Politecnico di Torino

Master's Degree in Electronic Engineering

Master's Degree Thesis

Development of an Advanced Configurable DMA System for Edge AI Accelerators in a 16nm Low Power RISC-V Microcontroller

Supervisors

Prof. Luciano Lavagno

Prof. David Atienza

PhD. Davide Schiavone

Eng. Luigi Giuffrida

Candidate

Tommaso Terzano

Academic Year 2023 - 2024

Summary

Artificial Intelligence has driven technological innovation over the past decade, impacting various fields such as image recognition, natural language processing, autonomous driving, and complex system modeling. **Edge AI** offers a promising alternative to traditional *cloud-based* solutions by processing data directly on devices. This enables real-time operations, improves privacy and data integrity, and allows edge devices to acquire data from multiple sensors simultaneously, enhancing their ability to extract meaningful information from the environment.

Microcontrollers (**MCUs**) are a popular choice for edge devices due to their versatility and short time-to-market. However, they face challenges such as limited computational resources and the need for low-power consumption, making it crucial to optimize their design for deployment in edge AI applications.

In this context, **X-HEEP** is a configurable, open-source RISC-V 32-bit MCU developed at the *Embedded Systems Laboratory* (ESL) at EPFL. The main goal of this thesis is to *enhance X-HEEP* by introducing new features that address one of the key challenges in deploying Edge AI models: the bottleneck in data movement and manipulation, especially on low-power platforms.

To handle the large amount of data generated by multiple sensors in edge computing, the **DMA system** of X-HEEP was extended with a configurable number of **channels**, which are connected to the system bus via customizable master ports. This extension allows memory bandwidth to scale with the number of channels, except in cases of bus conflicts. Additionally, power consumption can be reduced through **clock-gating**, which can be applied to individual unused DMA channels by utilizing the existing power management system.

Edge AI, like cloud-based AI, relies heavily on neural networks, particularly on **GEMM** (General Matrix Multiply) operations, which are fundamental to deep learning. To meet performance, timing, and energy efficiency requirements, specialized hardware accelerators such as multi-core clusters, in/near-memory macros, and systolic arrays have been developed. At *ESL*, for example, two innovative low-power architectures, **Caesar** and **Carus**, were designed to perform both memory operations and computations.

A key operation in convolutional neural networks (CNNs) is the transformation of input tensors and filters into a matrix form using a technique called **im2col** (image-to-column). This transformation simplifies convolutions by converting them into matrix multiplications, which can then be accelerated by specialized GEMM hardware. However, when handled by the CPU, the im2col operation can become time-intensive, potentially negating the benefits of the accelerator and turning data transfer into a performance bottleneck.

To address this issue, the thesis extends X-HEEP's DMA system with **2D capabilities** and the ability to perform on-the-fly data transformations such as *transpositions*, *zero-padding*, and *sign extension*. These enhancements allow the DMA to manage tensor reshaping tasks more efficiently, reducing the overall CPU load.

Furthermore, the thesis introduces the **Always On Peripheral Bus** (**AOPB**), which exposes the Always-On peripherals, including the DMA subsystem, via a simplified OBI protocol. This allows external accelerators to access pre-existing peripherals directly, enabling complex data manipulations with minimal area overhead, particularly for *DMA-based accelerators*.

To validate this interface, the thesis proposes a **Smart Peripheral Controller** (**SPC**) for the **im2col** transformation, designed to optimize tensor reshaping for GEMM operations. By leveraging the new DMA system's features, the SPC minimizes data movement overhead while keeping the area requirements low. Through extensive testing, the im2col SPC achieved a **6.1x performance improvement** compared to CPU-based im2col routines, while consuming **2% less power**, resulting in a **6.1x increase** in energy efficiency.

All the designs introduced in this thesis will be integrated into **HEEPatia**, a 16nm silicon implementation of X-HEEP aimed at low-power edge-computing applications. To ensure the correctness and performance of these designs, a comprehensive verification campaign was conducted. For this purpose, a software-based platform called **VerifHEEP** was developed. This Python library, tailored for X-HEEP, enables users to rapidly create a **SBST verification environment** that targets both simulation tools and the PYNQ-Z2 FPGA board. VerifHEEP was used to verify the DMA system and the im2col SPC, ensuring that the designs function as expected and deliver the intended performance improvements.

A significant objective of this thesis was to make all components easy to understand, use, extend, and adapt for future developers. To support this goal, extensive **documentation** was created for both the *DMA subsystem* and the *VerifHEEP* library, along with thoroughly commented hardware abstraction layers (HALs) and usage examples.

The impact of this work is already evident, as several **DMA-based projects** utilizing these new features have started development at both *ESL* and Polito's *VLSI* group before the completion of this thesis. This early adoption highlights the importance of these contributions in addressing data movement challenges and

advancing research in **Edge AI**.

Acknowledgements

A mio nonno Mario, per essere il mio più grande fan e per avermi fatto capire come affrontare la vita con ironia.
A mia nonna Giuliana, per l'infinito affetto.
A mio nonno Giancarlo, per avermi trasmesso con passione la storia della mia famiglia.
A mia nonna Letizia, per i puntualissimi "in bocca al lupo" prima di ogni esame.
Ai miei genitori, Raffaella e Massimiliano, per avermi insegnato a pensare.
A mio fratello Matteo, per essere sempre al mio fianco.
Ai miei amici, per aver condiviso con me mille avventure ed essere stati un grande punto di riferimento.
A Luigi, per l'incrollabile sostegno.

Table of Contents

List of Tables X

List of Figures XI

1 Background Foundations 1

1.1 Machine Learning Overview 1

1.1.1 Potential and Limits of Modern Artificial Intelligence 1

1.1.2 Types of Problems Addressed by Machine Learning 2

1.1.3 Core Computational Mechanisms in ML 21

1.2 Deploying ML Models on Edge Devices 27

1.2.1 Edge AI vs Cloud AI 27

1.2.2 Mitigating Critical Challenges in Edge AI 28

1.2.3 X-HEEP: eXtendable Heterogeneous Energy-Efficient Platform 29

2 Development 35

2.1 Data Movement in X-HEEP: the OBI Protocol 35

2.2 Designing a Multichannel DMA System for X-HEEP 37

2.2.1 The Challenge of Handling Multiple Data Streams in Edge AI 37

2.2.2 Structural Description 37

2.3 Extending X-HEEP's DMA System with Advanced Transactions Capabilities 42

2.3.1 Complex Data Manipulation in GEMM Accelerators 42

2.3.2 Structural Description 43

2.3.3 Functional Description 51

2.4 Developing the Always-On Peripheral Bus to Support DMA-Based Accelerators 58

2.4.1 DMA-Based Accelerators to Reduce CPU Utilization 58

2.4.2 Structural Description 59

2.5 Developing an Im2col SPC to Optimize On-Edge CNNs 63

2.5.1 *NCHW* vs. *NHWC*: Choosing the Optimal Format 63
2.5.2 DMA-based vs Specialized Im2col Accelerators 65
2.5.3 Structural Description 66
2.5.4 Parameter FSM 73
2.5.5 Register Interface Controller 74
2.6 Developing VerifHEEP, a Software-Based Self-Test Functional Verification Library for X-HEEP 75
2.6.1 Advantages of Software-Based Self-Test Techniques 75
2.6.2 Integrating the new designs into a 16nm silicon implementation of HEEP 76
2.6.3 VerifHEEP Library Structure 77
2.6.4 Developing a Verification Environment for the Im2col SPC using VerifHEEP 80

3 Results 83
3.1 Analysis of Performance and Area Overhead of the new DMA System 83
3.1.1 Test Set-Up 83
3.1.2 Performance Improvement with respect to CPU Routines 84
3.1.3 Area Occupation Analysis 86
3.2 Analysis of Performance, Power Consumption and Area Overhead of the Im2col SPC 87
3.2.1 Performance Improvement with respect to CPU-based Routines 88
3.2.2 Power Consumption Analysis 90
3.2.3 Area Occupation Analysis 92

4 Future Improvements and Conclusions 95
4.1 Optimizing Memory Accesses for Non-Word Datatypes 95
4.2 Further Optimizing Convolution Computation Using 3D DMA and Stream Accelerators 97
4.3 Conclusion 100

Bibliography 103

List of Tables

3.1 Area footprint of a single DMA channel 87
3.2 Comparison between the new and old DMA channel 87
3.3 Power consumption of different units and their total contribution across different tests 91
3.4 Estimation of energy efficency gains, considering the 100 MHz frequency used in the test 92
3.5 Area footprint of the im2col SPC 93

List of Figures

1.1 Four categories of problems addressed by ML 2
1.2 An example of a fourth-order linear regression model used to predict house prices based on service quality and crime rates in the area . . 3
1.3 Bias and variance contributions to the generalization error 5
1.4 Behaviour of the gradient descent optimization algorithm with three different learning rates 7
1.5 Average loss for a binary classification model trained once with *L2* and once with *Cross-Entropy* as loss functions, averaged over 500 runs 9
1.6 Decision Boundary model obtained using SVMs, with data randomly generated in two clusters 9
1.7 Decision Trees model with four classes 10
1.8 Structure of the Decision Trees model after training 10
1.9 AlexNet is a popular deep convolutional neural network architecture with 8 layers, designed for image classification, featuring stacked convolutional layers, ReLU activations, max-pooling, and dropout to improve performance and prevent overfitting [4] 11
1.10 Sigmoid function 13
1.11 Hyperbolic tangent function 13
1.12 ReLU function 13
1.13 Example of a Neural Network with two hidden layers 13
1.14 Example of an intermediate step of the training of a Neural Network with a Drop Out regularizer applied 14
1.15 Example of a single Convolutional Neural Network layer 15
1.16 Example of a Max Pooling with 3x3 filters and a 3 element stride . 16
1.17 The inception module in the GoogLeNet architecture is placed on the left side of the network, where it bypasses the main blocks . . . 16
1.18 Progression of K-Means clustering across three iterations, showing the iterative refinement of cluster centers as they converge towards their final positions 17
1.19 Simple two-dimensional example of Principal Component Analysis. 18

1.20 Accuracy obtained using different feature dimensions, as extracted from the 2017 paper *'Using Deep Autoencoders for Facial Expression Recognition'* [6]. This paper investigates the effectiveness of deep autoencoders for feature selection and dimension reduction in facial expression recognition, demonstrating that the features extracted from the stacked autoencoder outperform other state-of-the-art techniques, including the standard PCA, as shown in this graph. 19
1.21 Example of a Reinforcement Learning model training for ETH's quadrupedal system *ANYmal*, by Hwangbo et al [7]. 20
1.22 Two-dimensional example of a convolution computation, step one . 23
1.23 Two-dimensional example of a convolution computation, step two . 23
1.24 Example of *im2col* computation 25
1.25 Comparison between NHWC and NCHW execution times on Intel i9 CPU 26
1.26 Caesar Structure 29
1.27 Carus Structure 29
1.28 X-HEEP structure overview 30
1.29 MetaWearS addresses the challenges of a wearable system lifecyle, specifically the need for large training datasets and effective model updates 31
1.30 HEEPocrates layout, silicon photo, and physical realization (on a Swiss 5-cent franc coin) [13] 32
1.31 Structure of the VersaSens platform 33

2.1 Example of a basic OBI transaction 35
2.2 Structure of a general Edge AI system, with a variety of sensors connected to the edge device 37
2.3 Structural Overview of the proposed Multichannel DMA System . . 38
2.4 DMA-to-bus connection, composed with two 2-to-1 OBI crossbars . 40
2.5 DMA-to-bus connection, composed with a 3-to-1 OBI crossbar and a single direct connection 40
2.6 Memory map scheme of the DMA System. 41
2.7 Structural Overview of the Proposed DMA System with Advanced Transaction Channels 42
2.8 Structural Overview of the New DMA Channel 43
2.9 Details on the *READ FSM* main signals, including counters and source registers. On the top, the most critical OBI signals, while on the bottom the values of a few configuration registers. More details on the meaning behind these parameters can be found in the functional description section [**2.3.3**]. 44

2.10 4x4 matrix and the extracted 2x2 sub-matrix. 45
2.11 Details on the *READ FSM* main signals during a matrix transposition. Note the changes of the source address values with respect to figure **2.9**. 45
2.12 4x4 matrix and the extracted 2x2 sub-matrix, transposed. 46
2.13 A section of a Verilator simulation focusing on the main signals of the *PADDING FSM*. The FSM state progression is highlighted in orange, which determines the value of *pad_on*, setting the *WRITE FIFO* input to 0. 47
2.14 Details on the *READ FSM* main signals in a 1D transaction copying *bytes*. 49
2.15 4x4 matrix and the extracted 2x2 sub-matrix. 52
2.16 Padding example of a 3x2 matrix, explaining the meaning of *"top"*, *"bottom"*, *"left"* and *"right"* padding in the DMA System. 54
2.17 Padding example of a 2x2 matrix extracted from a 4x4 matrix, with 1 layer of zero padding on top and on the left. 54
2.18 Structural Overview of the AOPB embedded into the Always-On Peripheral System. 58
2.19 Structural Overview of the Always-On Peripheral Bus. 59
2.20 This is a representation of a 3x3x3 tensor, flattened in *NCHW* format on the right and *NHWC* format on the left. The dimensions are highlighted, and the channels are distinguished by different colors. This visualization demonstrates that the two formats can be easily interchanged through a simple transposition. 63
2.21 Initial steps of the im2col reshaping transformation of an *NCHW* tensor. 64
2.22 Initial steps of the im2col reshaping transformation of an *NHWC* tensor. 64
2.23 SPOT architecture overview. 65
2.24 Structural overview of the *im2col* Smart Peripheral Controller. . . . 66
2.25 Example of a non-aligned filter slide. 70
2.26 Overview of a SBST Verification Set-Up. 75
2.27 Screenshot of the Im2col SPC verification environment interface, built using VerifHEEP. 80

3.1 Performance comparison of a matrix copy operation between the CPU and DMA implementations. The y-axis shows the number of cycles required to perform the copy, while the x-axis represents the size of the output matrix, including the applied zero padding. The comparison highlights the efficiency of the DMA over the CPU, especially for larger matrix sizes. 84

3.2 Performance comparison of a matrix copy operation without padding. 85
3.3 Performance comparison of a matrix copy operation with a 2-wide padding applied to every border of the input matrix. 86
3.4 Performance comparison of an *im2col* reshaping transformation, performed using the XuanTie C implementation, its DMA mapped version and finally using the *im2col SPC*. The y-axis shows the number of cycles required to perform the operation, while the x-axis represents the size of the output matrix, including the applied zero padding. The comparison highlights the efficiency of the DMA over the CPU, especially for larger matrix sizes, and the additional advantage of the im2col SPC. 88
3.5 Performance comparison of an im2col reshaping operation without padding 89
3.6 Performance comparison of an im2col reshaping operation with a 2-wide padding applied to every edge of the input tensor 90
3.7 Graphical representation of the power consumption contributions for each type of test 92

4.1 Updated DMA channel scheme with a new stage, the Buffer Stage, which implements the data buffer optimization for non-word transactions 96
4.2 General scheme of a double input, single output stream accelerator 98
4.3 Scheme of a stream accelerator integrated in *X-HEEP* along with the *im2col SPC* 99

Chapter 1

Background Foundations

“Artificial intelligence is the science and engineering of making computers behave in ways that, until recently, we thought required human intelligence.”

Andrew Moore, Dean of Computer Science at Carnegie Mellon University. [1]

1.1 Machine Learning Overview

1.1.1 Potential and Limits of Modern Artificial Intelligence

Artificial Intelligence (AI) refers to the field of computer science that focuses on creating machines and systems capable of performing tasks that typically require human intelligence, like understanding natural language, recognizing patterns and making complex decisions.

AI systems can range from simple rule-based algorithms to complex neural networks with billions of parameters, and they find application across a wide spectrum of fields, from autonomous driving to advanced biomedical imaging.

The concept of AI was first articulated at the Dartmouth Conference in the summer of 1956. The goal of the conference was to explore the possibility of creating machines that could simulate human intelligence. It was during this event that the term *"Artificial Intelligence"* was coined by John McCarthy [2].

However, the past decade has witnessed a remarkable surge in interest in the field of AI, driven by significant advancements in computational power and the emergence of *"big data"*, often referred to as *"the new oil"* due to its immense value in today’s digital economy. In AI, data play a crucial role, as they are the foundational element required to train and refine algorithms.

Within the broad landscape of Artificial Intelligence, *Machine Learning* stands out as a highly promising approach. There are numerous complex problems, such as pattern recognition, human speech comprehension, and image generation, that are exceedingly difficult, if not impossible, to solve by explicitly programming the correct behavior.

The machine learning approach allows the developer to define the overall structure of the algorithm while enabling the system to autonomously learn and optimize its parameters during the training phase, and sometime even during inference.

Despite these advancements, modern AI systems are not without their limitations. One of the most significant challenges is the immense computational power required to train sophisticated models, often necessitating specialized hardware like GPUs and TPUs. Furthermore, the training process demands enormous datasets, which are not always readily available or easily accessible. In addition to these constraints, the power consumption during the inference phase can be substantial, particularly in real-time or mobile applications, where energy efficiency is crucial.

As a potential solution to these challenges, the concept of *Edge AI* has emerged, which refers to the deployment of AI algorithms directly on edge devices, rather than relying on centralized cloud-based servers. By processing data locally on the device, Edge AI can significantly reduce latency, improve privacy, and operate independently of an internet connection. This approach is especially beneficial for real-time applications and mobile environments, where the constraints of bandwidth, energy, and computational resources are more pronounced.

1.1.2 Types of Problems Addressed by Machine Learning

	Supervised	*Unsupervised*
Discrete	Classification	Clustering
Continuous	Regression	Dimensionality Reduction

Figure 1.1: Four categories of problems addressed by ML

The problems that Machine Learning tackle falls in one of four categories, as depicted in figure **1.1**, depending on whether the algorithm training is *supervised* or not and whether the data space is *continuous* or *discrete.*

Regression

In a nutshell, regression focuses on predicting continuous numerical values based on input features by minimizing a cost function.

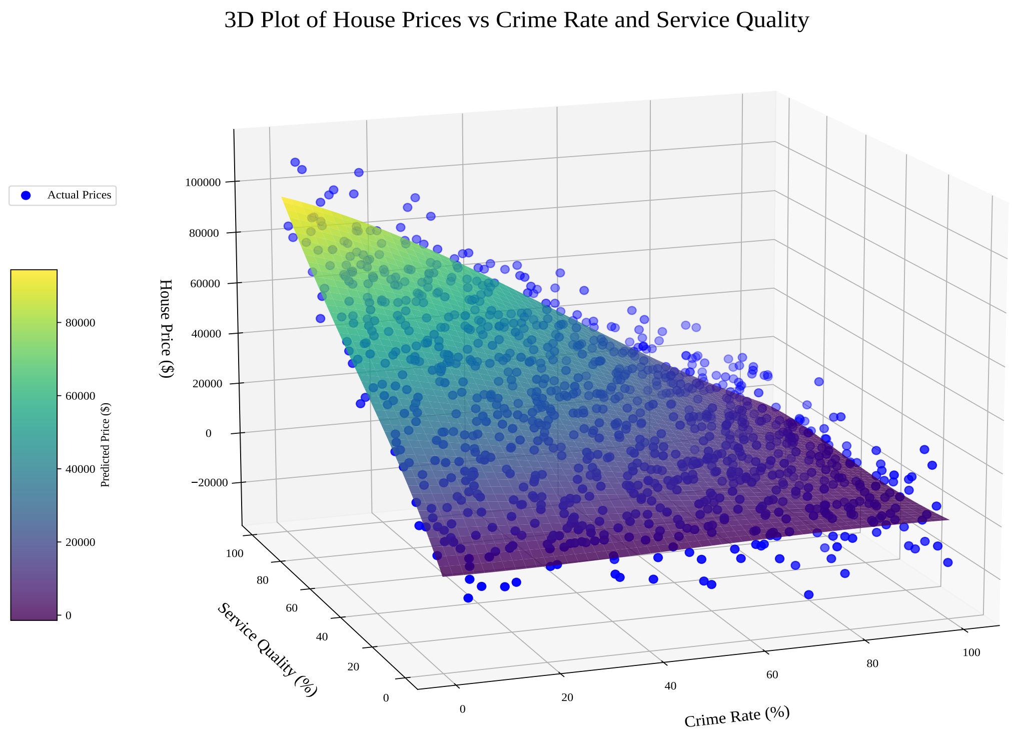


Figure 1.2: An example of a fourth-order linear regression model used to predict house prices based on service quality and crime rates in the area

Unlike classification, where the output is a discrete label, regression models aim to understand the relationship between independent variables (*features*) and a dependent variable (the *target* or output).

The general approach to regression is composed of a progression of steps:

1. Choose a *model* describing the relationships between variables of interest.

   e.g. Linear regression:

   $$\mathbf{y} = \mathbf{w} \cdot \mathbf{x} + b$$

   Weights (w) and *biases* (b) are the parameters of the algorithm that are learned during the training phase.

2. Define a *loss function* that quantifies the degree to which the model's predictions deviate from the actual data. The most popular choice is the L_2 norm.

$$L(x) = (t(x) - y(x, w, b))^2 \tag{1.1}$$

In this regard, it's crucial to select a *regularizer* that reflects the preference for simpler candidate explanations among those with similar loss function values, in accordance with Occam's principle [3].

3. Fit the model by minimizing the loss function using an optimization algorithm, effectively training the model.

In real-world scenarios, problems are often too complex to be analysed accurately with a simple linear regression.

Generalized regression **(1.2)** is a powerful expansion of this concept, as it's capable of capturing more complex, non-linear relationships within the data source by introducing higher-order terms.

$$y = \sum_{i=1}^{n} w_i \cdot x_i^{(i-1)} \tag{1.2}$$

However, choosing the order of the regression is a delicate matter, as it can have destructive consequences on the effectiveness of the model.

To address this challenging decision, the *generalization error* can be referred to, which represents the error that occurs when the model is applied to new, unseen data. It is influenced by two main components:

- **Bias error**: It arises from approximating a complex real-world problem with an overly simplified model. Bias represents the difference between the predictions of the *average model* and the *actual outcomes*, caused by incorrect assumptions and simplifications. Over-simplified models, such as those with a low order of regression, tend to produce high bias.
- **Variance error**: It reflects the model's sensitivity to *small fluctuations* in the training data, often resulting from a model with too high an order of regression. Variance measures how much the model's predictions would vary if it were trained on different datasets drawn from the same distribution.

To estimate the generalization error, the ground truth data is divided into two subsets:

- **Training set**: This subset is used to perform the regression and train the model.

