA realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 918–932, New York, NY, USA, February 2022. Association for Computing Machinery.

[33] Jingyuan Li, Yunhui Qiu, Guowei Zhu, Qilong Zhu, Wenbo Yin, and Lingli Wang. THRAM: A Template-based Heterogeneous CGRA Modeling Framework Supporting Fast DSE. In *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2023. ISSN: 2158-1525.

[34] Rubén Rodríguez Álvarez, Benoît Denkinger, Juan Sapriza, José Miranda Calero, Giovanni Ansaloni, and David Atienza Alonso. An Open-Hardware Coarse-Grained Reconfigurable Array for Edge Computing. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, CF '23, pages 391–392, New York, NY, USA, August 2023. Association for Computing Machinery.

[35] Satyajit Das, Kevin J. M. Martin, Philippe Coussy, Davide Rossi, and Luca Benini. Efficient mapping of CDFG onto coarse-grained reconfigurable array architectures. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132, January 2017. ISSN: 2153-697X.

[36] Shouyi Yin, Xianqing Yao, Dajiang Liu, Leibo Liu, and Shaojun Wei. Memory-Aware Loop Mapping on Coarse-Grained Reconfigurable Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1895–1908, May 2016.

[37] Shouyi Yin, Xianqing Yao, Tianyi Lu, Dajiang Liu, Jiangyuan Gu, Leibo Liu, and Shaojun Wei. Conflict-Free Loop Mapping for Coarse-Grained Reconfigurable Architecture with Multi-Bank Memory. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2471–2485, September 2017.

[38] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 16(4):42:1–42:27, October 2011.

[39] Shouyi Yin, Xianqing Yao, Tianyi Lu, Leibo Liu, and Shaojun Wei. Joint loop mapping and data placement for coarse-grained reconfigurable architecture with multi-bank memory. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, November 2016. ISSN: 1558-2434.

[40] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402, June 2017.

[41] James E. Smith. Decoupled access/execute computer architectures. *SIGARCH Comput. Archit. News*, 10(3):112–119, April 1982.

[42] Lennart Bamberg, Filippo Minnella, Roberto Bosio, Fabrizio Ottati, Yuebin Wang, Jongmin Lee, Luciano Lavagno, and Adam Fuks. eIQ Neutron: Redefining Edge-AI Inference with Integrated NPU and Compiler Innovations, September 2025. arXiv:2509.14388 [cs].

[43] George Charitopoulos, Charalampos Vatsolakis, Grigorios Chrysos, and Dionisios N. Pnevmatikatos. A decoupled access-execute architecture for reconfigurable accelerators. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 244–247, Ischia Italy, May 2018. ACM.

[44] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (RSVP/spl trade/). In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 141–150, December 2003.

[45] Tao Chen and G. Edward Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, October 2016.

[46] Jeckson Dellagostin Souza, Luigi Carro, Mateus Beck Rutzig, and Antonio Carlos Schneider Beck. A reconfigurable heterogeneous multicore with a homogeneous ISA. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1598–1603, March 2016. ISSN: 1558-1101.

[47] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. *IEEE Micro*, 32(5):38–51, September 2012.

[48] Dani Voitsechov, Oron Port, and Yoav Etsion. Inter-Thread Communication in Multithreaded, Reconfigurable Coarse-Grain Arrays. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 42–54, October 2018.

[49] Tao Chen, Shreesha Srinath, Christopher Batten, and G. Edward Suh. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 55–67, October 2018.

[50] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.*, 52(6):118:1–118:39, October 2019.

[51] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 235–244, July 2016.

[52] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective. *IEEE Access*, 8:146719–146743, 2020.

[53] Vu Trung Duong Le, Hoai Luan Pham, Thi Hong Tran, Van Duy Tran, Tuan Hai Vu, and Yasuhiko Nakashima. CTFE: A High-Efficient Heterogeneous Cryptographic CGRA for Diverse Security Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(5):1793–1806, May 2025.