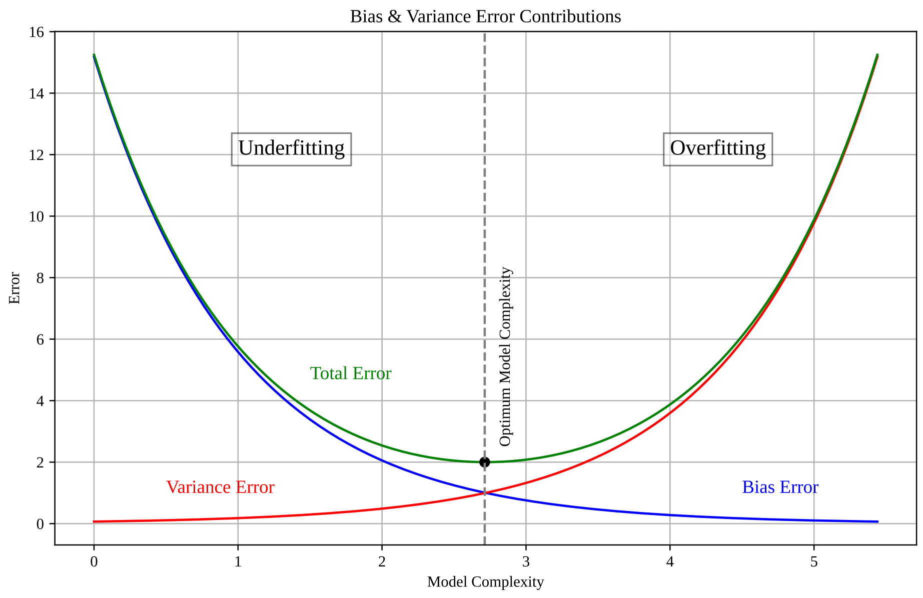


Figure 1.3: Bias and variance contributions to the generalization error

- **Validation set**: This subset simulates the data that will be encountered during inference and it is used to evaluate the model's fit and performance.

The contribution of variance to the generalization error typically follows an exponential pattern relative to model complexity, while the contribution of bias follows a decaying exponential curve. Figure **1.3** illustrates this relationship and highlights the combined effect of these contributions, i.e. the overall generalization error. Furthermore, it's possible to identify two distinct situations:

- **Underfitting**: This occurs when the model is *too simple* to capture all the relevant characteristics of the data. It is identified by both a *high training error* and a *high validation and test error*.
- **Overfitting**: Conversely, overfitting happens when the model is too complex and captures irrelevant details or noise in the data. This is characterized by a *low training error*, due to the model's high complexity, but a *high validation error*.

Overfitting is generally preferred over underfitting as a starting point, since it provides a better foundation for refinement. Various techniques can reduce the model's complexity to an optimal level, whereas increasing the complexity of an underfitting model is more challenging.

A simple yet effective approach to address overfitting is to *increase the dataset size.* Although this may slightly raise the training error, it helps reduce the validation error, thereby mitigating overfitting. In this context, the importance of data collection becomes even more crucial.

Another highly effective approach is to progressively reduce the number of parameters while maintaining a low training error. This powerful technique, known as *regularization*, allows for tuning model complexity to reduce both bias and variance.

The core principle of regularization involves incorporating *model complexity* directly into the *cost function.*

Two popular types of regularization applied to generalized regression are:

- **Ridge**: This method adds $\|w^2\|$, the squared sum of the model parameters, to the cost function, multiplied by a regularization factor λ.
- **Lasso**: This method adds $\|w\|$, the sum of the model parameters, to the cost function, also multiplied by a regularization factor λ.

The regularization factor λ itself is a parameter that must be carefully selected. Increasing λ too much can lead to underfitting by *overly simplifying* the model, while decreasing it too severly can result in overfitting by allowing the model to become *too complex.*

The final step in building a regression model is selecting a suitable optimization algorithm. One of the most widely used methods is *gradient descent*, an iterative technique that is often more effective than direct solutions in explicit form. In essence, gradient descent seeks to find the minimum of the loss function by following the steepest descent direction. At each iteration, the algorithm computes the *partial derivative* of the loss function with respect to each parameter w_j (denoted as $\frac{\partial L}{\partial w_j}$) and adjusts the parameters based on the sign of this derivative:

- If $\frac{\partial L}{\partial w_j} > 0$, increasing w_j would increase the loss L, so w_j should be decreased.
- If $\frac{\partial L}{\partial w_j} < 0$, increasing w_j would decrease the loss L, so w_j should be increased.

The parameters are updated using the rule in equation (**1.3**), (**1.4**) in vector form, where α is the learning rate.

$$w_j \leftarrow w_j - \alpha \frac{\partial L}{\partial w_j} \tag{1.3}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla L(\mathbf{w}) \tag{1.4}$$

The learning rate α is an example of a *hyperparameter*, which influences both the *speed* and the likelihood of *convergence* to a minimum. If α is too small, gradient descent progresses slowly. Conversely, if α is too large, the algorithm may *overshoot* the optimal point and potentially diverge.

Figure **1.4** illustrates the *Mean Squared Error* of a model trained with three different learning rates, highlighting the impact of suboptimal parameter choices.

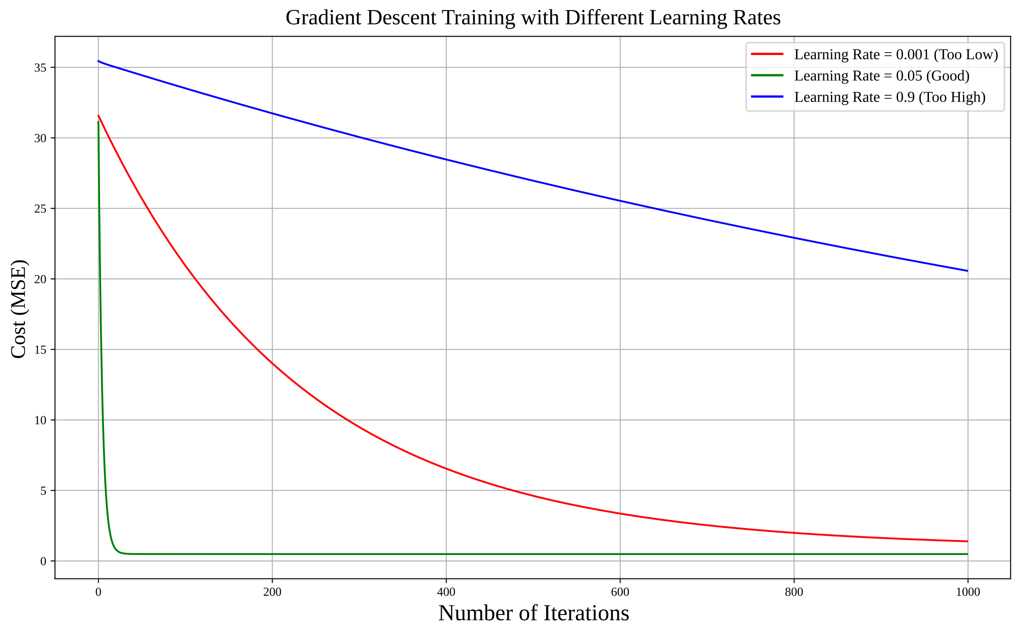


Figure 1.4: Behaviour of the gradient descent optimization algorithm with three different learning rates

When features have widely varying value ranges, or are skewed, convergence during optimization can become significantly slower. To mitigate this effect, the dataset can be *scaled* and *normalized* using the transformation in equation (**1.5**).

$$x_i' = \frac{x_i - \mathrm{mean}(x_i)}{\max(x_i) - \min(x_i)} \tag{1.5}$$

This process helps to standardize the feature values, ensuring that they are on a similar scale, which in turn can *accelerate convergence* and improve the efficiency of the optimization algorithm.

Gradient descent is a powerful method but it might be inefficient when the dataset is too big, which as seen previously is a necessary measure that contributes to the reduction of overfitting.

A potential solution is *Mini-Batch Stochastic Gradient Descent*, a technique that updates the weights based on the loss function computed from a small subset of the training data, known as a *"mini-batch"* or simply *"batch"*. The update rule is given by equation (**1.6**).

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot (y - t) \cdot \mathbf{X}_B \tag{1.6}$$

where $\mathbf{X}_B$ represents the mini-batch.

This approach balances the efficiency of *Stochastic Gradient Descent*, which uses a single data point per update, and the stability of traditional Gradient Descent, which instead uses the entire dataset.

Classification

This class of algorithms tries to apply a prediction function to a representation of the input data to get the desired output as a *discrete label*, as opposed to a continuous value. A wide range of solutions can be found, ranging from the simplest and most efficient solutions up to large networks with billions of parameters. Let's proceed in complexity order.

Decision boundary

This problem involves classifying data based on whether they fall inside or outside a certain boundary, which must be learned during training. The objective is to identify an appropriate *boundary*, or model, that can effectively predict the class of data based on its position relative to this boundary. By leveraging high-order features, it is possible to learn n-dimensional separation boundaries.

To address this classification problem, *logistic regression* is employed, which applies the principles of linear regression to a discrete case, as described with equation (**1.7**), where σ is the *sigmoid* activation function that enables classification.

$$y = \sigma(\mathbf{w} \cdot \mathbf{X} + b) \tag{1.7}$$

However, the typical loss function used in linear regression, the L_2 norm, is unsuitable here. When combined with the sigmoid function, it leads to a critical issue known as the *vanishing gradient* problem, which prevents the weights from being properly updated and halts learning.

This phenomenon occurs especially when the weights are very distant from 0, as they are randomly initialized at the beginning of each epoch. Instead, the *Cross-Entropy* loss function can be used, as it avoids the vanishing gradient issue, allowing effective training of the model. Figure **1.5** clearly demonstrates the impact of this situation on the training process.

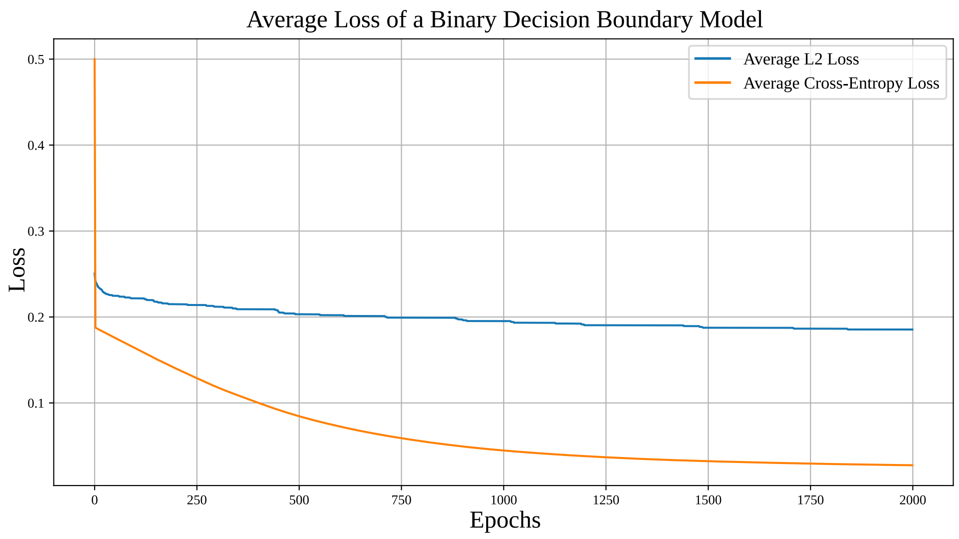


Figure 1.5: Average loss for a binary classification model trained once with *L2* and once with *Cross-Entropy* as loss functions, averaged over 500 runs

Support Vector Machines

Support Vector Machines (SVMs) are a robust classification technique that, like decision boundary methods, seek to separate data into distinct classes.

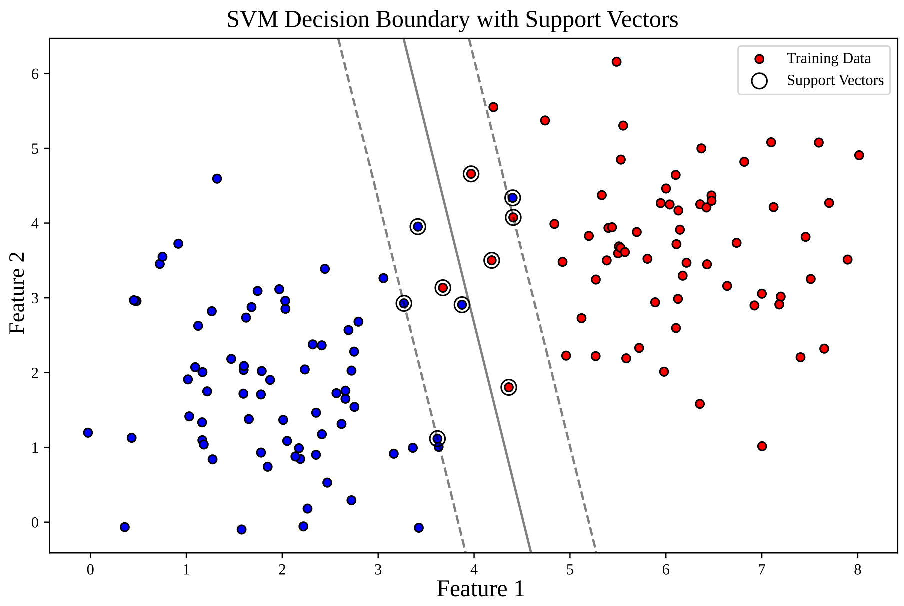


Figure 1.6: Decision Boundary model obtained using SVMs, with data randomly generated in two clusters

SVMs focuses on maximizing the margin between the closest data points from each class, known as *support vectors*. The *hyperplane* that SVMs identify as the decision boundary is the one that maximizes this margin.

The closest elements to the boundary are called *support vectors*, as shown in figure **1.6**. They influence the position and orientation of the hyperplane, defining the margin of separation between the two classes.

Decision Trees

Decision Trees are used to model decisions in a hierarchical structure, where each node represents a feature test, and each branch represents an outcome of that test, leading to a final decision at the leaf nodes.

This makes decision trees *highly interpretable*, as the decision-making process can be visualized and understood easily. They can handle both categorical and numerical data, and they naturally perform feature selection, making them versatile for various types of datasets.

Furthermore, decision trees are highly sequential and computationally cheap, making them perfect for CPUs and mobile devices.

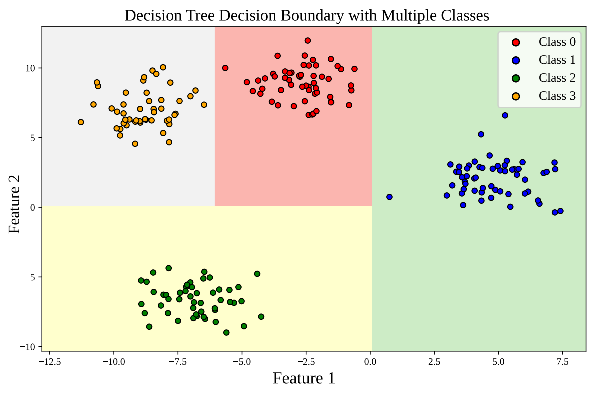


Figure 1.7: Decision Trees model with four classes

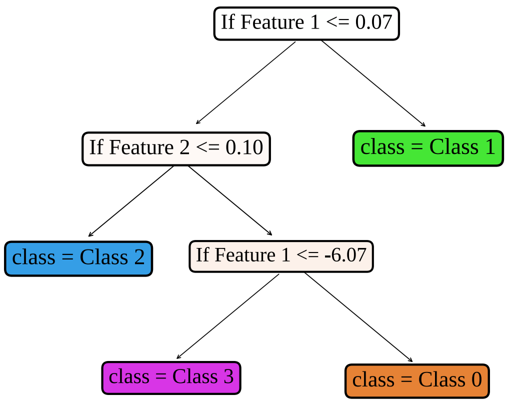


Figure 1.8: Structure of the Decision Trees model after training

However, decision trees have certain limitations. *Small trees* are inexpensive to train but tend to have high bias, while *large trees* are computationally expensive and often exhibit high variance. Additionally, they are generally prone to overfitting.

To achieve a balance between these extremes, small trees can be combined into ensembles known as *forests*. In this approach, *randomness* is crucial, as each tree is trained on a random subset of the data, which plays a more significant role than in typical mini-batch training.

The decisions of these trees are then combined through a voting process known as "*Bagging*" (Bootstrap Aggregating).

While decision trees are not well-suited for models with a large number of features, such as image recognition tasks, they offer a good compromise for applications on platforms with limited computational power and resources.

Neural Networks

Neural Networks are a class of machine learning models loosely inspired by early knowledge on the structure and function of the human brain. In classical neural networks, each *"neuron"* computes a linear combination of input features followed by a non-linear activation function, which is essential for enabling the network to generalize beyond the training data.

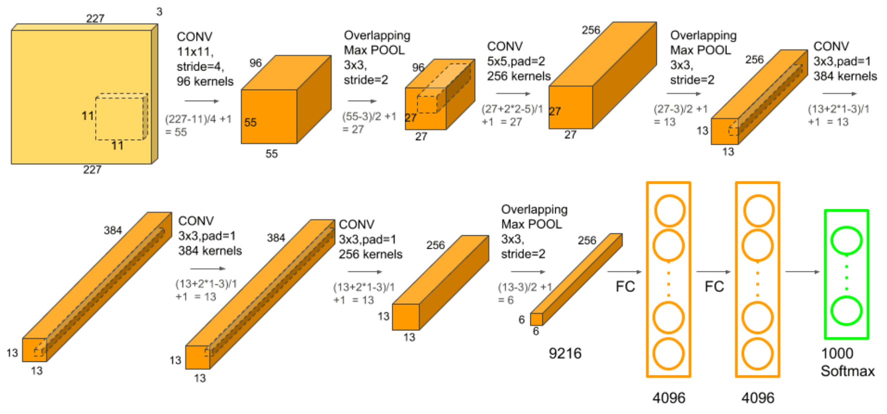

227
3
11
11
227
CONV 11x11, stride=4, 96 kernels
(227-11)/4 +1 = 55
96
55
55
Overlapping Max POOL 3x3, stride=2
(55-3)/2 +1 = 27
96
27
27
CONV 5x5,pad=2 256 kernels
(27+2*2-5)/1 +1 = 27
256
27
27
Overlapping Max POOL 3x3, stride=2
(27-3)/2 +1 = 13
256
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 256 kernels
(13+2*1-3)/1 +1 = 13
256
13
13
Overlapping Max POOL 3x3, stride=2
(13-3)/2 +1 = 6
256
6
6
9216
FC
FC
4096
4096
1000 Softmax

Figure 1.9: AlexNet is a popular deep convolutional neural network architecture with 8 layers, designed for image classification, featuring stacked convolutional layers, ReLU activations, max-pooling, and dropout to improve performance and prevent overfitting [4]

The linear combination, while computationally straightforward, involves numerous coefficients that require efficient handling of multiplications and additions.

Among the most commonly used activation functions are:

- **Sigmoid function**

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{1.8}$$

The sigmoid function was historically favored for its smooth gradient, as can be seen in figure **1.10**, which facilitates learning through gradient descent methods, and its bounded output, making it suitable for binary classification tasks.

However, it also presents significant drawbacks, including the *vanishing gradient problem*, as explained in depth in the decision boundary paragraph **1.1.2**, and the issue of a non-zero-centered output, which can cause inefficiencies during optimization.

- **Hyperbolic tangent (tanh)**

$$\tanh(z) = \frac{2}{1 + e^{-2z}} - 1 \tag{1.9}$$

This function is similar in form to the sigmoid but produces outputs between -1 and 1, as shown in figure **1.11**, providing a zero-centered output that helps mitigate some of the optimization issues associated with the sigmoid function.

Despite this advantage, the tanh function also suffers from the *vanishing gradient problem* for extreme input values.

- **Rectified Linear Unit (ReLU)**

$$\text{ReLU}(z) = \max(0, z) \tag{1.10}$$

This function and its variants have gained significant popularity in recent years and constitute foundational blocks in deep learning applications.

It offers a simple, non-linear transformation that retains only positive values, as depicted in figure **1.12**. ReLU is computationally efficient and, most importantly, it mitigates the vanishing gradient problem by providing a constant gradient for positive inputs.

However, it is not without issues, such as the potential for *"dying ReLUs"*, where neurons can become inactive during training. Variants like **Leaky ReLU** introduce small gradients for negative inputs to prevent neuron inactivity.

- **Softmax**, commonly used in the output layer of classification networks. It converts logits into probabilities that sum to one, making it suitable for multi-class classification tasks. By comparing different activation functions, it becomes clear that while each has specific strengths and weaknesses, the choice of activation function significantly impacts the performance and training efficiency of neural networks.

Neural networks are organized in a layered architecture, where each layer consists of a set of neurons that process input data and pass the results to the next layer.

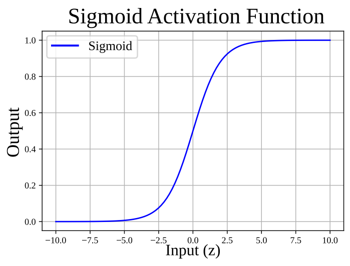


Figure 1.10: Sigmoid function

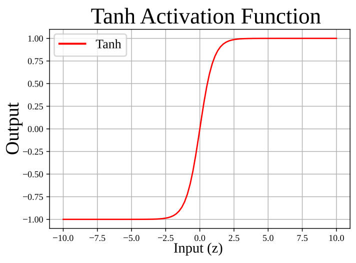


Figure 1.11: Hyperbolic tangent function

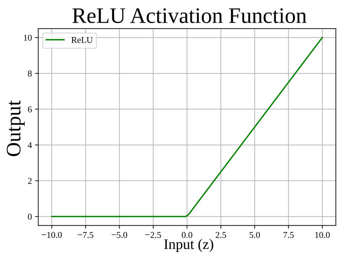


Figure 1.12: ReLU function

This structure, where neurons in one layer are connected to those in the subsequent layer, forms a *Directed Acyclic Graph* (DAG).

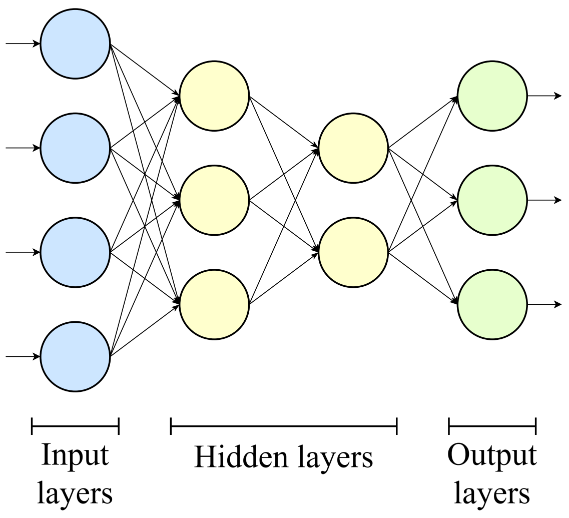


Figure 1.13: Example of a Neural Network with two hidden layers

The layers are typically divided into three main types: the *input* layer, *hidden* layers, and the *output* layer, as shown in figure **1.13**. The input layer receives the raw data, which is then transformed by one or more hidden layers through weighted connections.

Finally, the output layer produces the model's predictions. The connections between layers are *unidirectional*, meaning information flows in a single direction from the input to the output, ensuring the graph remains acyclic.

The process of training a neural network involves the *backpropagation* algorithm, which is an iterative method for optimizing the weights of the connections between neurons by minimizing the error in the network's predictions. After each input passes through the network, the prediction is compared to the actual target value, and an error is calculated.

This error is then propagated backward through the network, layer by layer, to update the weights. The weight updates are guided by the gradient of the error with respect to each weight, calculated using the *chain rule* of DAGs. This allows the network to iteratively adjust its parameters in a direction that reduces the error.

It is in this context that the *vanishing gradient* could effectively disrupt the training process. The repeated multiplication by small values can lead to the gradients becoming so small that they effectively *"vanish"*, making it nearly impossible for the weights in the earlier layers to be updated effectively. As a result, the network fails to learn important features in the early layers, which severely limits the overall learning capacity of the model.

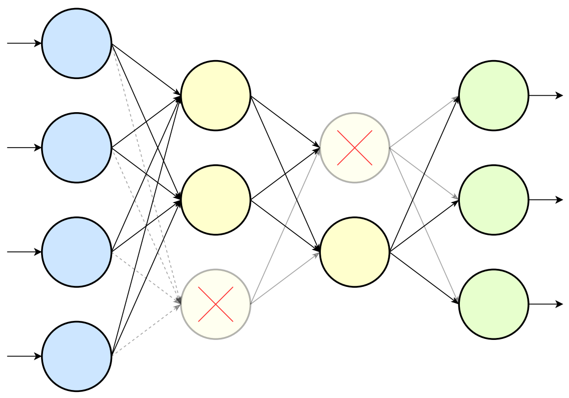


Figure 1.14: Example of an intermediate step of the training of a Neural Network with a Drop Out regularizer applied

Similarly to regression models, *regularization techniques* are used to reduce overfitting.

A very popular approach is the *Drop Out* method, represented in figure **1.14**. It operates by randomly *"dropping out"* or deactivating a subset of neurons during each training iteration. The remaining neurons must compensate for the missing connections, which encourages the network to develop redundant, distributed representations of the data.

Furthermore, Drop Out can reduce the memory required to store the model's weights when they are represented using a *sparse matrix representation.* By deactivating a significant number of neurons, the network effectively operates with fewer active parameters at any given time, which can be beneficial in memory-constrained environments.

Convolutional Neural Networks (CNNs) are a specialized class of deep neural networks designed specifically for tasks like image recognition and computer vision.

Traditional deep neural networks, while powerful, have limitations when applied to vision tasks due to their large number of parameters and computational load.

CNNs address these challenges by replacing full neuron-to-neuron connections with *convolutions*, where each neuron is connected only to a local region of the input, known as the *receptive field*. This localized connectivity allows the network to focus on small, meaningful patterns in the input data, such as edges or textures, which are then combined in deeper layers to recognize more complex structures.

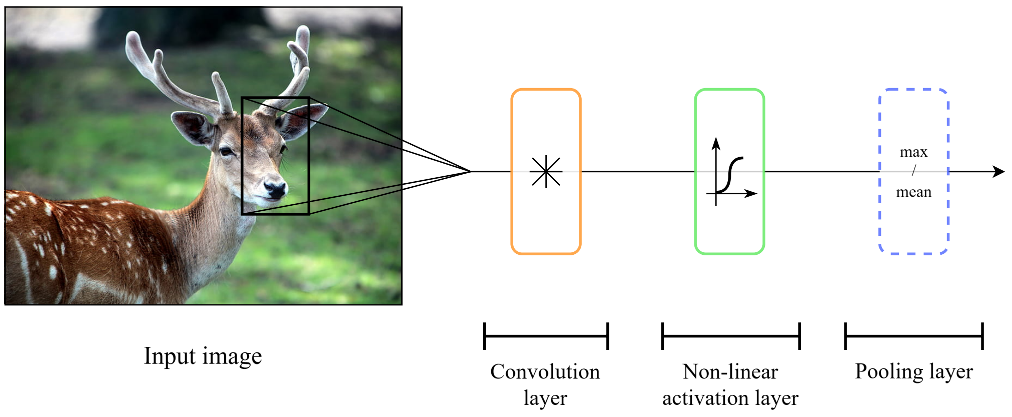


Figure 1.15: Example of a single Convolutional Neural Network layer

A key feature of CNNs is the use of shared weights across all neurons in a layer, significantly reducing the number of parameters and the need for large amounts of training data.

Figure **1.15** represents the structure of a generic CNN layer, composed of:

- **Convolutional layer**: This layer performs the core convolution operation.
- **Activation function**: As previously discussed, the activation function is crucial for enabling generalization.
- **Pooling layer**: This layer further limits connectivity by summarizing information in local regions of the feature map. The most common pooling techniques are *max pooling* and *mean pooling*, as shown in figure **1.16**.

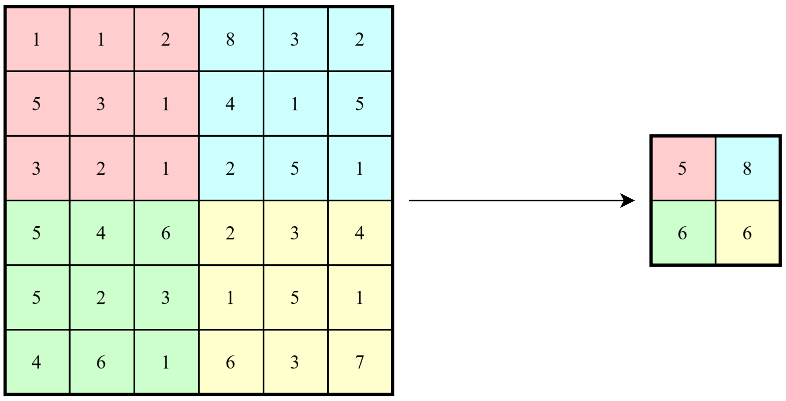


Figure 1.16: Example of a Max Pooling with 3x3 filters and a 3 element stride

However, convolutional layers alone do not comprise a CNN; they are typically followed by a limited number of *fully connected* layers and a *softmax* layer to produce a statistically sound output.

Despite the use of activation functions like *ReLU*, the vanishing gradient problem remains a significant challenge in the deepest convolutional neural networks. One potential solution is the introduction of modules that can partially or completely *bypass* groups of layers, ensuring effective backpropagation.

This approach was explored by GoogLeNet [5] with its *inception* modules, depicted in figure **1.17**.

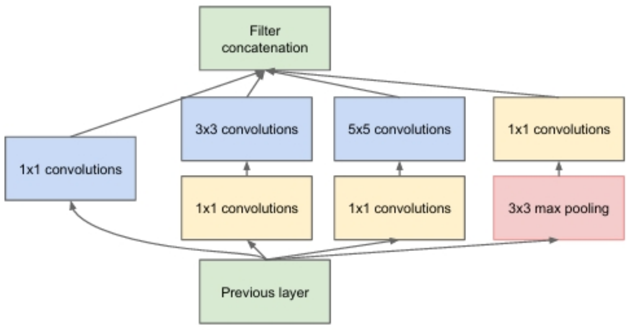


Figure 1.17: The inception module in the GoogLeNet architecture is placed on the left side of the network, where it bypasses the main blocks

Recurrent Neural Networks (RNNs) are a type of neural network designed to handle sequential data, making them particularly well-suited for tasks like language modeling, speech recognition, and time series prediction. Unlike traditional feedforward networks, RNNs have connections that form *cycles*, allowing information to persist across time steps. This enables RNNs to maintain a memory of previous inputs, which is crucial for understanding context in sequences.

However, RNNs are prone to challenges such as *vanishing* and *exploding gradients*, which can make training difficult, especially over long sequences. To address these issues, advanced variants like *Long Short-Term Memory* (LSTM) and *Transformers* have been developed, offering more robust performance by better managing the flow of information through the network.

Unsupervised Learning

Unsupervised learning refers to a type of machine learning where the model is trained on data without explicit labels or predefined outcomes. Instead of predicting specific outputs, the model seeks to identify patterns, structures, or relationships within the data itself.

This type of learning is particularly useful in situations where labeled data is scarce or where the goal is to explore the underlying structure of the data.

Two of the most important approaches to unsupervised learning are *clustering* and *dimensionality reduction*, which target *discrete* and *continuous* data spaces, respectively.

Clustering

This technique is most effective when the source data naturally forms clusters and is frequently used to uncover patterns within the data.

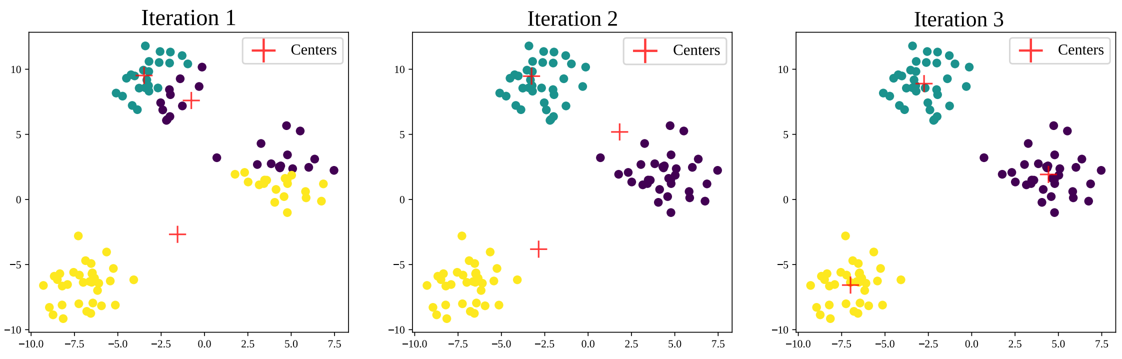


Figure 1.18: Progression of K-Means clustering across three iterations, showing the iterative refinement of cluster centers as they converge towards their final positions

A popular solution to this problem is *K-means clustering*, a *NP-hard* algorithm which is based on the assumption that each point in a cluster is close to its cluster center. The algorithm begins by randomly initializing *k centroids*, which serve as the initial centers of the clusters. It then iteratively assigns each data point to the nearest centroid, forming clusters, and updates the centroids to be the mean of the points in each cluster. This process repeats until the centroids stabilize, indicating that the clusters are well-formed.

While K-means is simple and efficient, its performance depends on the initial placement of centroids and the choice of k, which must be predetermined. Additionally, K-means assumes that clusters are *spherical* and *equally sized*, which may not suit all datasets, and it can be sensitive to outliers.

Dimensionality Reduction

This technique is particularly useful when the source data presents a *large number of features* without any indication of their relevance. It can improve both *training speed* and *prediction quality* by contributing to the reduction in overfitting.
A popular tool to perform dimensionality reduction is the *principal component analysis* (PCA). The algorithm operates by projecting data onto a subspace spanned by a basis of eigenvectors derived from the data space.
For each point x_i in the original space, the algorithm identifies the closest point x'_i within this subspace, effectively reducing the dimensionality while preserving as much variance as possible.

There are two possible ways to describe the PCA progression to find the best subspace in a 2D space. Let's take two subspaces, S1 and S2, represented in figure **1.19**. On one hand, S1 is a better subspace beacause has a lower average distance from the elements than S2. On the other hand, the projections of the elements on S1 are more *"spread out"* than S2's projections. Therefore, two criteria can be used to obtain the same result:

- *Minimize the projection error*
- *Maximize the variance of the projected errors*

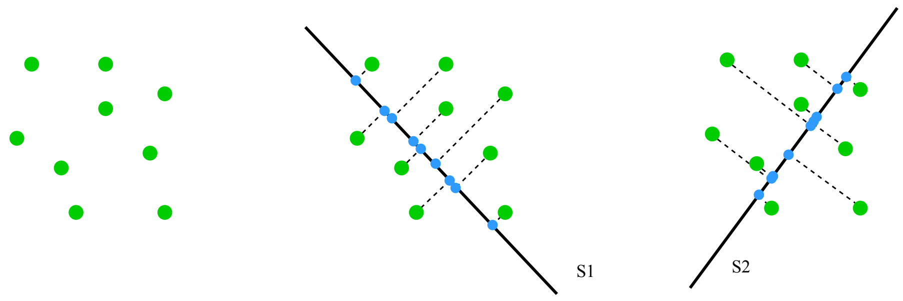


Figure 1.19: Simple two-dimensional example of Principal Component Analysis.

Generalizing the previous example, the algorithm for PCA relies on the spectral decomposition of matrices. A symmetric matrix A has a full set of eigenvectors, which can be chosen to be orthogonal, resulting in the decomposition $A = Q\Lambda Q^T$, where the columns of Q are the eigenvectors of A, and the elements of Λ are its eigenvalues.

PCA is typically performed on the empirical covariance matrix Σ, defined as $\Sigma = \sum(x_i - \mu)(x_i - \mu)^T/N$, where μ is the mean of the data points. It can be proven that Σ is positive semidefinite, meaning all its eigenvalues are non-negative.

PCA reduces the dimensionality of the data by selecting the eigenvectors corresponding to the largest K eigenvalues, which are known as the principal components of Σ. The orthogonality of these eigenvectors ensures that the new features derived from them are *uncorrelated*.

Principal Components can be learned using an *Autoencoder*, which is a feed-forward neural network designed to predict x from x, essentially approximating the identity function.

In the case of a simple 2-layer Autoencoder, the network learns the Principal Components, with the weights W_1 and W_2 corresponding to the matrices Q^T and Q, respectively.

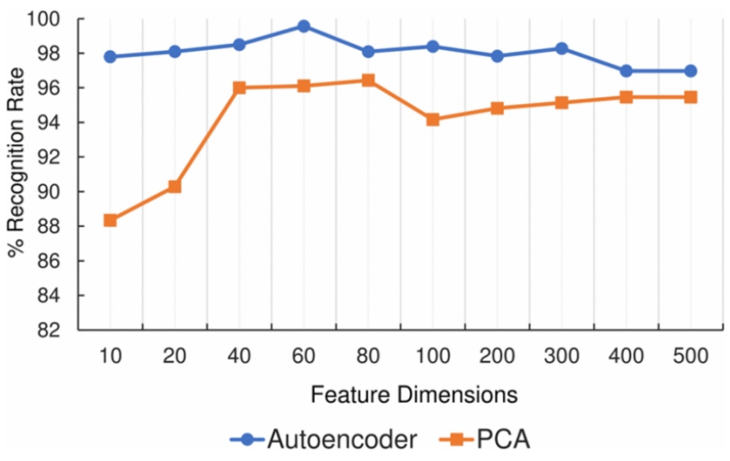


Figure 1.20: Accuracy obtained using different feature dimensions, as extracted from the 2017 paper *'Using Deep Autoencoders for Facial Expression Recognition'* [6]. This paper investigates the effectiveness of deep autoencoders for feature selection and dimension reduction in facial expression recognition, demonstrating that the features extracted from the stacked autoencoder outperform other state-of-the-art techniques, including the standard PCA, as shown in this graph.

While this setup allows the network to learn linear Principal Components, deeper networks can go beyond and learn *non-linear Principal Components*, providing a more powerful alternative to standard PCA. Deep Autoencoders, in fact, are more powerful than PCA when using the same number of parameters, similar to how deep neural networks generally outperform their shallow counterparts, as shown in figure **1.20**.

Reinforcement Learning

Reinforcement Learning (RL) is a distinct field of Machine Learning that emerged in the 1950s, building on the foundational work of Andrey Markov, who introduced the *"Markov Decision Process"* (MDP) as a framework for decision-making in uncertain and unsupervised environments.

At the core of MDPs is the concept of *Markov Chains*, which describe a sequence of states where transitions to the next state depend solely on the current state, without regard to prior states. MDPs extend this by incorporating decision-making, allowing an agent to select actions that influence these state transitions.

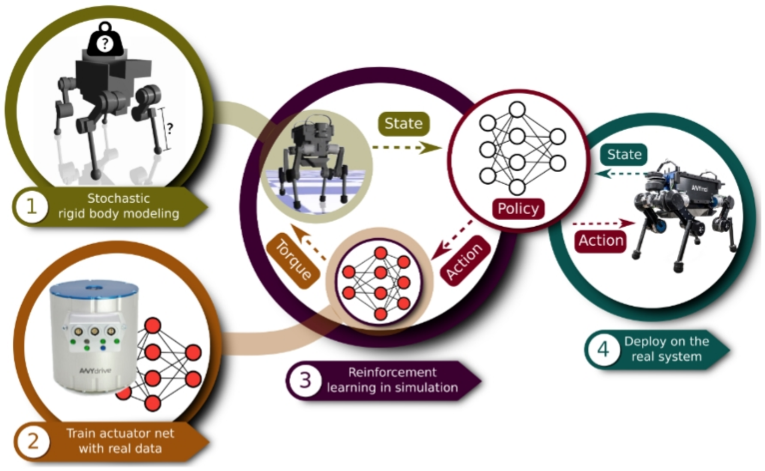

1 Stochastic rigid body modeling
2 Train actuator net with real data
3 Reinforcement learning in simulation
4 Deploy on the real system
State
Policy
Action
Torque
State
Action

Figure 1.21: Example of a Reinforcement Learning model training for ETH's quadrupedal system *ANYmal*, by Hwangbo et al [7].

In RL, the problem is framed as an agent exploring an environment to achieve a goal by maximizing expected cumulative rewards. The agent interacts with the environment by sensing its current state and taking actions that alter this state to obtain rewards.

While the reward signal represents the immediate benefit of a specific action, the value function captures the *long-term benefit*, or cumulative reward, expected from a given state. The central objective of RL algorithms is to determine an *optimal policy*, i.e. a strategy for selecting actions, that maximizes the value function and ensures the agent consistently achieves the highest possible cumulative rewards over time.

RL algorithms can be broadly categorized into model-free and model-based approaches. *Model-free* algorithms, which do not build an explicit model of the environment or the underlying Markov Decision Process (MDP), rely on trial-and-error interactions with the environment to learn optimal policies.

These algorithms are further divided into value-based and policy-based methods.

- **Value-based algorithms** focus on accurately estimating the *value function* of each state using the recursive Bellman equation. By sampling trajectories of states and rewards, the agent can estimate the value function and subsequently determine the optimal policy by acting greedily with respect to the learned value function. Well-known value-based algorithms include *SARSA* and *Q-learning*.

- **Policy-based algorithms** directly parametrize the policy, turning the learning process into an explicit optimization problem. The agent samples trajectories to adjust the policy in a way that maximizes the expected cumulative reward, as seen in methods like REINFORCE and deterministic policy gradient (DPG).

While policy-based approaches can model continuous action spaces, they suffer from high variance and instability during training. Value-based methods are more stable but less effective in continuous action domains.

Actor-critic algorithms combine both approaches by parametrizing both the policy (actor) and the value function (critic), leveraging the strengths of each to stabilize training and enhance performance.

1.1.3 Core Computational Mechanisms in ML

In the previous section, the primary types and structures of Machine Learning algorithms were discussed. This section now delves deep into the core computational mechanisms essential for effective training and inference of these models.

Vector and matrix products: the workhorse of ML

Vectors and matrix products are fundamental to a wide range of computations essential to most ML algorithms. Data points are typically represented as vectors, while datasets are organized as matrices, with each row corresponding to a data point and each column to a feature. These vectors and matrices are crucial for applying linear transformations, such as matrix-vector multiplication, which facilitates data manipulation and transformation within models.

A general NN can be represented by a single equation that relates input activations, weights, biases, and the activation function, as shown in equation (**1.11**).

$$\mathbf{y} = g(\mathbf{x}) \cdot (\mathbf{W} \cdot \mathbf{x} + \mathbf{b}) \tag{1.11}$$

This formulation highlights that at the core of a neural network lies the matrix product of *weights* ($\mathbf{W}$) and *inputs* ($\mathbf{x}$), which is the primary computational challenge in machine learning, and optimizing its execution can significantly reduce processing time.

This neural network structure is simply an instance of the *General Matrix Multiplication* (GEMM), a fundamental operation in linear algebra. Due to extensive research in this field, GEMM operations have been highly optimized for an wide variety of targets.

Popular libraries like TensorFlow and PyTorch leverage these optimized GEMM routines to improve the performance of neural networks built with them.

Furthermore, in convolutional neural networks (CNNs), convolutions can be restructured as matrix multiplications using techniques like *im2col*, which will be explained in detail further along this section.

Thanks to this transformation it is possible to take advantage of highly optimized GEMM routines or even specialized hardware, from GPUs and TPUs to heterogeneous systems with specific accelerators.

CNNs Computational Structure

As the name states, *convolution* is at the base of CNNs. It is a fundamental mathematical operation that combines two functions to produce a third function, reflecting how the shape of one is modified by the other.

For continuous functions $f(t)$ and $g(t)$, the convolution $(f * g)(t)$ is defined as the integral in equation (**1.12**).

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) \, d\tau, \tag{1.12}$$

$f(\tau)$ is one function, and $g(t-\tau)$ is the other, reversed and shifted. In discrete settings, such as digital signal processing, the convolution of two sequences $f[n]$ and $g[n]$ is given by a discrete formulation, as shown in equation (**1.13**).

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] \cdot g[n - m]. \tag{1.13}$$

This sum captures the overlap between the sequences as one shifts across the other.

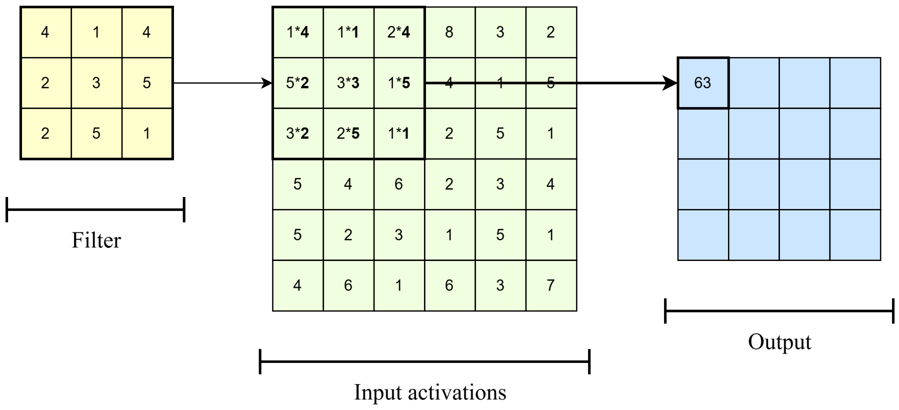


Figure 1.22: Two-dimensional example of a convolution computation, step one

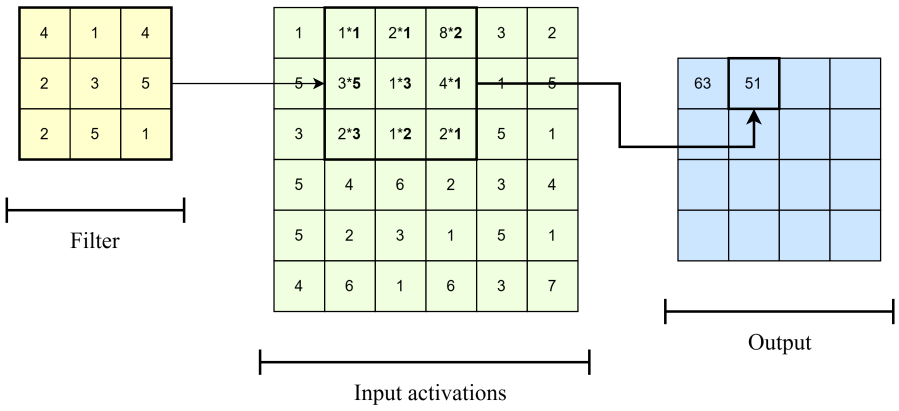


Figure 1.23: Two-dimensional example of a convolution computation, step two

Figure **1.22** and **1.23** illustrates, in two steps, part of the convolution process on a two-dimensional input, such as an image. The filter, or kernel, is *"passed"* over the image, and at each step, a pixel of the output is computed and stored.

The movement of the filter is governed by a parameter called the *stride*, which defines how many elements are skipped during the filter's traversal.