[54] Miaomiao Jiang, Yilan Zhu, Honghui You, Cheng Tan, Zhaoying Li, Jiming Xu, and Lei Ju. FHE-CGRA: Enable Efficient Acceleration of Fully Homomorphic Encryption on CGRAs. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, DAC '24, pages 1–6, New York, NY, USA, November 2024. Association for Computing Machinery.

[55] Dan Wu, Peng Chen*, Thilini Kaushalya Bandara, Zhaoying Li, and Tulika Mitra. Flip: Data-centric Edge CGRA Accelerator. *ACM Trans. Des. Autom. Electron. Syst.*, 29(1):22:1–22:25, December 2023.

[56] Luca Zulberti, Matteo Monopoli, Pietro Nannipieri, Silvia Moranti, Geert Thys, and Luca Fanucci. Efficient Coarse-Grained Reconfigurable Array architecture for machine learning applications in space using DARE65T library platform. *Microprocess. Microsyst.*, 113(C), March 2025.

[57] Ensieh Aliagha, Najdet Charaf, Nitin Krishna Venkatesan, and Diana Göhringer. DA-CGRA: Domain-Aware Heterogeneous Coarse-Grained Reconfigurable Architecture for the Edge. In *2024 27th Euromicro Conference on Digital System Design (DSD)*, pages 410–417, August 2024. ISSN: 2771-2508.

[58] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, September 2017.

[59] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, October 2017.

[60] Tommaso Terzano, Luigi Giuffrida, Juan Sapriza, Pasquale Davide Schiavone, Guido Masera, David Atienza, Luciano Lavagno, and Maurizio Martina. Just TestIt! An SBST Approach To Automate System-Integration Testing, April 2025. arXiv:2504.07555 [cs].

[61] Thomas Benz, Michael Rogenmoser, Paul Scheffler, Samuel Riedel, Alessandro Ottaviano, Andreas Kurth, Torsten Hoefler, and Luca Benini. A High-Performance, Energy-Efficient Modular DMA Engine Architecture. *IEEE Transactions on Computers*, 73(1):263–277, January 2024.

[62] Alessandro Varaldi, Alessio Naclerio, Fabrizio Riente, Maurizio Zamboni, Mariagrazia Graziano, and Marco Vacca. Optimizing TCN Inference: A Hardware-Software Co-Design Approach with CGRA Acceleration. In *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, volume 1, pages 1–6, July 2025. ISSN: 2159-3477.

[63] Satyabrata Sarangi and Bevan Baas. DeepScaleTool: A Tool for the Accurate Estimation of Technology Scaling in the Deep-Submicron Era. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2021. ISSN: 2158-1525.

[64] Alessio Naclerio, Fabrizio Riente, Giovanna Turvani, Marco Vacca, Maurizio Zamboni, and Mariagrazia Graziano. Mage: a Decoupled Access-Execute CGRA tailored for Static Control Applications. In *2025 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2025. ISSN: 2158-1525.

[65] Renzo Andri, Enrico Reggiani, and Lukas Cavigelli. Flex-SFU: Activation Function Acceleration with Non-Uniform Piecewise Approximation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2025.

[66] Andrea Belano, Yvan Tortorella, Angelo Garofalo, Luca Benini, Davide Rossi, and Francesco Conti. A Flexible Template for Edge Generative AI With High-Accuracy Accelerated Softmax and GELU. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 15(2):200–216, June 2025.

[67] Run Wang, Gamze Islamoglu, Andrea Belano, Viviane Potocnik, Francesco Conti, Angelo Garofalo, and Luca Benini. VEXP: A Low-Cost RISC-V ISA Extension for Accelerated Softmax Computation in Transformers. In *2025 IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, pages 37–44, May 2025. ISSN: 2576-2265.

[68] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. I-BERT: Integer-only BERT Quantization, June 2021. arXiv:2101.01321 [cs].

[69] Zhikai Li and Qingyi Gu. I-ViT: Integer-only Quantization for Efficient Vision Transformer Inference. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 17019–17029, October 2023. ISSN: 2380-7504.

[70] Jacob R. Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers. In *Proceedings of the 58th Annual ACM/IEEE Design Automation Conference*, DAC '21, pages 469–474, San Francisco, California, United States, May 2022. IEEE Press.