This parameter has several significant effects on the CNN:

- **Dimensionality reduction**: The stride down-samples the data, similar to pooling. Within certain limits, this can positively affect the CNN's inference performance.
- **Loss of detail**: Inevitably, some data is lost due to striding, particularly at the edges of images, where central pixels receive more attention as they are processed multiple times.

To mitigate these effects, *zero-padding* can be applied, which involves adding a border of zeros around the input image. This technique has two main consequences:

- **Increased detail**, especially at the edges, counteracting the stride effect.
- **Increased memory usage**, due to the enlargement of the input tensor.

Depending on the technique used, there could be a substantial increase in memory accesses needed to perform padding, potentially slowing down the inference process.

This mechanism reflects the mathematical definition of the convolution operation, but it clearly requires many cycles due to the movement of the filter.
Each time that the filter moves, it means that data has to be loaded from memory, RAM or external, which slows down the convolution operation. For low-latency applications, like obstacle-avoidance, the delay introduced with this implementation of the convolution can compromise the application.

Im2col transformation

The *im2col* technique, *"image to column"*, is a computational technique used to reshape an input image (or generally, any input tensor) into a format that makes convolution operations more efficient to compute.
Specifically, it converts the sliding window approach into a matrix multiplication problem, which can be efficiently handled using highly optimized General Matrix Multiply (GEMM) routines. Figure **1.24** shows some of the steps of the *im2col* process, applied to a 3x3, 3 channel tensor, using a 2x2 kernel.

The example in figure **1.24** uses one of the two most popular tensor representation formats, known as *NCHW*. N represents the batch size, C denotes the channel, H stands for height, and W for width.

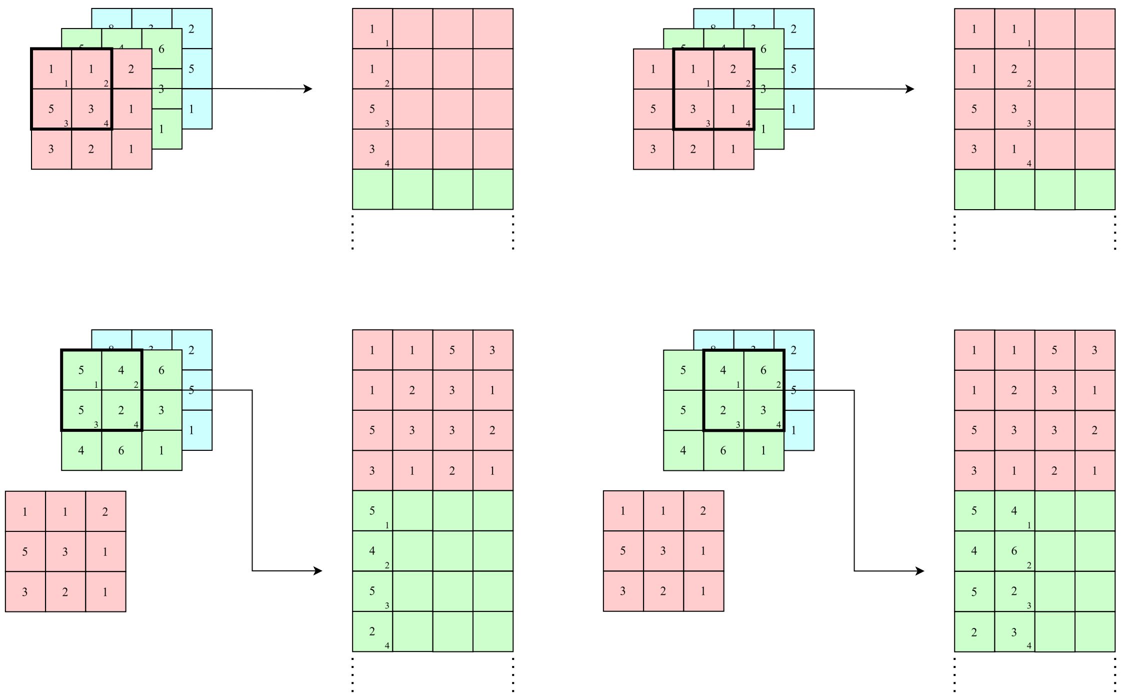


Figure 1.24: Example of *im2col* computation

However, there is another commonly used format called *NHWC*. The matrix in example **1.14** provides an example of this format, representing a 1x2x2x3 tensor, i.e. batch size 1, height and width of 2, and 3 channels.

$$\begin{bmatrix} (1,2,3) & (4,5,6) \\ (7,8,9) & (10,11,12) \end{bmatrix} \tag{1.14}$$

(1, 2, 3) are the values of the first *"pixel"* of the tensor across different channels, where "1" corresponds to CH0, "2" to CH1, and "3" to CH2. As the format suggests, height (H) comes first, followed by width (W), and finally the channel (C).

For reference, matrix **1.15** represents the same tensor but in the *NCHW* format. The submatrix (1, 4, 7, 10) is the 2x2 slice of channel 0, and similarly for the other two submatrices.

$$\begin{bmatrix} (1) & (4) \\ (7) & (10) \\ (2) & (5) \\ (8) & (11) \\ (3) & (6) \\ (9) & (12) \end{bmatrix} \tag{1.15}$$

The question of which of these two formats is the better choice naturally arises, and it is not a trivial one to answer.

Today, neural networks are developed using frameworks such as *TensorFlow* and *PyTorch*, which abstract away the low-level details of the underlying algorithms. As a result, there is limited information on tensor representation formats in the research community, and much of it is biased toward the framework's developers.
Both frameworks incorporate the *im2col* operation in their convolutional layers, with PyTorch using the NCHW format by default, while TensorFlow uses NHWC.

NVIDIA, the developer of TensorFlow, states in its documentation on Convolutional Layers [8] that "[format choice] has an effect on performance, as convolutions implemented for Tensor Cores require NHWC layout and are fastest when input tensors are laid out in NHWC." In other words, TensorFlow uses NHWC by default, so if the input data are in NCHW format, inference will require an additional conversion and, therefore, be slower.
It is worth noting that no detailed reasoning is provided for why NHWC was chosen as the default format.

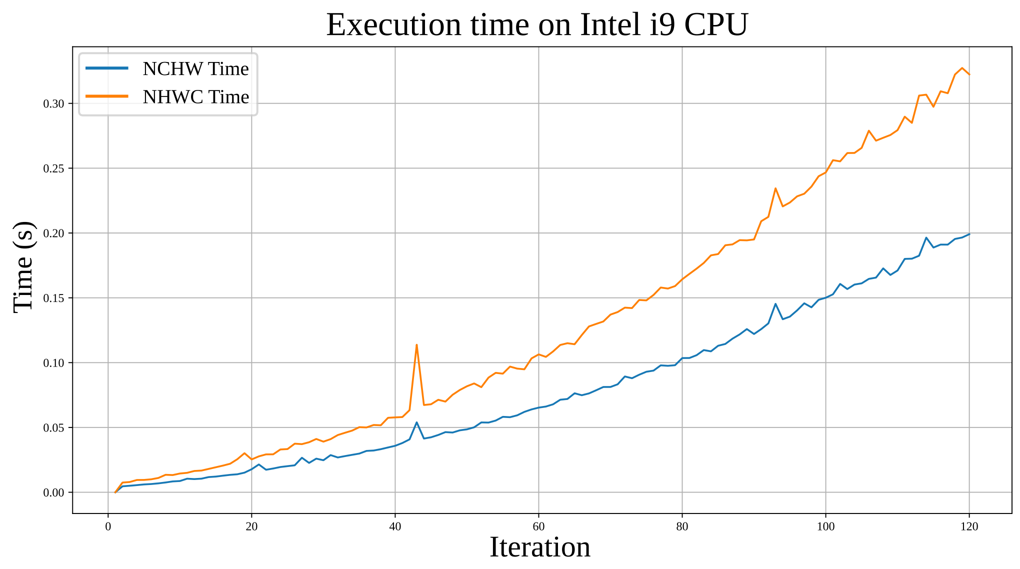


Figure 1.25: Comparison between NHWC and NCHW execution times on Intel i9 CPU

However, it is arguable that NHWC exhibits greater spatial locality than NCHW, a characteristic that can be advantageous on highly parallelized machines such as GPUs, of which NVIDIA is a leading producer. In the NHWC format, channel data for the same spatial location (height and width) are stored adjacently, allowing for more efficient parallel processing. This layout aligns well with the architecture of GPUs, where multiple channels of the same pixel are processed simultaneously, leading to more coalesced memory accesses and optimized performance on hardware like *Tensor Cores*.

On the other hand, NCHW is generally faster on CPUs than NHWC. The NCHW format benefits from better cache utilization on CPUs, where the spatial dimensions (height and width) are stored contiguously, leading to more efficient sequential memory access. This observation is confirmed by a test campaign based on the *im2col* implementation provided by XuanTie [9], a core member of the RISC-V foundation.

Figure **1.25** shows the results of the tests in terms of machine cycles. The tests were conducted on a platform equipped with an Intel Core i9 CPU, starting with a batch of 5 on a 100x100 input tensor with 3 channels, and increasing both height and width of the input at each iteration.

Moreover, in non-cached systems with low processing power, *Direct Memory Access* (DMA) controllers can still benefit from the linear access pattern of NCHW, much like caches do. In such systems, the sequential memory access provided by NCHW allows DMA controllers to efficiently transfer large blocks of data with minimal overhead.

1.2 Deploying ML Models on Edge Devices

1.2.1 Edge AI vs Cloud AI

Cloud AI refers to the use of cloud computing infrastructure to train, deploy, and execute artificial intelligence (AI) models. It is the most common method used in many consumer applications due to its ability to handle large-scale data processing and computation.

In cloud AI, data is sent from a device to remote servers, where it is processed by powerful AI models, and the results are then returned to the device. This system is ideal for applications such as virtual assistants, image recognition, and recommendation engines, which rely on massive datasets and continuous model updates.

One of the key advantages of cloud AI is its ease of *scalability*, which allows businesses to quickly increase or decrease their computational resources as demand changes. This flexibility is crucial for organizations that need to manage large volumes of data or run multiple AI applications simultaneously, without investing in additional expensive hardware infrastructure.

While cloud AI excels at processing vast amounts of data, it has limitations in situations that require real-time analysis, instant responses or operation in environments with limited or no connectivity. This is where edge AI comes into play.

Unlike cloud AI, which relies on remote servers, *Edge AI* processes data locally, on the device itself, without needing to send information back to the cloud. This makes it particularly useful in scenarios where *quick decision-making* is critical, such as in autonomous vehicles, industrial automation, or critical healthcare equipment.

By removing reliance on constant internet connectivity, Edge AI ensures the continuity of its services across diverse environments. Furthermore, edge devices enhance the *integrity* of personal data and significantly reduce the risk of *privacy* breaches.

The task of running effective *NN* models on small and mobile devices is quite a challenging one. There are several constraints that has to be taken into account when designing a model to run on edge devices:

- **Computing resources**: These devices have *limited computational power* due to their small size and energy constraints. They are rarely equipped with powerful vector processing units, which are traditionally used to accelerate inference via GEMM operations, like GPUs.
- **Power consumption**: Edge devices are often *battery-powered*, requiring them to be highly efficient, and the models must also be optimized and adapted accordingly.

1.2.2 Mitigating Critical Challenges in Edge AI

In recent years, numerous solutions have emerged to address the limitations of Edge AI, focusing on both *performance* and *efficiency*.

From a hardware perspective, new accelerators have been introduced, ranging from general-purpose to highly specialized designs. *General-purpose accelerators*, like GPUs and TPUs, offer great flexibility, enabling them to handle a wide variety of workloads.

However, this flexibility often comes with increased area and power consumption due to their need to accommodate diverse computational tasks.

In contrast, *specialized accelerators* integrated into heterogeneous platforms, such as *Systolic Arrays* and *Near-Memory Computing* (NMC) units, are highly optimized for specific tasks. These accelerators deliver greater power and area efficiency, but at the cost of reduced adaptability for tasks outside their design focus.

NMC and *In-Memory Computing* (IMC) specifically address the inefficiencies of the traditional von Neumann architecture, where memory and processing are separated, creating bottlenecks in data-intensive AI workloads. With SRAM accesses consuming up to 100× more energy than arithmetic operations, NMC and IMC minimize costly data transfers by bringing computation closer to the data.

NMC places processing units near memory to optimize bandwidth and reduce bus pressure, while IMC embeds computation directly within memory cells. Both architectures offer energy-efficient, high-performance solutions for AI workloads, significantly improving power and computational efficiency.

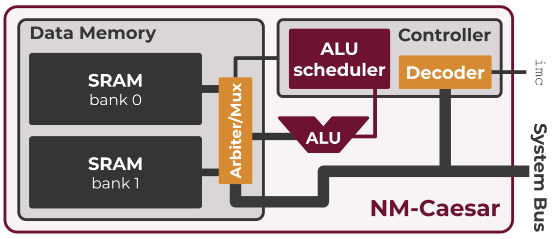


Figure 1.26: Caesar Structure

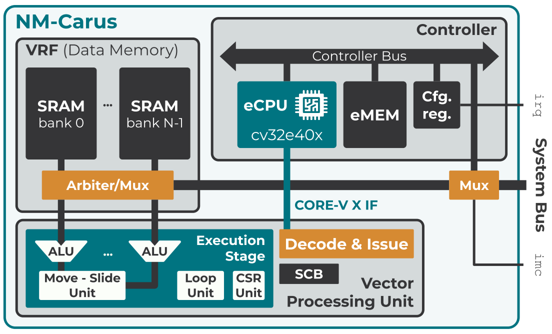


Figure 1.27: Carus Structure

NM-Caesar and NM-Carus [10] are two promising examples of such architectures. They have been developed to address the inefficiencies of traditional architectures in edge AI applications.

NM-Caesar, figure **1.26**, is optimized for area efficiency and lower-power operations, making it suitable for simpler AI tasks, such as *TinyML* benchmarks. It leverages a SIMD-based approach to reduce data movement and execute tasks like peak detection and lightweight neural networks with minimal overhead.

On the other hand, **NM-Carus**, figure **1.27**, is a fully programmable, RISC-V-based architecture designed for higher performance and more complex, data-intensive applications like deep neural networks. By integrating near-memory computing with vector processing capabilities, NM-Carus significantly improves both throughput and energy efficiency, achieving up to 50× lower execution time and 33× better energy efficiency compared to traditional RISC-V CPUs.

1.2.3 X-HEEP: eXtendable Heterogeneous Energy-Efficient Platform

Among the notable advancements in efficient edge computing, the *X-HEEP* project stands out as a versatile and successful platform.

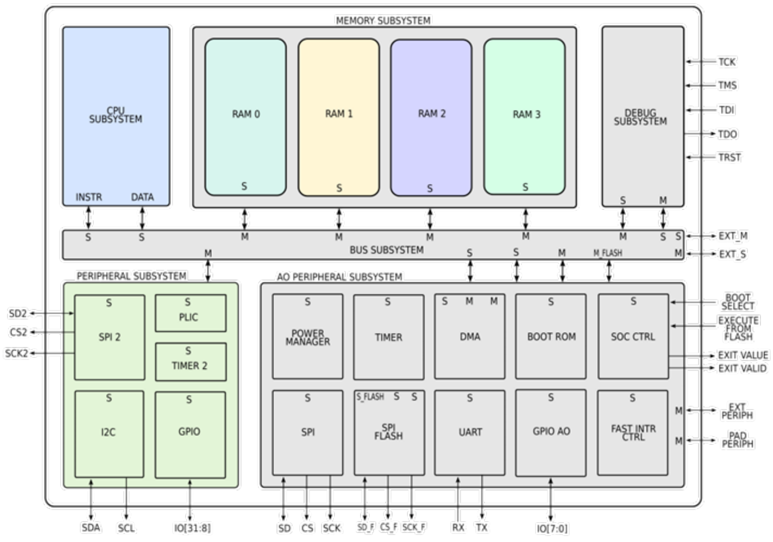


Figure 1.28: X-HEEP structure overview

X-HEEP [**11**] is an open-source, configurable, and extensible RISC-V microcontroller developed at the Embedded Systems Laboratory (ESL) of EPFL.

It is designed to serve a variety of purposes: it can be used as a standalone low-cost microcontroller, integrated into existing systems as a peripheral subsystem, or customized with external peripherals and accelerators.

One of its main strengths is the flexibility it offers developers, especially for those designing novel accelerators or peripherals, by allowing them to easily drive their Intellectual Property (IP) blocks using simple controllers.

X-HEEP is built using well-established, open-source IPs such as RISC-V CPUs and peripherals from groups like *OpenHW*, *OpenTitan* and *PULP*. The platform supports both simulation, FPGA and ASIC flows, which makes it suitable for rapid prototyping and testing.

In terms of architecture, it consists of a RISC-V CPU, a system bus, configurable SRAM memory banks, and a wide array of peripherals, including timers, UART, GPIO, DMA, and SPI, among others.

One of the key advantages of X-HEEP is its *extensibility*, allowing users to add custom peripherals and accelerators without modifying the core platform. This agility makes it ideal for domains like machine learning, where custom accelerators for tasks like matrix computations can be seamlessly integrated.

Furthermore, X-HEEP supports low-power operations through features like *clock-gating*, *power-gating*, and multiple power domains, making it highly efficient for edge computing applications.

The *MetaWearS* project and X-HEEP's role

Among the notable applications of X-HEEP, the *MetaWearS* [**12**] system stands out for its innovative use of the platform in edge AI and biomedical domains, as it levergaes X-HEEP for efficient biomedical signal processing in wearable devices.

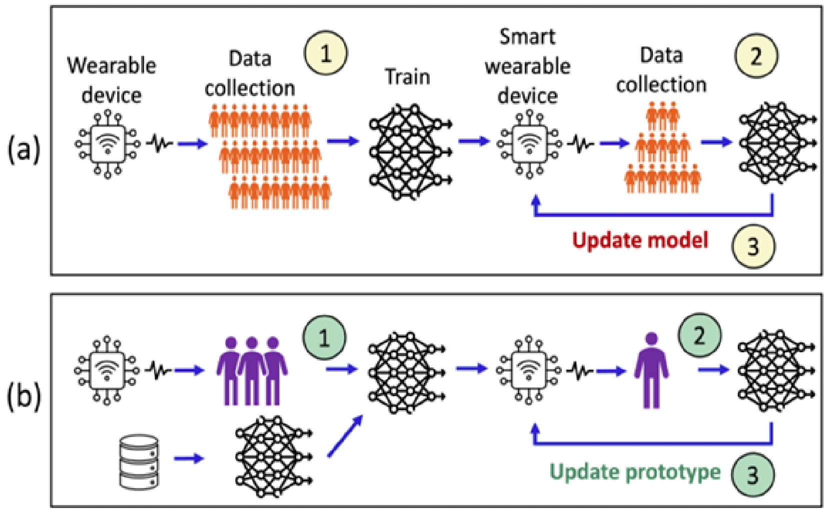


Figure 1.29: MetaWearS addresses the challenges of a wearable system lifecyle, specifically the need for large training datasets and effective model updates

MetaWearS addresses key challenges in health monitoring, such as the need for large amounts of labeled data and frequent model updates that can drain battery life. By utilizing a *meta-learning* approach, it significantly reduces the data required for both model training and updates.

X-HEEP plays a crucial role in MetaWearS by providing a low-power, flexible hardware platform that supports rapid, energy-efficient model updates. The RISC-V processing unit handles data collected from wearables while the system updates models through *Bluetooth Low Energy* (BLE) with minimal overhead.

This approach drastically improves the energy efficiency of model updates by 456x for epileptic seizure detection and 418× for atrial fibrillation detection, extending the battery life of wearable devices used for continuous health monitoring.

HEEPocrates, a 65nm implementation of X-HEEP

HEEPocrates [13] is a silicon implementation of the X-HEEP platform, specifically designed for ultra-low-power edge computing in healthcare applications.

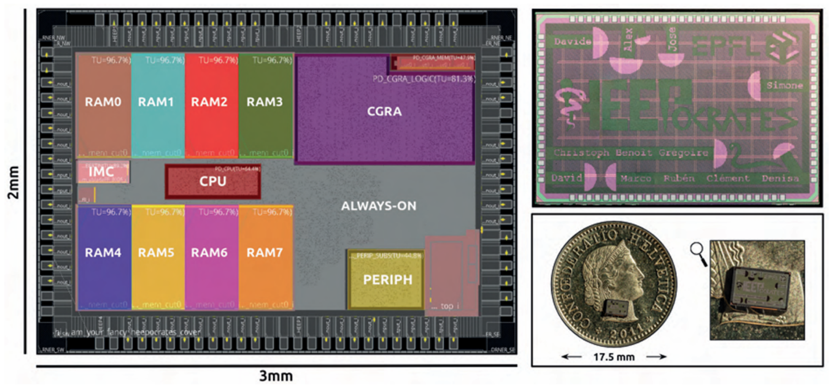


Figure 1.30: HEEPocrates layout, silicon photo, and physical realization (on a Swiss 5-cent franc coin) [13]

It leverages the core architecture of X-HEEP, enhancing it with specialized accelerators to meet the strict performance and energy efficiency demands of biomedical signal processing.

Built using a *65nm technology* node, HEEPocrates integrates the X-HEEP microcontroller with a coarse-grained reconfigurable array (*CGRA*) and *in-memory computing* accelerators. They optimize computational efficiency, particularly in tasks like seizure detection and heartbeat classification, while maintaining low power consumption.

HEEPocrates has also been integrated into *VersaSens*, a versatile wearable sensor platform designed for real-time acquisition, synchronization, and processing of bio-signals.

The VersaSens system leverages HEEPocrates in its *HEEPO* module, where it serves as a *co-processor* dedicated to accelerating machine learning and deep learning algorithms. This integration enables the platform to perform real-time inference during bio-signal acquisition, such as *electrocardiogram* (ECG), *electroencephalogram* (EEG), and *electrodermal activity* (EDA) signals.

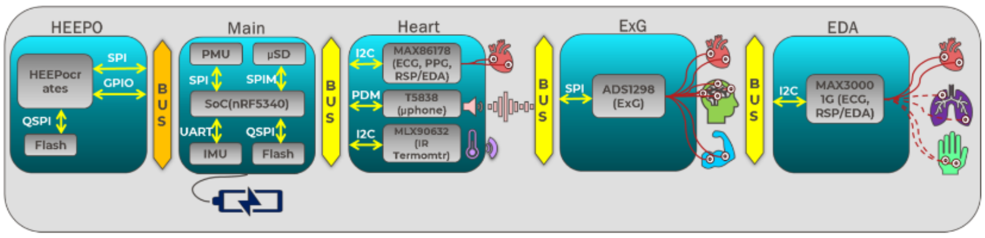


Figure 1.31: Structure of the VersaSens platform

HEEPocrates' ultra-low-power architecture makes it ideal for the wearable device, ensuring that machine learning computations are efficient without compromising battery life or performance.

The modular and customizable nature of VersaSens, combined with the HEEPocrates SoC, allows the platform to provide highly accurate, real-time data processing while maintaining wearability and user comfort. This makes it particularly useful in clinical and biomedical applications where continuous monitoring is required.

Chapter 2

Development

2.1 Data Movement in X-HEEP: the OBI Protocol

The main focus of this thesis is to reduce the limitations and obstacles in the deployment of Edge AI models on X-HEEP.

One of the most significant factors contributing to the shortcomings of edge computing is *data movement*. Therefore, it is essential to analyze how data is transferred within the X-HEEP microcontroller.

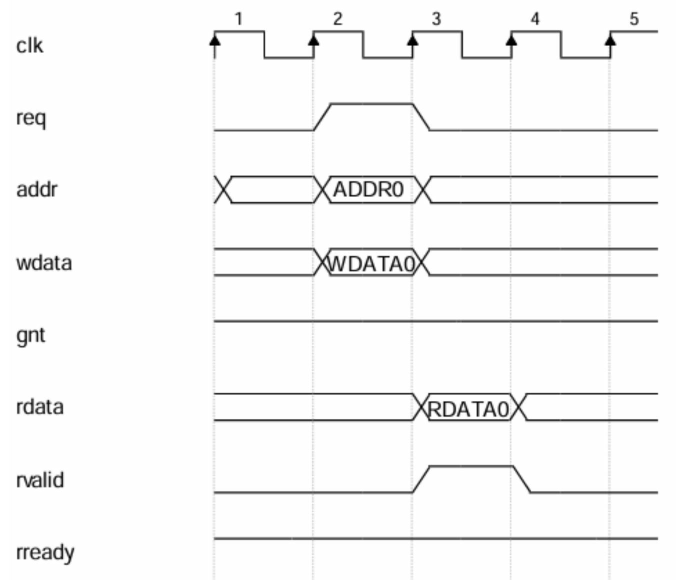


Figure 2.1: Example of a basic OBI transaction

The *OBI protocol* [14] is employed to manage data transmission and reception between any memory component, from the RAM banks within X-HEEP to the SPI-connected FLASH. The *Open Bus Interface* (OBI) protocol is a streamlined, point-to-point bus interface designed for managing data transactions between various components in the system. It consists of two main channels: the *Address Channel* (A) and the *Response Channel* (R). The Address Channel facilitates the transfer of *requests* (e.g., read/write commands), while the Response Channel handles the corresponding data *responses*.

A typical OBI transaction consists of two phases: the address phase, where the manager sends a request (e.g. data address and control signals), and the response phase, where the subordinate provides the data or an acknowledgment.

In the *address phase*, the manager initiates a transaction by asserting the request signal (*req*) to indicate that the address phase signals, including address (*addr*), write data (*wdata*), write enable (*we*), and control signals (be, aid, prot, etc.), are valid.

The subordinate then signals its readiness to accept these signals by setting the grant signal (*gnt*) high. The address phase completes on the rising edge of the clock when both *req* and *gnt* are high.

Following the address phase, the response phase begins when the subordinate asserts the response valid signal (*rvalid*), indicating that the response data (*rdata*), along with other response signals (err, rid, rchk, etc.), is ready. The manager, upon asserting the ready signal (*rready*), completes the response phase. The transaction concludes on the rising edge of the clock when both *rvalid* and *rready* are high.

This handshake process ensures synchronized communication between units, minimizing the risk of data corruption or misalignment.

2.2 Designing a Multichannel DMA System for X-HEEP

2.2.1 The Challenge of Handling Multiple Data Streams in Edge AI

In Edge AI, data is typically gathered from multiple sensors, each providing a continuous stream of information. When these data streams converge on a single edge device, such as an X-HEEP-based system like HEEPocrates [**1.2.3**], the device's CPU is often tasked with managing the flow of data from each sensor.

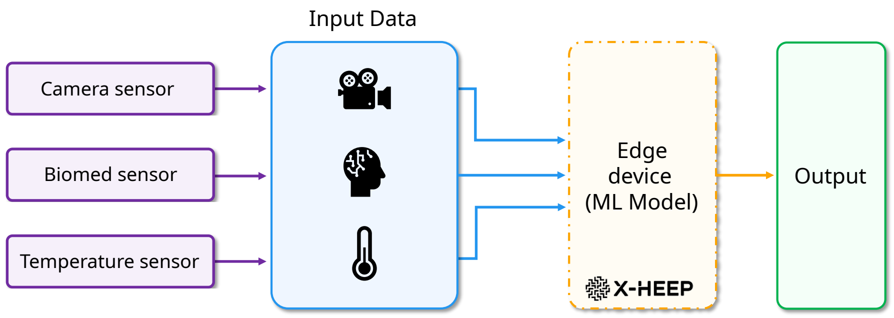


Figure 2.2: Structure of a general Edge AI system, with a variety of sensors connected to the edge device

This constant handling of input can create a significant bottleneck. Instead of focusing on the computational tasks required for data processing and inference, the CPU must allocate considerable resources to managing and synchronizing the sensor data. As a result, the efficiency of the system is reduced, with processing power diverted away from performing the critical AI algorithms.

This challenge highlights a fundamental problem in Edge AI systems, where managing concurrent data streams can degrade overall performance and delay real-time decision-making.

2.2.2 Structural Description

To address the previously mentioned limitation, this thesis introduces a novel *Multichannel DMA System*, specifically developed for the X-HEEP microcontroller. Before this work, the DMA system within X-HEEP was limited to a simple monochannel design.

The development of the multichannel system presented here marks a significant contribution, enhancing the platform's capacity to handle multiple concurrent data streams efficiently.

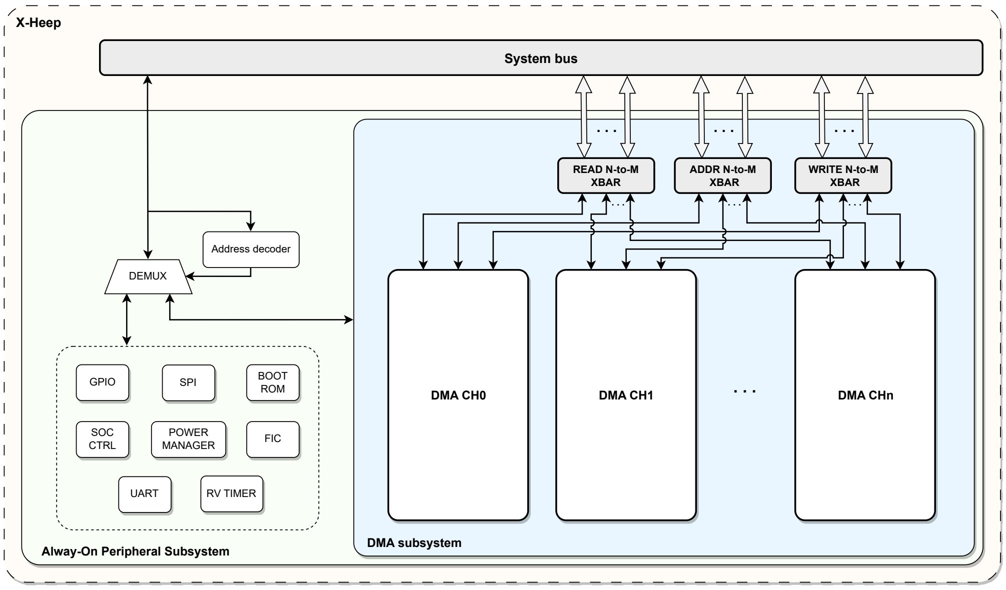


Figure 2.3: Structural Overview of the proposed Multichannel DMA System

The DMA subsystem consists of a customizable number of channels, each representing an identical instance of a shared base design. Additionally, the unit is equipped with a configurable number of system bus master ports to scale bandwidth linearly, barring any bus conflicts.

The entire X-HEEP platform is developed using *SystemVerilog* as the hardware description language, and the DMA is no exception.

In practical terms, the *dma_subsystem.sv* file is responsible for generating the DMA instances, based on parameters extracted from the project's general configuration file, *mcu_cfg.hjson*, of which an extract is reported in Code **2.1**. This configuration file specifies, for each unit, the base address and length of its memory map, along with parameters that can be utilized by both hardware design and software applications.

Furthermore, depending on the master port configuration, specific *OBI crossbars* are generated and connected to the bus master ports and the appropriate DMA channel instance ports, ensuring efficient data flow across the system.

The configuration of the DMA's *master ports* is managed through two parameters:

- *Number of master ports*
- *Maximum number of DMA channels per master port*

Listing 2.1: Code snippet of mcu_cfg.hjson

```

{

cpu_type: cv32e20

linker_script: {
    stack_size: 0x800,
    heap_size: 0x800,
}

debug: {
    address: 0x10000000,
    length:  0x00100000,
},

ao_peripherals: {
    address: 0x20000000,
    length:  0x00100000,
    soc_ctrl: {
        offset:  0x00000000,
        length:  0x00010000,
        path:    "./hw/ip/soc_ctrl/data/soc_ctrl.hjson"
    },

    [...]

    dma: {
        offset:  0x00030000,
        length:  0x00010000,
        ch_length:    0x100,
        num_channels:   0x4,
        num_master_ports: 0x2,
        num_channels_per_master_port: 0x2,
        path:    "./hw/ip/dma/data/dma.hjson"
    },
```

While the first parameter is straightforward, the second requires a more detailed explanation. Consider a project requiring a configuration with four channels and two master ports. There are two possible ways to connect the four channels to the two ports:

- CH0, CH1 connected to port 0 and CH2, CH3 connected to port 1
- CH0, CH1, CH2 connected to port 0 and CH3 connected to port 1

In the first configuration, each master port is connected to two channels, which corresponds to a channels-per-master-port ratio of 2. Conversely, the second configuration has a ratio of 3, where the first three channels are connected to port 0, and the remaining channel is connected to port 1.

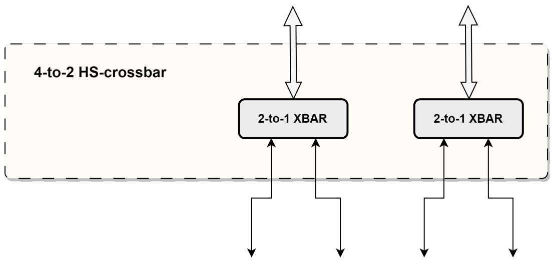


Figure 2.4: DMA-to-bus connection, composed with two 2-to-1 OBI crossbars

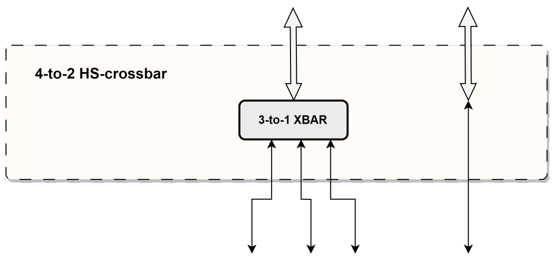


Figure 2.5: DMA-to-bus connection, composed with a 3-to-1 OBI crossbar and a single direct connection

While the first configuration is a general-purpose, balanced setup, the second configuration may be more suitable for applications requiring a low-latency channel for high-priority tasks. This mechanism ensures maximum flexibility, allowing users to tailor the DMA subsystem to their specific requirements, optimizing for both area and performance.

From a functional perspective, utilizing the multichannel DMA is straightforward. Each DMA channel is accessible through software via a structure generated by OpenTitan's *regtool*.

This tool uses an HJSON configuration file to describe the registers of a unit, specifying attributes such as size, name, software/hardware access privileges, reset types, and other relevant parameters. Regtool then generates SystemVerilog files for integration into the main design, along with header files that aid in the development of a Hardware Abstraction Layer (HAL).

Among these there is a C structure that provides a clean and efficient mean to access the unit's registers. For instance, a register can be accessed as shown in Code **2.2**.

Listing 2.2: Register access via HAL

```

    unit->SRC_PTR = (uint32_t) input_ptr;
```

In the case of the DMA, all channels are identical and share the same set of registers. To facilitate access to individual channels, a flexible approach has been implemented, as shown in Code **2.3**, where each DMA channel can be accessed by calculating the appropriate base address.

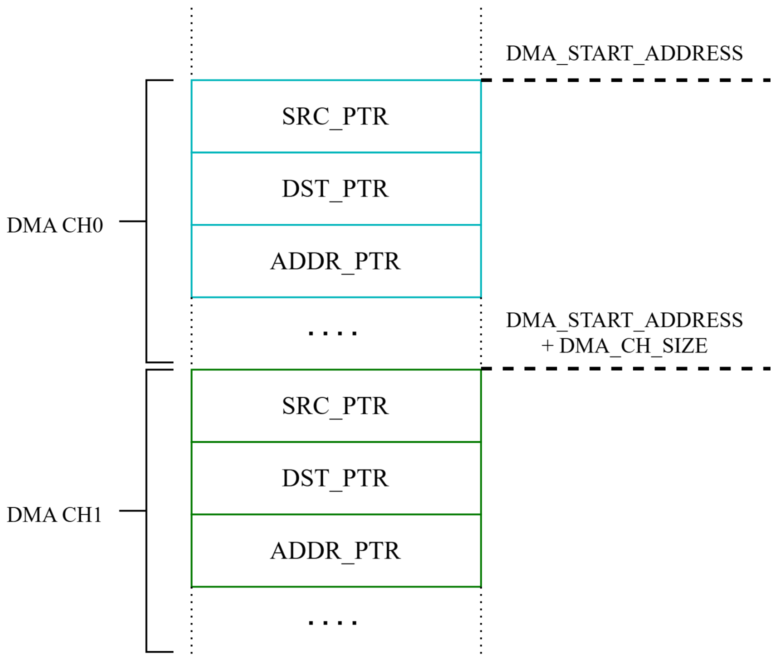


Figure 2.6: Memory map scheme of the DMA System.

Listing 2.3: Macro for accessing DMA channels

```c

#define dma_peri(channel) ((volatile dma *) (DMA_START_ADDRESS
+ DMA_CH_SIZE * channel))
```

Thus, the registers for each DMA channel can be accessed independently. For example, accessing and configuring channel 2 can be done as shown in Code **2.4**.

Listing 2.4: Accessing DMA CH2

```c

dma_ch_2 = dma_peri(2);
dma_ch_2->SRC_PTR = (uint32_t) input_ptr;
```

2.3 Extending X-HEEP's DMA System with Advanced Transactions Capabilities

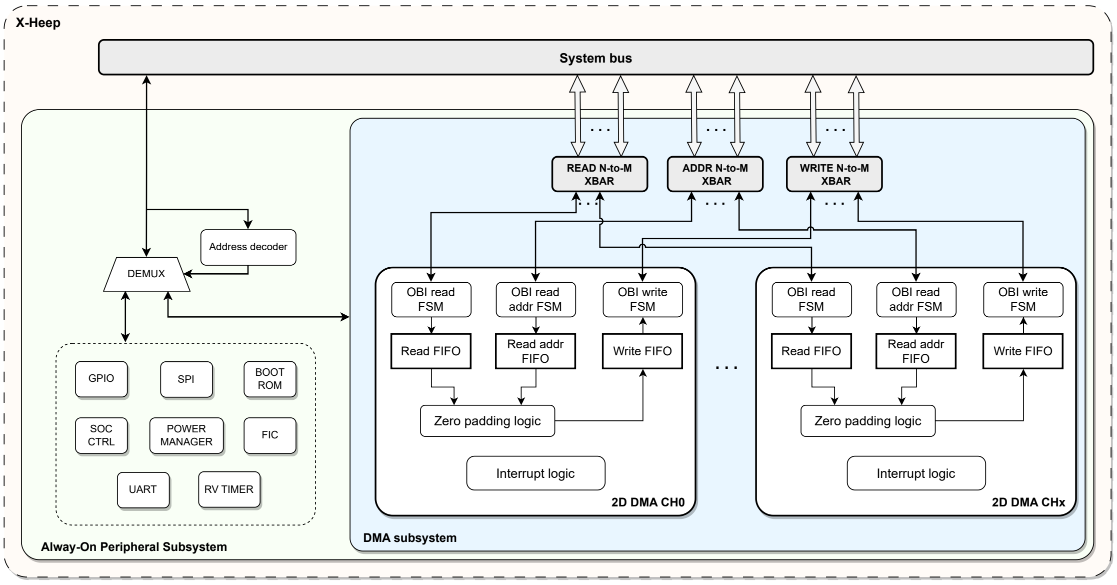


Figure 2.7: Structural Overview of the Proposed DMA System with Advanced Transaction Channels

2.3.1 Complex Data Manipulation in GEMM Accelerators

General Matrix Multiplication (*GEMM*) accelerators, such as systolic arrays and in-memory computing described in [**1.2.2**], are highly efficient at performing large-scale matrix computations.

However, to fully utilize their potential, complex data manipulation is often required. These accelerators rely on carefully structured data movement patterns to feed operands into the computation units. This involves *reshaping*, *tiling*, or streaming data in specific ways to match the requirements of the hardware architecture, possibly becoming a performance bottleneck.

The need for extensive data rearrangement can hinder overall performance, as it may require significant CPU intervention to manage the flow of data between memory and the accelerator.

Consequently, *high CPU utilization* becomes a concern, as the CPU must spend a considerable amount of time coordinating data movement instead of focusing on higher-level processing tasks. This inefficiency can offset the performance gains offered by GEMM accelerators, particularly in real-time or resource-constrained environments.

2.3.2 Structural Description

To further enhance X-HEEP's data management capabilities, the core DMA unit that forms each channel has been completely redesigned in this thesis.

This redesign introduces advanced features such as *2D transactions*, along with *zero-padding*, *transpositions*, and *sign extensions* performed on-the-fly, significantly optimizing the complex data manipulations required by edge computing accelerators.

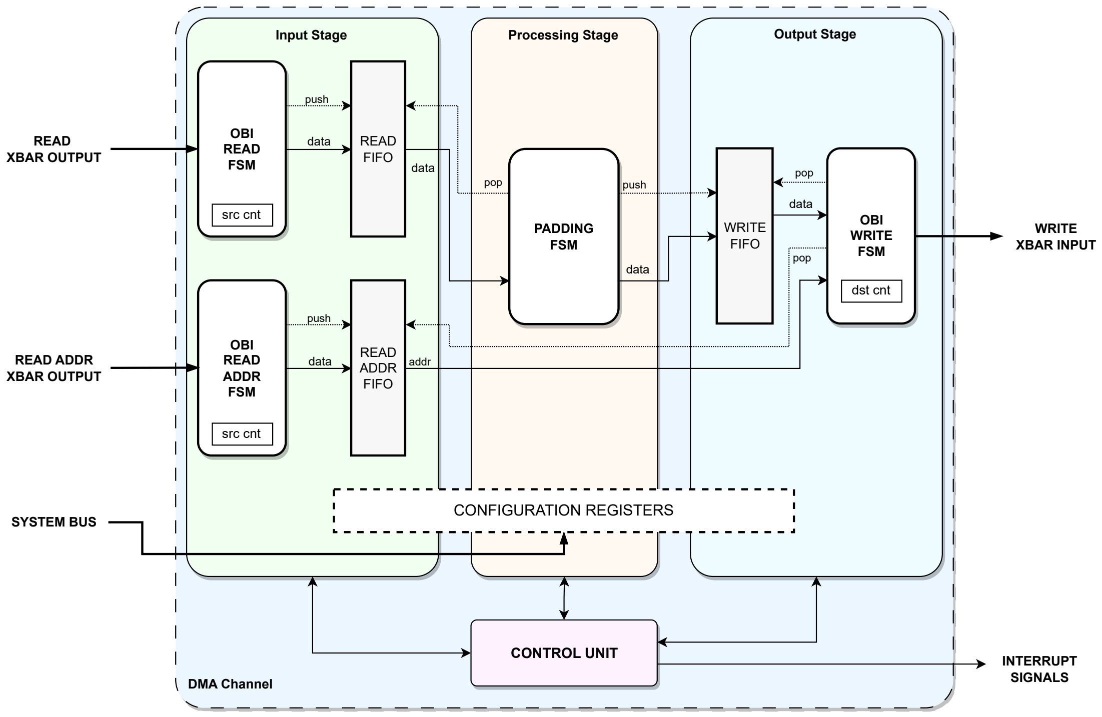


Figure 2.8: Structural Overview of the New DMA Channel

As shown in Figure **2.8**, each DMA Channel is composed of three main stages:

- *Input Stage*
- *Processing Stage*
- *Output Stage*

These stages are managed by a central Control Unit (CU), which is responsible for coordinating each stage and asserting interrupt signals when enabled. The CU and all stages share access to a set of configuration registers, allowing the DMA Channel to be configured and enabling communication of important information to the CPU. These registers are generated using OpenTitan's *regtool* and a specific HJSON configuration file.

Input Stage

The Input Stage consists of two aggregates of *Finite State Machines* (FSMs) that manage the OBI interface with the crossbars connecting the DMA System to the System Bus, as shown in figure **2.4**.

The *OBI READ FSM* issues read requests based on a source address calculated according to the transaction parameters, specifically the increments for the first and second dimensions. The *read_ptr_reg* register stores this computed source address and increments whenever the *READ FSM* receives an OBI *GNT* signal. A secondary signal, *read_ptr_valid_reg*, ensures synchronization between *read_ptr_reg* and OBI's *RVALID*, which indicates that the requested data is valid. This *RVALID*-synchronized value is then used to perform data shifts necessary for handling different data types, as detailed in [**2.3.2**].

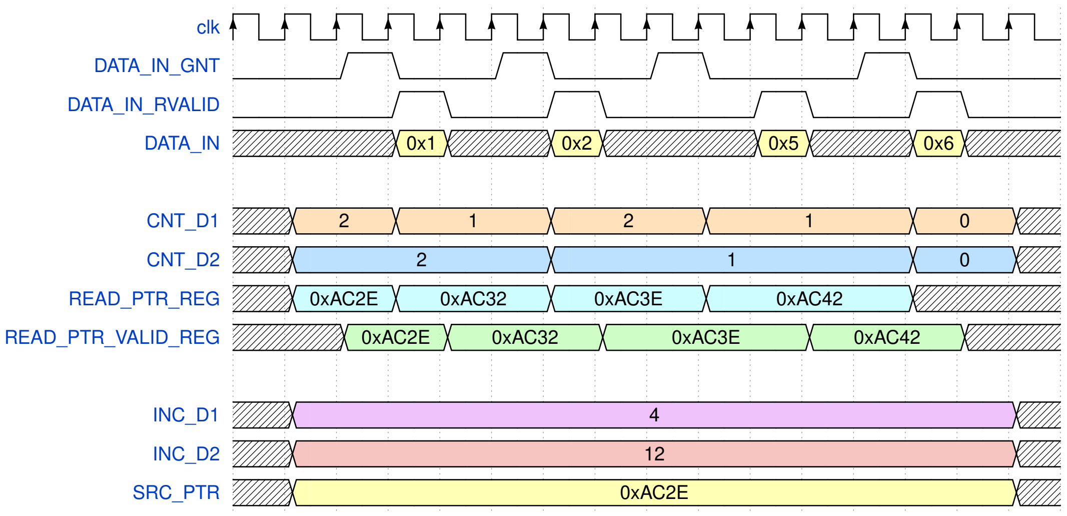


Figure 2.9: Details on the *READ FSM* main signals, including counters and source registers. On the top, the most critical OBI signals, while on the bottom the values of a few configuration registers. More details on the meaning behind these parameters can be found in the functional description section [**2.3.3**].

To track the progress of the transaction, two *counters* are used, one for each dimension. When a request is granted, the first dimensinon (D1) counter is decremented, and upon reaching zero, it resets while the second dimension (D2) counter is decremented. When both counters reach zero, all necessary elements have been copied and the FSM enters the idle state. Each data read is pushed in the *READ FIFO*, ready to be processed by the *PADDING FSM*.