[71] Karl Leboeuf, Ashkan Hosseinzadeh Namin, Roberto Muscedere, Huapeng Wu, and Majid Ahmadi. High Speed VLSI Implementation of the Hyperbolic Tangent Sigmoid Function. In *2008 Third International Conference on Convergence and Hybrid Information Technology*, volume 1, pages 1070–1073, November 2008.

[72] Babak Zamanlooy and Mitra Mirhassani. Efficient VLSI Implementation of Neural Networks With Hyperbolic Tangent Activation Function. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(1):39–48, January 2014.

[73] Andrea Belano, Yvan Tortorella, Angelo Garofalo, Luca Benini, Davide Rossi, and Francesco Conti. SoftEx: A Low Power and Flexible Softmax Accelerator

with Fast Approximate Exponentiation. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–2, March 2025. ISSN: 1558-1101.

[74] Gamze Islamoglu, Moritz Scherer, Gianna Paulin, Tim Fischer, Victor J.B. Jung, Angelo Garofalo, and Luca Benini. ITA: An Energy-Efficient Attention and Softmax Accelerator for Quantized Transformers. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, August 2023.

[75] Christodoulos Peltekis, Kosmas Alexandridis, and Giorgos Dimitrakopoulos. Reusing Softmax Hardware Unit for GELU Computation in Transformers. In *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*, pages 159–163, April 2024. ISSN: 2834-9857.

[76] Kangjoon Choi, Sungho Kim, Jeongmin Kim, and In-Cheol Park. Hardware-Friendly Approximation for Swish Activation and Its Implementation. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 71(10):4516–4520, October 2024.

[77] Yusheng Xie, Alex Noel Joseph Raj, Zhendong Hu, Shaohaohan Huang, Zhun Fan, and Miroslav Joler. A Twofold Lookup Table Architecture for Efficient Approximation of Activation Functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(12):2540–2550, December 2020.

[78] Ye Lin, Yanyang Li, Tengbo Liu, Tong Xiao, Tongran Liu, and Jingbo Zhu. Towards fully 8-bit integer inference for the transformer model. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI'20, pages 3759–3765, Yokohama, Yokohama, Japan, January 2021.

[79] Yue Li, Wei Cao, Xuegong Zhou, and Lingli Wang. A Low-Cost Reconfigurable Nonlinear Core for Embedded DNN Applications. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 35–38, December 2020.

[80] Guido Baccelli, Dimitrios Stathis, Ahmed Hemani, and Maurizio Martina. NACU: A Non-Linear Arithmetic Unit for Neural Networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, July 2020. ISSN: 0738-100X.

[81] Haonan Du, Chenyi Wen, Zhengrui Chen, Li Zhang, Qi Sun, Zheyu Yan, and Cheng Zhuo. Algorithm-Hardware Co-Design of a Unified Accelerator for Non-Linear Functions in Transformers. In *2025 Design, Automation & Test in Europe Conference (DATE)*, pages 1–7, March 2025. ISSN: 1558-1101.

[82] Xing Hu, Yuan Cheng, Dawei Yang, Zhihang Yuan, Jiangyong Yu, Chen Xu, and Sifan Zhou. I-LLM: Efficient Integer-Only Inference for Fully-Quantized Low-Bit Large Language Models, June 2024. arXiv:2405.17849 [cs].

[83] Antonio Pullini. *ETH Library Design of Energy Efficient Microcontrollers*. PhD thesis, ETH.

[84] Juan Sapriza, Beatrice Grassano, Alessio Naclerio, Filippo Quadri, Tommaso Terzano, David Mallasén, Davide Schiavone, Robin Leplae, Jérémie Moullet, Alexandre Levisse, Christoph Müller, Mariagrazia Graziano, Matías Miguez, and David Atienza. HEEPidermis: a versatile SoC for BioZ recording, September 2025.

[85] Jonah Van Assche and Georges Gielen. A 10.4-ENOB 0.92-5.38 W Event-Driven Level-Crossing ADC with Adaptive Clocking for Time-Sparse Edge Applications. In *ESSCIRC 2022- IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, pages 261–264, September 2022.