Figure **2.9** illustrates the key signals of the *READ FSM* during a 2D transaction that extracts the top-left 2x2 sub-matrix from a 4x4 matrix, as shown in matrix **2.10**.

$$\begin{matrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{matrix} \quad \Rightarrow \quad \begin{matrix}1 & 2\\ 5 & 6\end{matrix}$$

Figure 2.10: 4x4 matrix and the extracted 2x2 sub-matrix.

Matrix transposition is performed solely by the READ FSM. To enable this feature, the CPU has to set a specific control bit, *dim_inv*, which stands for *"dimensionality inversion"*.

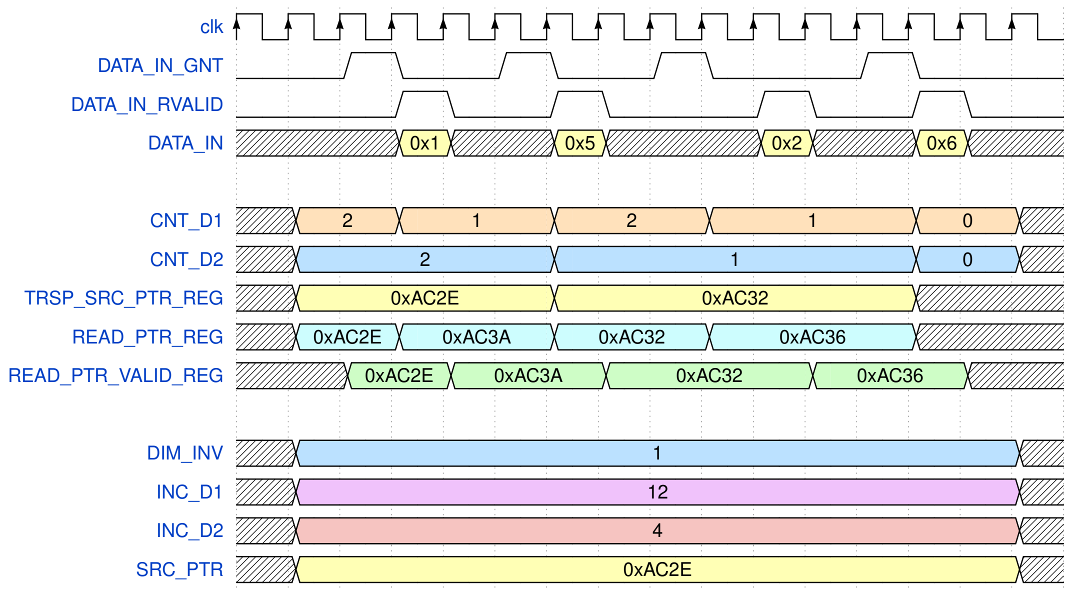


Figure 2.11: Details on the *READ FSM* main signals during a matrix transposition. Note the changes of the source address values with respect to figure **2.9**.

To handle matrix transposition, the source pointer updates differ from the standard incrementing pattern. In this case, *inc_d1* is set by the CPU to the length of a row in bytes, representing the offset needed to move to the next element in the same column. As each element is copied, the source pointer increments by *inc_d1* bytes, effectively traversing down the column.

A secondary register, *trsp_src_ptr_reg*, is initialized with the starting source address of the transaction—the address of the first element in the matrix. When an entire column has been copied, the source pointer resets to the value in *trsp_src_ptr_reg*. Then, *trsp_src_ptr_reg* is incremented (e.g., by *inc_d2* element size) to point to the first element of the next column.

This mechanism ensures that the FSM begins copying from the first element of each new column. The source address moves down a column by increments of *inc_d1* bytes and then jumps back to the top of the next column when a column is completed. Figure **2.11** represent the behaviour of the FSM's main signals during the transposition of the same 2x2 matrix of the past example, reported in matrix **2.12**.

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{matrix} \Rightarrow \begin{matrix} 1 & 5 \\ 2 & 6 \end{matrix}$$

Figure 2.12: 4x4 matrix and the extracted 2x2 sub-matrix, transposed.

The Input Stage also includes the *READ ADDR FSM*, which is used when the DMA operates in *Address Mode*. In this mode, the DMA fetches destination pointers from an address list stored in memory, rather than computing them.

The *READ ADDR FSM* generates read requests for these addresses and pushes the received data into the *READ ADDR FIFO*, which is later accessed by the *WRITE FSM* as addresses used to write data to the correct location.

Processing Stage

The Processing Stage is central to the functionality of the new DMA System. It features the *PADDING FSM*, which moves data from the *READ FIFO* to the *WRITE FIFO*, applying necessary transformations according to the transaction parameters.

Similar to the Input Stage, the *PADDING FSM* monitors the transaction status using counters, accounting for both the padding parameters and transaction sizes. Padding is achieved by refraining from popping data from the *READ FIFO* and instead pushing zeros into the *WRITE FIFO*, based on the value of the *pad_on* signal, which is controlled by a dedicated FSM whose states are determied on the counters value.

Once both dimensional counters reach zero, the FSM enters the idle phase.

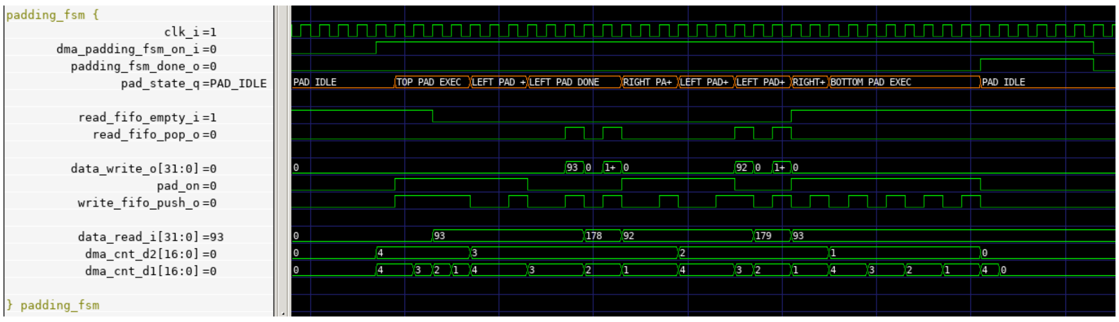


Figure 2.13: A section of a Verilator simulation focusing on the main signals of the *PADDING FSM*. The FSM state progression is highlighted in orange, which determines the value of *pad_on*, setting the *WRITE FIFO* input to 0.

Output Stage

The Output Stage mirrors the Input Stage but performs the opposite function. It consists of the *WRITE FSM*, which writes data by popping it from the *WRITE FIFO*. Like other stages, the *WRITE FSM* tracks the status of the transaction using counters and computes on its own the destination pointer for each element.

In *Address Mode*, the destination pointer is instead sourced from the *READ ADDR FIFO*.

When both dimensional counters reach zero and the *WRITE FIFO* is empty, the *WRITE FSM* generates a completion signal, allowing the CU to trigger an interrupt, if enabled.

~

Managing Datatypes and Sign Extension

In both the *Input* and *Output Stages*, a sophisticated mechanism is implemented to handle different data types, such as words, half-words, and bytes. The challenge arises from the fact that the RAM of X-HEEP is *word-addressable*, like most modern SRAMs, meaning it can only store and output 32-bit words.

Consequently, the memory discards the lower 4 bits of the address; for instance, the addresses `0x0ABC3` and `0xABC1` both refer to the same word.

During the read operation, the *READ FSM* always accesses the full word, as addressing anything smaller would have no effect due to the word-addressable architecture. Once the word is retrieved, a shift operation ensures that the relevant data type (whether it be a byte or half-word) is correctly positioned in the least significant bits (LSB).

This is accomplished by a dedicated FSM in the Input Stage, which operates based on the address offset, i.e. the last four bits. When the address is word-aligned (offset 0), no shift is necessary. However, for offsets such as 1, indicating that the desired data is the second byte, the FSM shifts the corresponding byte into the LSB.

For example, in the case of `0xABCDEF39`, if the second byte is needed, the FSM shifts `EF` to replace `39`, yielding `0xABCDEFEF`.

The complete switch-case logic is shown in Code **2.5**.

Listing 2.5: Snippet of the Input Stage: a compact FSM shifts the input data based on the source address offset, which is influenced by the data type being read. The signal *fifo_input* holds the processed data that is pushed into the *READ FIFO*.

```systemverilog

always_comb begin : proc_input_data

  fifo_input[7:0]   = data_in_rdata[7:0];
  fifo_input[15:8]  = data_in_rdata[15:8];
  fifo_input[23:16] = data_in_rdata[23:16];
  fifo_input[31:24] = data_in_rdata[31:24];

  case (read_ptr_valid_reg[1:0])
    2'b00: ;
    2'b01: fifo_input[7:0] = data_in_rdata[15:8];

    2'b10: begin
      fifo_input[7:0]  = data_in_rdata[23:16];
      fifo_input[15:8] = data_in_rdata[31:24];
    end

    2'b11: fifo_input[7:0] = data_in_rdata[31:24];
  endcase
end
```

Figure **2.14** illustrates an example of a 4-element 1D copy with bytes as the source datatype. By combining the *fifo_input* value with the last two bits of *read_valid_ptr_reg*, the effect of this mechanism becomes clear: the byte indexed by these two bits is always placed as the LSB in *fifo_out*.

The *Padding Stage* is entirely unaware of the input and output data types, simplifying its architecture and improving the robustness of the entire DMA Channel.

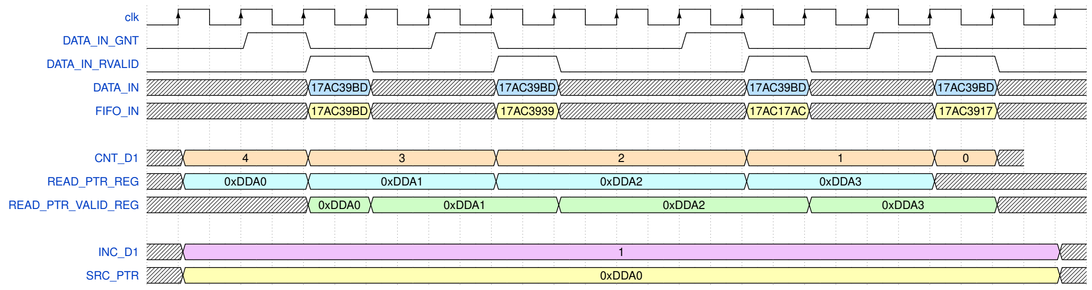


Figure 2.14: Details on the *READ FSM* main signals in a 1D transaction copying *bytes*.

In the *Output Stage*, the data type becomes critical again, as this is where *sign extension* is applied, when enabled. The situation is similar to the *Input Stage* but in reverse. The RAM only accepts full 32-bit words, making it necessary to handle sub-word writes, such as bytes or half-words, carefully.

This is accomplished in two steps: first, by computing the appropriate byte enable mask based on the *output data type* and communicating it to the RAM via the *byte enable* signal provided by the OBI protocol, and second, by correctly shifting the data into the appropriate position within the word.

During this process, if the output data type is larger than the source type and sign extension is enabled, the FSM ensures that the sign bit is properly extended to the higher bits of the destination. For example, when copying signed byte data (such as quantized neural network weights) into half-words or words, the sign extension can be performed to ensure that arithmetic operations on the larger data type behave correctly.

The FSM responsible for this in the Output Stage operates similarly to the one described in Code **2.5**, but with the added responsibility of handling sign extension.

The Role of FIFOs in a DMA Channel

In the new DMA Channel architecture, FIFOs play a critical role in ensuring that stages are *decoupled*, allowing for efficient data synchronization between stages, especially when padding is required. Each FSM that interacts with a FIFO must ensure that the FIFO is not empty before popping data and that it is not full before pushing new data into it. This coordination ensures smooth data flow through the stages without overflow or underflow conditions.

Furthermore, the FIFO sizes for each DMA channel can be individually configured, allowing the architecture to be tailored to specific application needs. This flexibility is achieved through a parameterization mechanism, as demonstrated in Code **2.6** from the *dma_subsystem.sv* file, where the DMA channels are instantiated. By enabling the EN_SET_FIFO_CH_SIZE definition, the size of the FIFOs for each channel can be adjusted using a predefined configuration array.

Some applications might benefit from larger FIFOs, allowing more data to be buffered when the bus is heavily utilized, or when the target peripheral, such as an SPI interface, operates at a slower speed. Conversely, other applications may not require large FIFOs, allowing for area savings by reducing FIFO size.

A hybrid system, where certain channels have larger FIFO sizes while others have smaller ones, can optimize performance by catering to both high-throughput and low-latency requirements. This flexible configuration allows the DMA system to efficiently handle varying data workloads and peripheral speeds.

Listing 2.6: Snippet from *dma_subsystem.sv*: Configurable FIFO sizes for each DMA channel.

```

//`define EN_SET_FIFO_CH_SIZE;

`ifdef EN_SET_FIFO_CH_SIZE

localparam int unsigned LARGE_FIFO_CH_SIZE = 8;
localparam int unsigned MEDIUM_FIFO_CH_SIZE = 4;
localparam int unsigned SMALL_FIFO_CH_SIZE = 2;

typedef enum {L, M, S} fifo_ch_size_t;

localparam fifo_ch_size_t FIFO_CH_ARRAY [core_v_mini_mcu_pkg::
 DMA_CH_NUM] = '{L, M, M, S};

`endif
```

Synchronizing Peripherals and DMA: the Trigger System

In *memory-to-peripheral* and *peripheral-to-memory* operations, it is common for the peripheral to have a response time that cannot keep pace with the system clock. For example, SPI typically transmits and receives data with a period of approximately 30 clock cycles.

This disparity in timing necessitates a communication mechanism between the DMA subsystem and the peripheral, allowing the DMA operations to be paused based on the peripheral's state. These control signals are referred to as *triggers*.

Triggers can be utilized in both directions, whether the peripheral is writing data to memory via the DMA or the DMA is reading data from the peripheral. The DMA can be configured to respond to these triggers by enabling the corresponding slot via software, using the DMA HAL.

2.3.3 Functional Description

As previously discussed in [**2.3.2**], the DMA supports both 1D and 2D transactions, allowing it to handle complex data manipulations such as matrix operations, padding, and sign extension.

A DMA transaction involves copying data from a source pointer to a destination pointer, which can either be another memory location or a peripheral buffer. The DMA operates in different modes, including *single mode*, *circular mode*, and *address mode*, each tailored to specific use cases.

To initiate a transaction, the configuration must first be loaded into the DMA registers. Once the first dimension size is written to its respective register, the transaction begins.

During execution, the DMA System can handle multiple transactions concurrently, provided they target different channels. A key feature is its ability to automatically re-launch transactions in *circular mode*, useful for continuous data streams from peripherals like SPI, where the peripheral's response time is much slower than the system clock. In cases like this, *triggers* [**2.3.2**] are used to synchronize DMA operations with peripheral readiness.

Increment System in DMA Transactions

The DMA increment system allows for flexible data transfers, making it possible to achieve both *contiguous* and *non-contiguous* read and write operations in 1D and 2D transactions. By carefully setting the increments for the source and destination addresses, the DMA can efficiently handle various data manipulation tasks, such as sub-array extractions or matrix operations.

1D Transactions

In a 1D transaction, the increment determines how much the DMA should move the read and write pointers after each data transfer. This feature is particularly useful for non-contiguous data handling, where only specific parts of the data need to be copied. For example, consider an array of 4 word-type elements (each word being 4 bytes), as reported in Code **2.7**.

Listing 2.7: Example of a 4 word array

```
    | 0xBADC0DE5 | 0xBADC0FFEE | 0xC0FFEEED | 0xFEEDBEEF |
```

To copy only the first 2 bytes from each word, the following settings would be required:

- Set the data type of source and destination to half-word (2 bytes).
- Set the source increment to 4 bytes.
- Set the destination increment to 2 bytes.
- Set the 1D transaction size to 4 data units.

With these configurations, after each read operation, the DMA will increment the read pointer by 4 bytes (skipping to the next word), and the write pointer will be incremented by 2 bytes. This allows the DMA to efficiently copy non-contiguous data.

2D Transactions

In 2D transactions, the DMA requires a second increment value to handle multi-dimensional data structures like matrices. The second increment is used to define how many data units the DMA should *"skip"* to move from one row to the next, allowing for matrix manipulations in a single DMA transaction.

Consider a 4x4 matrix from which we want to extract a 2x2 sub-matrix, placed on the top-left corner, as shown in matrix **2.15**.

$$\begin{matrix} 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 \\ 0 & 2 & 4 & 6 \end{matrix} \Rightarrow \begin{matrix} 3 & 5 \\ 2 & 4 \end{matrix}$$

Figure 2.15: 4x4 matrix and the extracted 2x2 sub-matrix.

To achieve this, the following increment values are set:

- **Source increments**:
  - 1D increment: 1 word (4 bytes)
  - 2D increment: 3 words (12 bytes)
- **Destination increments**:
  - 1D increment: 1 word (4 bytes)
  - 2D increment: 1 word (4 bytes)

In this case, the 1D increment specifies the step within a row, while the 2D increment tells the DMA how far to jump to move to the next row of the matrix. By setting these increments correctly, the DMA can skip over unnecessary elements and extract the desired submatrix.

Moreover, by configuring the 2D increments, the system can also handle *non-contiguous* 2D reads and writes, allowing for even more complex data manipulations such as skipping rows or columns, achieving a *stride*.

Configuring Padding in DMA Transactions

Additionally, the DMA channel can be configured to add *zero padding* to extracted data, an essential feature for handling 2D data in machine learning tasks like pattern recognition, as it is used to increase the detail of the edges of the image. The padding feature allows users to add extra rows or columns of zeros around the input matrix, ensuring alignment or preparing the data for further processing. The padding is controlled by four parameters, which define the amount of padding applied to each side of the extracted matrix:

- *Top*
- *Bottom*
- *Left*
- *Right*

It is essential to note that the padding is applied *conceptually* only after the matrix has been extracted, meaning that the padding parameters refer solely to the extracted submatrix, not the original source matrix.

For example, let's consider a 2x3 matrix with padding applied on all sides, as shown in matrix **2.16**. In this case, zeros (represented by T, B, L, and R) are added around the original data (x), resulting in a padded matrix.

$$\begin{matrix} T & T & T & T & T \\ L & x & x & x & R \\ L & x & x & x & R \\ B & B & B & B & B \end{matrix}$$

Figure 2.16: Padding example of a 3x2 matrix, explaining the meaning of *"top"*, *"bottom"*, *"left"* and *"right"* padding in the DMA System.

To further illustrate, matrix **2.17** revisits the example of extracting a 2x2 submatrix from a 4x4 matrix, but this time with a left and top padding of 1 element.

$$\begin{matrix} 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 \\ 0 & 2 & 4 & 6 \end{matrix} \Rightarrow \begin{matrix} 0 & 0 & 0 \\ 0 & 3 & 5 \\ 0 & 2 & 4 \end{matrix}$$

Figure 2.17: Padding example of a 2x2 matrix extracted from a 4x4 matrix, with 1 layer of zero padding on top and on the left.

DMA Hardware Abstraction Layer

In order to facilitate the development of DMA-based applications, the existing DMA *Hardware Abstraction Layer* (HAL) has been updated and expanded in this thesis to support the newly introduced features.

The HAL is structured around three core functions: *dma_validate_transaction()*, *dma_load_transaction()*, and *dma_launch()*, each responsible for managing different aspects of DMA operations. Additionally, the initialization function, *dma_init()*, resets transaction structures and clear DMA registers for each channel, ensuring that the DMA subsystem starts in a known state.

dma_validate_transaction()

The function *dma_validate_transaction()* ensures that the configuration of a DMA transaction is correct by performing thorough *sanity* and *integrity checks*. These checks validate target configurations, verify data alignment, and inspect increment, padding, trigger, and mode settings to prevent potential execution errors. The function returns configuration flags that provide feedback on any detected issues, allowing developers to address them before proceeding.

dma_load_transaction()

Once a transaction is validated, the *dma_load_transaction()* function configures and loads the transaction into the appropriate DMA channel. This step involves setting pointers, increments, padding, and operation modes by writing the relevant registers, while ensuring that no other transaction is currently running.

Crucially, the transaction is not launched immediately; the *SIZE_D1* register, which starts the transaction, is left unwritten at this stage to allow for controlled launching later.

dma_launch()

Finally, *dma_launch()* is responsible for initiating the pre-configured DMA transaction. It checks for any ongoing transactions and, once the path is clear, writes to the *SIZE_D1* register to start the operation.

If the transaction is set to trigger an interrupt upon completion, the function will block until the interrupt is received, ensuring that the CPU can wait for the operation to complete without further intervention. The HAL ensures that DMA transactions are handled robustly, with built-in validation and error handling to support the new DMA features introduced in this work.

Interrupt Request Handler

Due to hardware constraints, the DMA subsystem has only a single fast interrupt line, regardless of how many channels are present in the system. This fast line is managed by the X-HEEP's *FIC* (Fast Interrupt Controller, an IP provided by *OpenHW*) and used to signal the transaction done interrupt, while the window done interrupt is managed from a slower unit, the *PLIC* (Platform-Level Interrupt Controller).

To determine which channel raised the interrupt, the DMA HAL checks the *interrupt flag register* (IFR) of each channel. This register is set when a transaction completes and the interrupt is enabled, and it is cleared once the CPU reads its content.

When the IFR for a channel is found to be high, the HAL invokes the weakly defined *dma_intr_handler_trans_done()* function, passing the channel ID as an argument. This default handler can be customized by the user to implement specific actions when a transaction is completed.

To optimize interrupt handling, the HAL also offers an additional customization mechanism to prioritize certain channels. By defining the constant *DMA_HP_INTR_INDEX* in *dma.h* (the DMA HAL's header file), channels with IDs less than or equal to this index are treated as *high-priority*.

When one of these high-priority channels raises an interrupt, the *handler_irq_dma()* function immediately calls the user-defined interrupt handler and *exits*, bypassing the loop that checks other channels. This ensures that high-priority channels are serviced *faster*, but could potentially *starve* low-priority channels if high-priority interrupts are frequent and the associated handler performs long tasks.

To prevent this, two design choices are available: minimizing the workload within the IRQ handler itself (a universal good design practice) and setting the *DMA_NUM_HP_INTR* parameter, which limits the number of consecutive interrupts a high-priority channel can trigger before the system checks for pending interrupts from low-priority channels.

The loop-based mechanism for handling interrupts is shown in Code **2.8** from the DMA's HAL *dma.c*.

This interrupt management system ensures the DMA remains responsive and adaptable in real-time applications. Furthermore, this mechanism is also applicable to *window interrupt*s, which trigger when a predefined amount of data has been transferred during a transaction.

The integration of prioritization and configurable limits enhances the robustness of the DMA system by allowing developers to tailor interrupt handling to the specific needs of their application, ensuring a balance between responsiveness and fairness across DMA channels.

Listing 2.8: Interrupt handling mechanism in the DMA HAL with customizable prioritization

```

void fic_irq_dma(void) {

    for (int i = 0; i < DMA_CH_NUM; i++)
    {
        if (dma_subsys_per[i].peri->TRANSACTION_IFR == 1)
        {
            dma_subsys_per[i].intrFlag = 1;
            dma_intr_handler_trans_done(i);

            #ifdef DMA_HP_INTR_INDEX

            #ifdef DMA_NUM_HP_INTR

            if (i <= DMA_HP_INTR_INDEX && dma_hp_tr_intr_counter < DMA_NUM_HP_INTR)
            {
                dma_hp_tr_intr_counter++;
                return;
            }
            else if (i > DMA_HP_INTR_INDEX)
            {
                dma_hp_tr_intr_counter = 0;
            }

            #else

            if (i <= DMA_HP_INTR_INDEX)
            {
                return;
            }
            #endif

            #endif
        }
    }
return;
}
```

2.4 Developing the Always-On Peripheral Bus to Support DMA-Based Accelerators

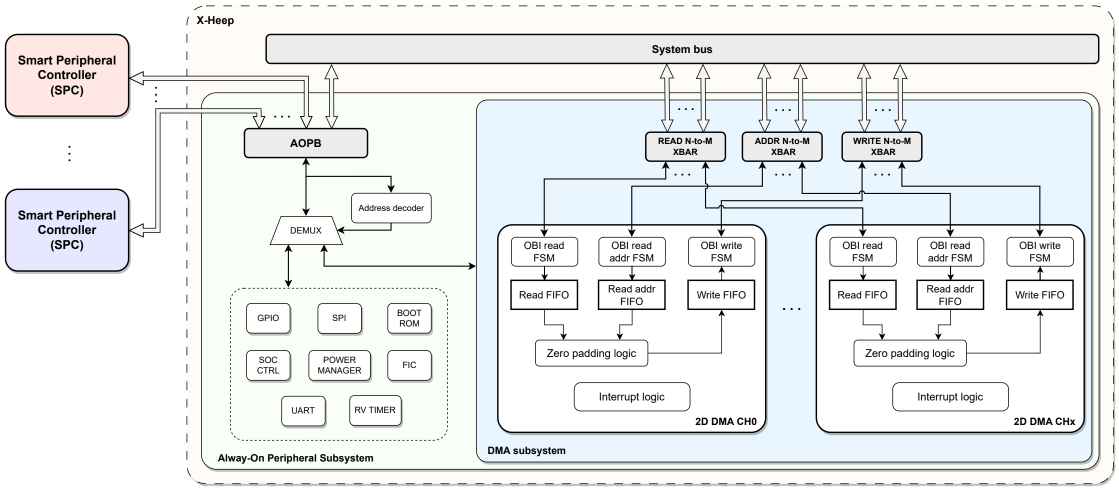


Figure 2.18: Structural Overview of the AOPB embedded into the Always-On Peripheral System.

2.4.1 DMA-Based Accelerators to Reduce CPU Utilization

During the development of the Advanced DMA System for X-HEEP, it became apparent that *CPU utilization* remained a significant contributor to memory overhead. This issue is especially pronounced in applications that require the continuous configuration of DMA channels. In these cases, the CPU must frequently set up new DMA transactions and wait for their completion, which leads to prolonged periods where the CPU is occupied with managing data transfers.

As highlighted in the Multichannel chapter **2.2**, this is one of the key limitations in edge computing that this thesis seeks to address.

A prime example of such an application is the *im2col* reshaping transformation. Performing this operation requires executing multiple DMA runs proportional to the number of channels, the batch size, and the filter dimensions. When X-HEEP is employed to run the inference of a convolutional neural network (CNN) algorithm, the CPU is stalled by the transformation process, unable to process data until it completes.

Even though the advanced DMA system reduces the overall number of runs needed for operations like *im2col*, the CPU remains bogged down by the need to repeatedly configure the DMA.

This situation highlights the necessity for a solution that allows the CPU to focus on data processing rather than data movement. Although the multichannel feature reduces the complexity of managing multiple data flows and the advanced DMA reduces the number of runs required for operations like the *im2col*, the fundamental issue of *prolonged CPU blockage* persists.

To address this limitation, the thesis explores the implementation of *DMA-based accelerators*. These accelerators are designed to autonomously access and configure the DMA controller to perform necessary data movements without constant CPU intervention. By offloading data management tasks to the accelerators, the CPU is freed to process data concurrently, thus enhancing system performance.

This strategy aligns with the ongoing trend in computing architecture towards heterogeneity, where specialized hardware components are leveraged to perform specific tasks more efficiently than general-purpose CPUs.

2.4.2 Structural Description

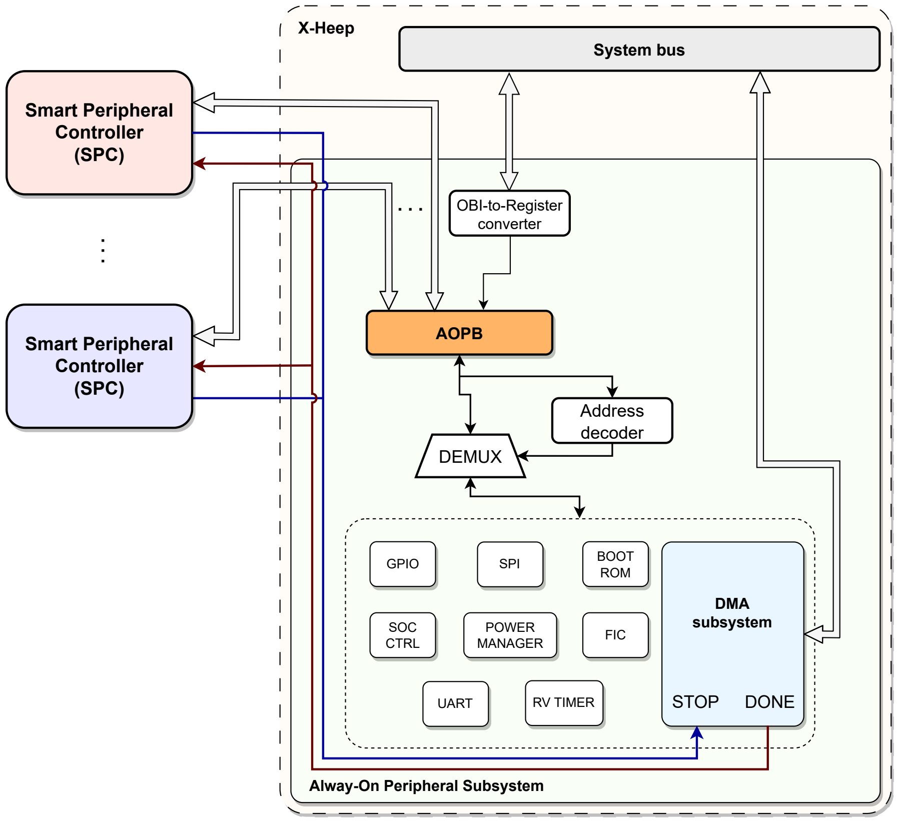

X-Heep
System bus
Smart Peripheral Controller (SPC)
Smart Peripheral Controller (SPC)
OBI-to-Register converter
AOPB
Address decoder
DEMUX
GPIO
SPI
BOOT ROM
SOC CTRL
POWER MANAGER
FIC
UART
RV TIMER
DMA subsystem
STOP
DONE
Alway-On Peripheral Subsystem

Figure 2.19: Structural Overview of the Always-On Peripheral Bus.

To alleviate the CPU bottleneck and support DMA-based accelerators, this thesis introduces a new Peripheral Bus specifically designed for efficient communication between accelerators and peripheral devices. Although X-HEEP already provides an extension interface via system bus master ports, utilizing this interface for accelerators to program the DMA's registers presents two significant drawbacks:

1. **Quadratic Cost of Adding Bus Masters**: Adding additional master ports to the system bus increases its complexity and area footprint, which grows quadratically with the number of masters due to the nature of crossbar buses. This not only consumes more silicon area but also complicates the bus architecture, making it less efficient for the intended purpose.

2. **Unnecessary Protocol Complexity**: The system bus employs the OBI protocol, which is designed for complex communication scenarios involving potential conflicts and advanced features. For the simple register writes required by the accelerators, this protocol introduces unnecessary complexity.

Implementing an OBI-compatible finite state machine (FSM) within the accelerators solely for peripheral register access adds unnecessary overhead and complexity, making it a suboptimal solution.

Listing 2.9: Register interface package.

```

    package reg_pkg;

      typedef struct packed {
        logic         valid;
        logic         write;
        logic [3:0]   wstrb;
        logic [31:0]  addr;
        logic [31:0]  wdata;
      } reg_req_t;

      typedef struct packed {
        logic         error;
        logic         ready;
        logic [31:0]  rdata;
      } reg_rsp_t;

    endpackage
}
```

To address these issues, the *Always-On Peripheral Bus* (AOPB) was developed and integrated into X-HEEP. The AOPB employs a simplified version of the OBI protocol, designed for efficient communication with peripheral registers. It seamlessly interfaces with the register top module generated by OpenTitan's *Regtool*, as it shares the same register interface used throughout the X-HEEP project, shown in figure **2.9**. Compared to OBI, this protocol uses a single *ready* signal in place of OBI's separate *rvalid* and *grant* signals.

Located within the Always-On domain, which includes critical peripherals such as the DMA System, Power Manager, Timer, and GPIO System, the AOPB provides a dedicated communication pathway for accelerators to interact with peripherals independently of the system bus.

External accelerators can access the AOPB through exposed ports, while the system bus remains connected to the AOPB for regular operations. This design offers several advantages over using the system bus for programming configuration registers.

1. **Increased Area Efficiency**: Crossbar buses consume area that grows *quadratically* with the number of masters:

   $$M^2 > M1^2 + M2^2, M1 + M2 = M$$

   $$M1 + M2 = M$$

   Partitioning communication into two smaller buses significantly reduces overall area consumption. By separating the CPU and accelerator communication onto distinct buses, the complexity and size of each bus is minimized, resulting in a more efficient hardware implementation.

2. **Simplified Arbitration and Protocol Optimization**: The arbitration system within the AOPB is simplified due to fewer masters and a less complex protocol. This leads to improved area and energy efficiency.

   Additionally, the streamlined protocol for peripheral register access allows the AOPB to optimize the FSMs used by accelerators, reducing latency and simplifying the accelerator architecture.

This inequality shows that the sum of the squared areas of two buses is less than the squared area of a single bus, which justifies splitting the system.

From a structural standpoint, the AOPB is inserted between the OBI-to-register converter and the register demux provided by PULP. This component uses the OBI's address to index the bus request among the different peripherals of the AO Subsystem.

The AOPB itself is composed of a register multiplexer developed by Florian Zaruba for the PULP Project. This IP is designed to handle multiple input ports, each representing a master needing access to peripheral registers. Its main components are:

- **Input Interface**: Receives register access requests from multiple accelerators, each with signals for address, data, write enable, and valid indicators.
- **Arbitration Mechanism**: Employs a round-robin scheme with non-starving, rotating priorities, or a priority-based method to grant access when multiple requests are active, ensuring fairness and preventing starvation.
- **Output Interface**: Forwards the selected request to the peripheral bus, following the simplified Open Bus Interface (OBI) protocol used in the AOPB.
- **Response Handling**: Routes responses, including data read and write acknowledgments, back to the correct accelerator.

The PULP's register multiplexer accepts a general request-response protocol, so in this case the protocol passed was the one shown in **2.9**.

Accelerators that leverage the AOPB are referred to as *Smart Peripheral Controllers* (SPCs). The AOPB is completely transparent to both the X-HEEP core and the SPCs, as both simply access peripherals by indexing their memory-mapped addresses.

To facilitate the development of DMA-based SPCs, the DMA channel exposes two key signals that can be utilized by the SPCs. The first signal, *dma_done*, indicates when a transaction has been completed. The second, an input signal called *dma_stop*, resets the channel's finite state machines (FSMs). This feature is particularly useful in scenarios where fewer data need to be written than originally anticipated

For instance, one application of the AOPB, which will be developed at the EPFL's Embedded Systems Laboratory in the near future, is a *Level-Crossing Subsampler* (LCS). In brief, this system will read data from an ADC and instruct the DMA to store only specific values—those samples that have crossed a predefined threshold. This mechanism avoids storing insignificant data, such as low-amplitude noise, which would ideally remain within the LCS threshold and thus go undetected.

By skipping unnecessary memory writes, this system significantly reduces memory accesses, a key advantage for low-power platforms like X-HEEP.

However, in such a scenario, the DMA channel configured to copy N samples might ultimately need to copy only $N-m$ samples, as the LCS effectively subsamples the data. When this occurs, the *dma_stop* signal becomes essential, allowing the external accelerator to halt the DMA transaction as needed.

2.5 Developing an Im2col SPC to Optimize On-Edge CNNs

2.5.1 *NCHW* vs. *NHWC*: Choosing the Optimal Format

As discussed in Section **1.1.3**, two prevalent formats for representing tensors are *NHWC* and *NCHW*. The cited section highlighted that the *NCHW* format is more advantageous for low-power, RISC-V microcontrollers such as X-HEEP.

Building upon this insight, an application was developed during this thesis to convert *NHWC* tensors to *NCHW* format, leveraging the 2D DMA capabilities.

During the development and testing of the application, it became apparent that the conversion between *NHWC* and *NCHW* formats can be performed as a simple matrix transposition in both directions.

To better visualize this, consider flattening the HxW matrices that compose the channels in the *NCHW* representation. In this flattened view, each *NCHW* row, representing a channel, corresponds to a column in the *NHWC* format. This realization is significant because it means that the conversion from one format to the other can be efficiently executed using the new DMA system's transposition feature, with minimal overhead involved.

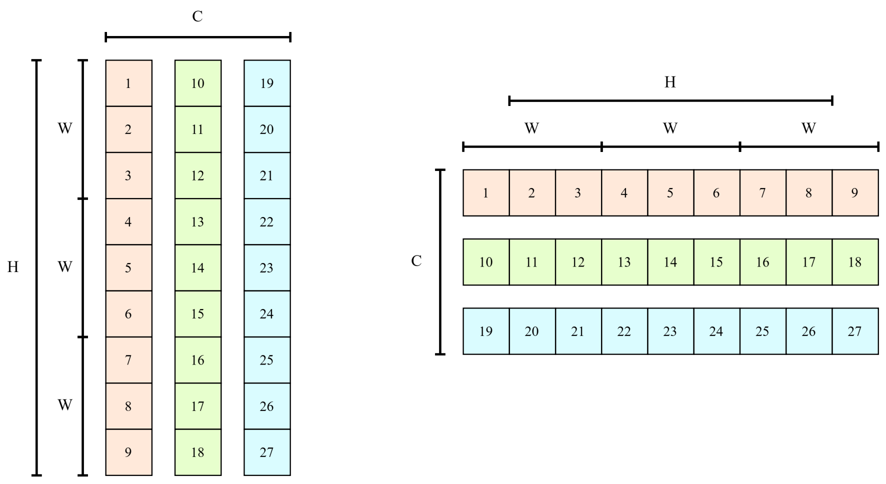


Figure 2.20: This is a representation of a 3x3x3 tensor, flattened in *NCHW* format on the right and *NHWC* format on the left. The dimensions are highlighted, and the channels are distinguished by different colors. This visualization demonstrates that the two formats can be easily interchanged through a simple transposition.

This efficient bidirectional conversion is great news, as it simplifies the interchangeability between the two formats without incurring significant computational costs. A single configuration of the DMA is sufficient to handle the conversion of an entire tensor, making the process both swift and resource-efficient.

Figures **2.21** and **2.22** provide a graphical illustration of the access patterns for both *NCHW* and *NHWC* formats, showing how both the input tensors and the algorithm change. It highlights how both the input tensors and the algorithm change.

The *NCHW* format is *linear*, relying on repeated regular matrix copies, which are efficiently mapped to the new DMA system.

In contrast, the *NHWC im2col* operation is less straightforward. Even in this simple example, the DMA must operate across three 'dimensions': while elements 1 and 2 are contiguous within the same column, copying elements 3 and 4 requires moving to the next row block. The two dimensions of the new DMA are quickly exhausted, as the first is used to move from element 1 to 2 along the same column, and the second to shift from element 2 to 3. A *third dimension*, along with functionality akin to transposition, would be needed to move from element 4 to 5, which would require additional hardware resources.

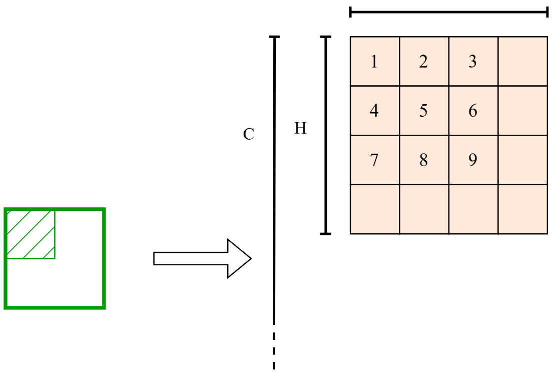


Figure 2.21: Initial steps of the im2col reshaping transformation of an *NCHW* tensor.

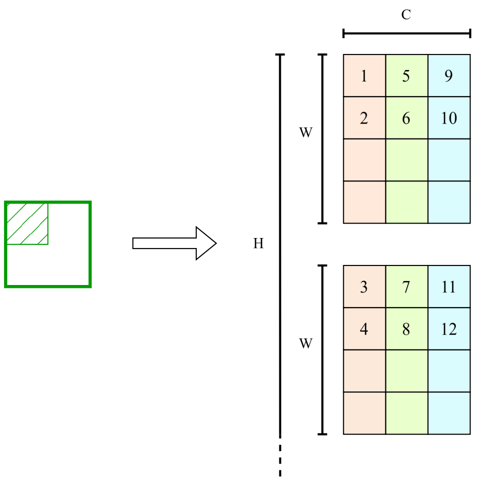


Figure 2.22: Initial steps of the im2col reshaping transformation of an *NHWC* tensor.

This example clearly shows that there is no universal algorithm to perform *im2col* on both *NHWC* and *NCHW* formats. Implementing both would require separate logic in an accelerator, resulting in double the area overhead, with only half of it utilized at any given time. This consideration, along with the low cost of format conversion, supports the decision to accelerate a single tensor format.

Among the two, *NCHW* emerges as the optimal choice, as it avoids the need for additional hardware resources.

2.5.2 DMA-based vs Specialized Im2col Accelerators

Due to the importance of the *im2col* transformation, several accelerators have been proposed in the research community, such as SPOT [**15**]. SPOT is a hardware accelerator designed for sparse convolutional neural networks (CNNs). It integrates a novel *im2col* unit with a systolic array-based general matrix multiplication (GEMM) unit, streamlining the *im2col* transformation to maximize data reuse and minimize redundant memory accesses.

SPOT's architecture supports dynamic reconfigurability, allowing it to adapt to various CNN layers while exploiting sparsity in both input feature maps and weights.

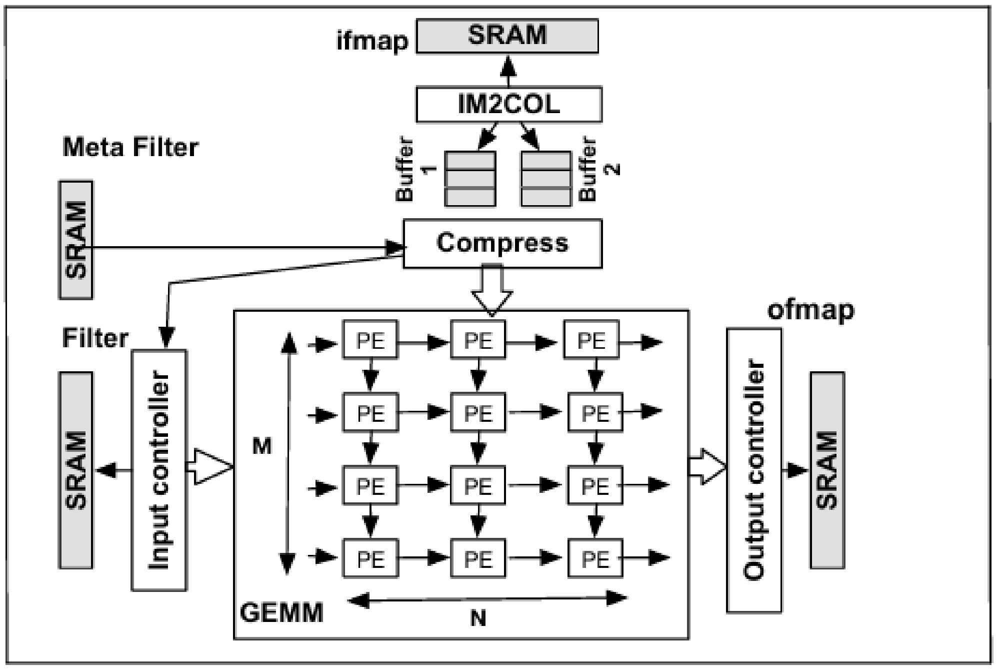


Figure 2.23: SPOT architecture overview.

In contrast, a DMA-based accelerator could leverage the existing DMA peripheral in a system, offering a simpler and more efficient approach. While SPOT performs the function of a DMA itself, this increases complexity and area overhead. Without needing specific data, it is evident that a DMA-based accelerator—by reusing a peripheral already present in the system—would require significantly less hardware, making it ideal for ultra-small, low-power edge-computing platforms where minimizing area is essential.

SPOT's strategy of reducing memory accesses through buffering and reuse is innovative, but it results in a much larger hardware footprint, which might not be practical for constrained edge devices.

Furthermore, while SPOT benefits from *sparse matrix representations*, this technique is less relevant for simpler CNNs, where performance bottlenecks are typically found in the input rather than the filters. In such cases, sparse filter representation would have little effect on overall performance.

2.5.3 Structural Description

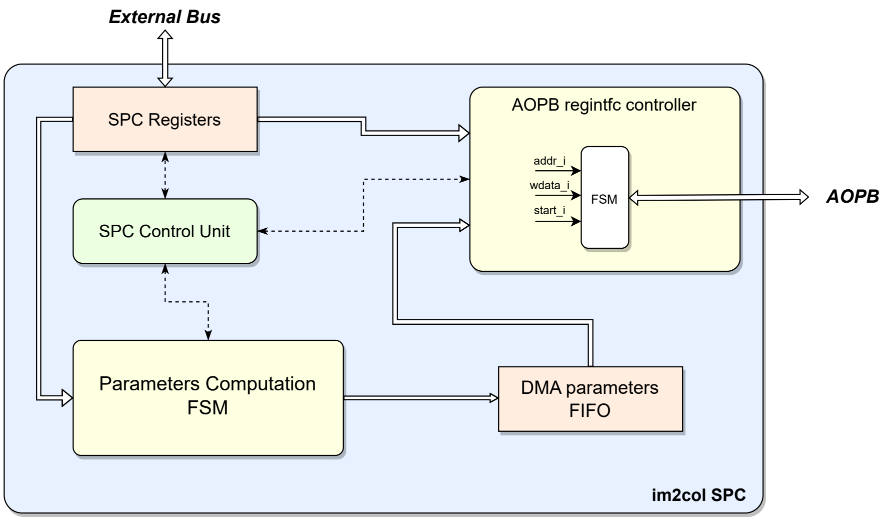


Figure 2.24: Structural overview of the *im2col* Smart Peripheral Controller.

This thesis proposes an implementation of an *im2col Smart Peripheral Controller* (SPC), which takes advantage of the *AOPB* introduced in the previous chapter to leverage the new DMA system for reshaping *NCHW* tensors, preparing them for GEMM accelerators.

With respect to specialized accelerators as *SPOT*, this approach is more suitable for low-power, small edge-computing platforms, as it reuses the system's existing DMA with minimal overhead, while remaining flexible enough to integrate with any type of GEMM accelerator.

The SPC is composed of three main blocks:

- **Parameter FSM**: Responsible for calculating the necessary parameters required to configure each DMA run.

- **Register Interface Controller**: This block interfaces with the AOPB, programming one DMA channel register at a time.
- **Top Module**: This unit instantiates the Parameter FSM and the Register Interface Controller. It includes an FSM that manages the DMA channel configuration by calculating the correct register address, selecting the appropriate data to be written, and initiating the Register Interface Controller.

To buffer the parameters and enhance throughput during the DMA configuration time, a FIFO is placed between the two FSMs. Similar to the architecture of a DMA channel, the FIFO decouples the different stages, simplifying the design and improving robustness.

Each FIFO entry holds all the parameters needed to configure the DMA that change with each iteration, while other parameters, such as the datatype, remain static and are directly taken from the `im2col` SPC configuration register. The structure of the FIFO's datatype is declared in a special package file, as reported in code **2.10**.

Listing 2.10: DMA interface datatype.

```
package dma_if_pkg;

  typedef struct packed {
    logic [31:0] input_ptr;
    logic [31:0] output_ptr;
    logic [22:0] in_inc_d2;
    logic [7:0]  n_zeros_top;
    logic [7:0]  n_zeros_bottom;
    logic [7:0]  n_zeros_left;
    logic [7:0]  n_zeros_right;
    logic [15:0] size_du_d1;
    logic [15:0] size_du_d2;
  } dma_if_t;

endpackage
```

Mapping the im2col algorithm to the new DMA system

The algorithm that the SPC implements in hardware was first developed in software. Specifically, a C code was written to map the XuanTie `NCHW im2col` algorithm [**9**] onto the advanced DMA, taking advantage of its 2D, stride, and zero-padding capabilities to remove two loops. The SPC then implements this algorithm in hardware.

The pseudocode for the original XuanTie im2col algorithm for NCHW tensors is reported in code **2.11**.

Listing 2.11: XuanTie C code implementation of the im2col algorithm for NCHW tensors.

```
height_col = (height + pad_top + pad_down - ksize_h)/stride_h + 1;
width_col = (width + pad_left + pad_right - ksize_w)/stride_w + 1;
channel_col = channel * ksize_h * ksize_w;

for c = 0 to channel_col - 1 do
    w_offset = c % ksize_w;
    h_offset = (c / ksize_w) % ksize_h;
    c_im = c / (ksize_h * ksize_w);
    for b = 0 to batch - 1 do
        for h = 0 to height_col - 1 do
            for w = 0 to width_col - 1 do
                im_row = h_offset + h * stride_h - pad_top;
                im_col = w_offset + w * stride_w - pad_left;
                col_index = ((c * batch + b) * height_col + h) * width_col + w;
                if im_row < 0 or im_col < 0 or im_row >= height or im_col >= width then
                    output_data[col_index] = 0.0;
                else
                    output_data[col_index] = input_data[get_index(b, c_im, im_row, im_col)];
                end if
            end for
        end for
    end for
end for
```

The variables `w_offset` and `h_offset` represent the current horizontal and vertical positions within the convolution filter window. These offsets define the specific element in the filter being considered by the algorithm at each step.

To explain the computation of the number of zeros to pad on the left side (`n_zeros_left`), let's analyse the three possible cases, based on the current offset and the left padding (`LEFT_PAD`). The code snippet is reported in code **2.12**.

The meaning of the three cases of the previous snipped are:

- **No Padding Required:** If the filter's horizontal offset (`w_offset`) is greater than or equal to the left padding (`LEFT_PAD`), no additional zeros are needed on the left side.

Listing 2.12: Zeros on the left computation.

```c
if (w_offset >= LEFT_PAD) {
    n_zeros_left = 0;
}
else if ((LEFT_PAD - w_offset) % STRIDE_D1 == 0) {
    n_zeros_left = (LEFT_PAD - w_offset) / STRIDE_D1;
}
else {
    n_zeros_left = (LEFT_PAD - w_offset) / STRIDE_D1 + 1;
}
```

- **Perfect Stride Alignment:** When the remaining padding aligns exactly with the stride (`STRIDE_D1`), the number of zeros is calculated as the integer division of the padding difference by the stride. For example, consider a 3-element wide filter applied to an input with a left padding of 3 and a stride of 2, where `w_offset` is 1. Here, the filter's second element encounters the padding only once, resulting in `n_zeros_left = (3 - 1) / 2 = 1`.
- **Partial Stride Coverage:** In cases of misalignment, an additional zero is needed to account for the partial stride, ensuring complete coverage. For instance, with the same 3-element filter and stride 2, but with `w_offset = 0`, the filter's first element encounters the padding twice. In this case, `n_zeros_left` is calculated as `(3 - 0) / 2 + 1 = 2`.

The computation for the number of zeros on the top (`n_zeros_top`) follows a similar logic, adjusting for the vertical offset and top padding (`TOP_PAD`), and its reported in code **2.13**.

Listing 2.13: Zeros on top computation.

```c
if (h_offset >= TOP_PAD) {
    n_zeros_top = 0;
}
else if ((TOP_PAD - h_offset) % STRIDE_D2 == 0) {
    n_zeros_top = (TOP_PAD - h_offset) / STRIDE_D2;
}
else {
    n_zeros_top = (TOP_PAD - h_offset) / STRIDE_D2 + 1;
}
```

For the right and bottom padding, two additional parameters are required:

`fw_minus_w_offset` and `fh_minus_h_offset`, which represent the mirrored horizontal and vertical offsets, calculated as shown in code **2.14**.

Listing 2.14: Mirrored filter offset computation.

```
fw_minus_w_offset = FW - 1 - w_offset;
fh_minus_h_offset = FH - 1 - h_offset;
```

Additionally, the *adapted padding regions* (`ADPT_PAD_RIGHT` and `ADPT_PAD_BOTTOM`) are introduced. These parameters adjust the padding to handle cases where the filter does not perfectly align with the input tensor due to the stride and filter size. To explain their meaning and computation, let's analyse the right padding case, whose computation is reported in **2.15**.

Listing 2.15: Adapted padding region computation.

```
ADPT_PAD_RIGHT = (STRIDE_D1 * (N_PATCHES_W - 1)) + FW - (LEFT_PAD
    + IW);
```

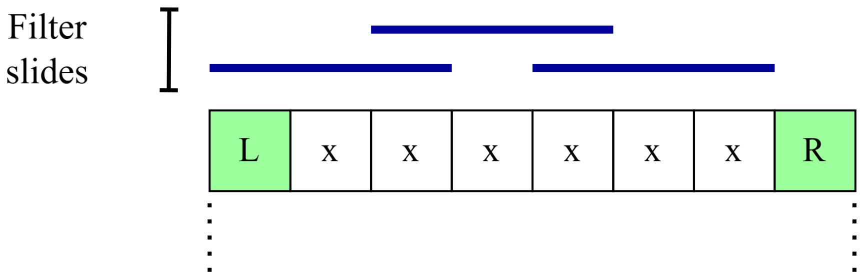


Figure 2.25: Example of a non-aligned filter slide.

Let's take a practical example: a 6-wide input with a 3-wide filter, 1 padding on both the left and right, and a stride of 2. Figure **2.25** shows a single row of a channel from such a tensor.

When sliding the filter over the input, it becomes evident that the filter cannot fully cover the entire row. Thus, even with non-zero right padding, there will be no padding in the output `im2col` transformation. The adapted padding region reflects this exact scenario: in this case, it computes to $(2 \times (3-1)+3-1-6)=0$.

Now, let's analyze the computation of the number of zeros on the right (`n_zeros_right`) using the adapted padding region parameter, reported in code **2.16**.

Listing 2.16: Zeros on the right computation.

```
if (fw_minus_w_offset >= RIGHT_PAD || ADPT_PAD_RIGHT == 0) {
    n_zeros_right = 0;
}
else if ((ADPT_PAD_RIGHT - fw_minus_w_offset) % STRIDE_D1 == 0) {
    n_zeros_right = (ADPT_PAD_RIGHT - fw_minus_w_offset) /
   STRIDE_D1;
}
else {
    n_zeros_right = (ADPT_PAD_RIGHT - fw_minus_w_offset) /
   STRIDE_D1 + 1;
}
```

- **No Padding Needed:** If the mirrored offset exceeds the right padding or the adapted padding region is zero, no zeros are needed.
- **Perfect Stride Alignment:** If the difference between the adapted padding region and the mirrored offset is divisible by the stride, the number of zeros is computed as $(ADPT\_PAD\_RIGHT - fw\_minus\_w\_offset)/STRIDE\_D1$.
- **Partial Stride Coverage:** If misalignment occurs, we add one with respect to the previous case to cover the incomplete stride.

This calculation ensures accurate handling of the right padding, especially when the filter's slide does not perfectly cover the input tensor due to stride misalignment. The adapted padding region adjusts the necessary padding to guarantee complete coverage without overlapping.

The computation for the number of zeros on the bottom (`n_zeros_bottom`) is analogous to the right padding, adjusting for the vertical dimension and using `fh_minus_h_offset` and `ADPT_PAD_BOTTOM`, and reported in code **2.17**.

Additional parameters are needed for the configuration of the DMA, including the transaction size in both dimensions. This is computed using the formulas reported in code **2.18**.

The stride for the first dimension is obtained via a constant parameter, which remains unchanged throughout the algorithm. However, the stride for the *second dimension* varies with each run, as it depends on the transaction size, which in turn is influenced by the number of zero-padding elements.

Additionally, the *source pointer* for each run is calculated by adding an offset to the pointer of the first element of the tensor. This offset is derived using the function reported in code **2.19**.

The function is called every iteration as shown in **2.20**.

Listing 2.17: Zeros on the bottom computation

```
if (fh_minus_h_offset >= BOTTOM_PAD || ADPT_PAD_BOTTOM == 0)
{
    n_zeros_bottom = 0;
}
else if ( (ADPT_PAD_BOTTOM - (fh_minus_h_offset)) % STRIDE_D2 ==
   0)
{
    n_zeros_bottom = (ADPT_PAD_BOTTOM - (fh_minus_h_offset)) /
   STRIDE_D2;
}
else
{
    n_zeros_bottom = (ADPT_PAD_BOTTOM - (fh_minus_h_offset)) /
   STRIDE_D2 + 1;
}
```

Listing 2.18: Computation of the transaction size

```
size_transfer = N_PATCHES_W - n_zeros_left - n_zeros_right;
size_transfer_d2 = N_PATCHES_H - n_zeros_top - n_zeros_bottom;
```

After the transaction is configured and launched, the output pointer is incremented by the size of the output row in the output matrix.

Furthermore, the `w_offset`, `h_offset`, and `im_c` (the channel index) parameters are updated using an optimized mechanism that leverages modulo operators and counters. This optimization improves the compatibility with hardware mapping and reduces CPU load.
The horizontal offset `w_offset` value is incremented by one after each batch. If `w_offset` reaches `FW - 1`, it is reset to zero. This method eliminates the need for modulo operations by utilizing simple increment and reset logic.

Similarly, the vertical offset `h_offset` is updated using an auxiliary counter, `h_offset_counter`. After each batch, `h_offset_counter` is incremented, and when it reaches `FW - 1`, it is reset to zero and `h_offset` is incremented. If `h_offset` reaches `FH - 1`, it is also reset to zero. This approach replaces the division and modulo operations with simpler increment logic.

Lastly, for the image channel index `im_c`, calculated as `im_c` $= \left\lfloor \frac{c}{FW \times FH} \right\rfloor$, another counter, `im_c_counter`, is used. This counter is incremented after each

Listing 2.19: Function to get the input pointer offset.

```c
int get_index(int dim1, int dim2, int dim3, int index0, int index1
   , int index2, int index3) {
    return ((index0 * dim1 + index1) * dim2 + index2) * dim3 +
   index3;
}
```

Listing 2.20: Call to get the input index.

```c
int index = get_index(CH, IH, IW, b, im_c, im_row + n_zeros_top*
   STRIDE_D2, im_col + n_zeros_left*STRIDE_D1);
```

batch, and when it reaches `FW` × `FH - 1`, it is reset to zero and `im_c` is incremented. This optimization reduces computational overhead by avoiding direct division operations.

By replacing expensive arithmetic calculations with efficient increment and reset mechanisms, these optimizations significantly enhance the performance of the `im2col` mapping process, leading to faster computation times and reduced resource utilization.

2.5.4 Parameter FSM

The Parameter FSM is designed to implement the algorithm described in the previous section in hardware, with a few optimizations.

Firstly, the algorithm has been simplified by imposing a constraint on the D1 and D2 strides, limiting them to multiples of 2. This reduces the multiplications and divisions required for computing the DMA parameters involving strides to simple shift operations. The modulo operation, used for calculating zero-padding sizes, has also been optimized thanks to this additional constraint. To check if a number is divisible by a multiple of 2, the formula shown in **2.21** is used.

Secondly, the computation of the input offset and zero-padding sizes has been divided into multiple stages using simple *pipeline registers*. This optimization was introduced to improve the critical path of the FSM, as meeting timing constraints in FPGA synthesis proved challenging.

Listing 2.21: Optimized divisibility condition computation.

```
divisible_condition = number & ((1 << strides) - 1)
```

2.5.5 Register Interface Controller

This unit communicates with the *AOPB* to configure the DMA system, accessing the parameters stored in the FIFO by the Parameter FSM. It is a simple FSM, composed of four states:

- **IDLE**: The default state. The FSM enters this state after a reset or when a transaction is completed. When the *start_i* signal is asserted, the FSM initiates the transaction process.
- **SENDING**: In this state, the FSM sets the register interface protocol signals, including *valid*, *write*, *addr*, and *wdata*.
- **WAITING_READY**: The FSM waits for the *ready* signal from the AOPB in this state before moving to the next one.
- **DONE**: The transaction is complete, and the FSM returns to the IDLE state.

This unit has been designed with a general structure to support future developments of *Smart Peripheral Controllers*, such as the im2col SPC described in this section.

2.6 Developing VerifHEEP, a Software-Based Self-Test Functional Verification Library for X-HEEP

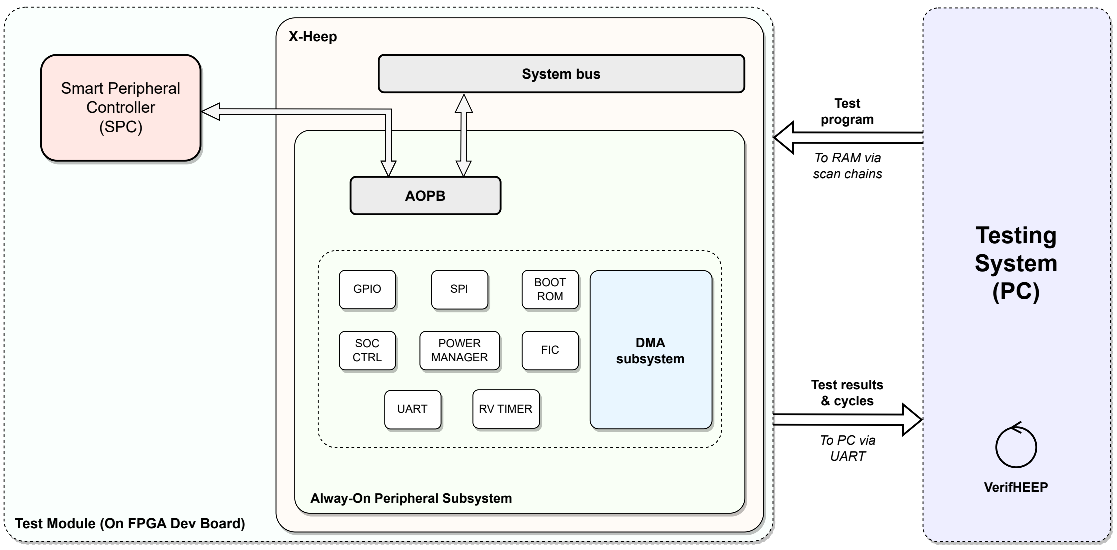


Figure 2.26: Overview of a SBST Verification Set-Up.

2.6.1 Advantages of Software-Based Self-Test Techniques

Software-Based Self-Test (SBST) is a technique traditionally used to *test* the functionality of digital systems, particularly processors and embedded cores, by running dedicated software routines on the target hardware.

Originally, SBST was introduced to perform *self-test operations* on deeply embedded systems where external testing equipment, such as *Automatic Test Equipment* (ATE), may be impractical or costly. In this context, SBST allows the hardware to test itself by executing pre-defined software patterns, checking for defects or malfunctions in real-time.

This approach is especially valuable in *post-manufacturing* phases, where the focus is on identifying operational faults and ensuring the system functions as intended.

In contrast, *verification techniques* are typically employed during the design phase to ensure that a system or component meets its functional and design specifications before hardware is physically built.

Formal verification methods use mathematical proofs to ensure that the system behaves correctly under all possible inputs, providing guarantees of correctness.

Techniques such as model checking and theorem proving are common in this domain.

However, formal verification can be *highly complex* and time-consuming, especially for large or intricate systems, often requiring specialized tools and expertise. Scaling formal methods to handle the full complexity of modern hardware designs can be difficult, which sometimes limits their application to specific system components

Functional verification, on the other hand, involves testing the system's response to specific, often practical, scenarios to confirm that the design meets its requirements, typically without exhaustive guarantees. Simulation, emulation, and prototyping are widely used approaches for functional verification.

While traditionally a *testing* technique, *SBST* can also be adapted to serve as a *functional verification* tool, especially when implemented on an *FPGA* or other reconfigurable hardware platforms during the design phase.

By automating the execution of SBST routines across thousands of configurations, it is possible to empirically verify the correct functionality of a unit in a wide range of scenarios, akin to functional verification. This process validates that the system behaves according to its specifications, even though it does not provide formal guarantees.

Thus, SBST can be extended beyond post-silicon testing to functionally verify the design of a unit in a pre-silicon or prototyping environment.

2.6.2 Integrating the new designs into a 16nm silicon implementation of HEEP

HEEPatia is the latest silicon implementation of X-HEEP, developed using TSMC's 16nm technology and designed for low-power edge-computing applications. At the time of this thesis, the project is in its final design stages and is expected to be produced by the end of 2024 by a team from the Embedded Systems Laboratory at EPFL.

All the new designs introduced in this thesis, including the DMA system, the im2col SPC, and the AOPB, will be integrated into the *HEEPatia* chip. This represents a deeply significant achievement for this thesis, as the opportunity to validate these designs on a high-performance technology is both a privilege and a testament to the quality of this work.

Additionally, this integration allows for their enhancement within powerful and advanced units such as the *CARUS* in-memory computing IPs and *CGRAs*.

As a result of this integration, it became essential to develop a methodology for *verifying* and *testing* the new designs. Due to the necessity of developing a solid and straightforward solution, *Software-Based Self-Test* (SBST) *functional verification* was deemed the most viable solution.

Moreover, given the need to test a large number of configurations in a short time, testing on a simulated model proved inadequate. Thus, the target platform for the verification system was necessarily an *FPGA* development board, capable of supporting the test harness and programmable via a PC.

The *PYNQ-Z2* FPGA development board by *AMD*, already supported on X-HEEP and available in the Embedded Systems Laboratory, was chosen as the primary target for the verification script.

In alignment with X-HEEP's open-source nature, it was deemed beneficial to not only create a verification system for the designs introduced in this thesis but also to develop a system adaptable to any X-HEEP-based application. Thus, *VerifHEEP* was born, a Python library offering a range of methods that enable users to rapidly construct a *complete verification environment*. *VerifHEEP* covers every aspect, from model compilation and synthesis of the model to random data generation and golden results computation, including completion time estimation.

To achieve this, VerifHEEP takes advantage of the flows already developed for X-HEEP's model building and synthesis, automating them, and integrating with functionalities to generate datasets, automate the interface with the board itself, like the GDB debugger to load executables and start the execution and serial communication threads to manage the reception of results.

2.6.3 VerifHEEP Library Structure

Model Compilation and Synthesis

The VerifHeep *class* provides robust methods for compiling and synthesizing X-HEEP models, leveraging existing flows developed for the X-HEEP project. The `compileModel` method generates the X-HEEP configuration by specifying parameters such as memory banks, CPU type, and bus configuration. This setup ensures that the verification environment is prepared with the correct hardware configuration to build X-HEEP.

The `buildModel` method compiles the microcontroller for various targets, including *Verilator* and *QuestaSim*, based on the specified optimization settings and hardware requirements. This flexibility allows users to select the most suitable compilation toolchain and target platform for their specific verification needs, making the process both efficient and *adaptable*.

Input Data and Golden Result Generation

Another significant part of VerifHeep is its capability to *generate input datasets* and *compute golden outputs* for verification purposes.

The genInputDataset method creates random input datasets, which can be customized based on several parameters, such as datatype, size, and value range. These datasets are used as stimuli for data-processing applications, helping to validate the proper functioning of the system under test.

The generation of random input data can be tailored to meet specific requirements. Users can specify the datatype (e.g., uint32_t, int8_t), allowing the data to be suited for different types of processing units and test scenarios.

Additionally, users can control the size of the dataset, as well as the minimum and maximum range for the values, ensuring the generated data is appropriate for the intended verification goals.

The genInputDataset method also supports the inclusion of *additional parameters* to further customize the dataset. These parameters, such as configuration constants or metadata about the dataset structure, are defined and included in the generated header file. This feature enables the provision of metadata that may be useful for interpreting or processing the dataset, like dimensions of a tensor in terms of number and sizes of channels, as reported in code **2.23**.

Listing 2.22: Input data C file for the im2col application developed to test the im2col SPC and compare its performance to the CPU.

```c
#include "im2col_input.h"

const uint32_t input_image_nchw[25] = {
 45280, 44040, 39555, 48322, 43761,
 46986, 61580, 31407, 53015, 50099,
 48311, 52851, 12647, 34880, 26310,
 1708, 38684, 39145, 34646, 20833,
 38991, 55892, 20081, 22175, 48554
};
```

Additionally, the genGoldenResult method is used to compute the expected output (golden result) using a provided reference function. This ensures that the application's results match the expected behavior, which is fundamental for verifying the correctness of computations performed by accelerators and processing units.

The golden result is generated based on the input dataset and can include any relevant parameters returned by the reference function. These parameters are then written into the header file alongside the golden output, providing all the necessary information for comparison during verification.

Listing 2.23: Input data header file for the im2col application developed to test the im2col SPC and compare its performance to the CPU. It includes the parameters of the input tensor and the padding to apply to it.

```c
#include <stdint.h>

#define IH 5
#define IW 5
#define CH 1
#define BATCH 1
#define FH 3
#define FW 3
#define TOP_PAD 1
#define BOTTOM_PAD 1
#define LEFT_PAD 1
#define RIGHT_PAD 1
#define STRIDE_D1 1
#define STRIDE_D2 1

extern const uint32_t input_image_nchw[25];

#endif // INPUT_IMAGE_NCHW_H
```

Board Interface and Debugging

The library also includes methods to establish communication with the target board and facilitate debugging.

The `serialBegin` method initializes a serial communication channel between the host and the target, enabling the transfer of test data and retrieval of results.

Debugging methods such as `setUpDeb` and `stopDeb` help configure GDB sessions and remotely control the target, allowing users to load and debug the software running on the PYNQ-Z2 FPGA board.

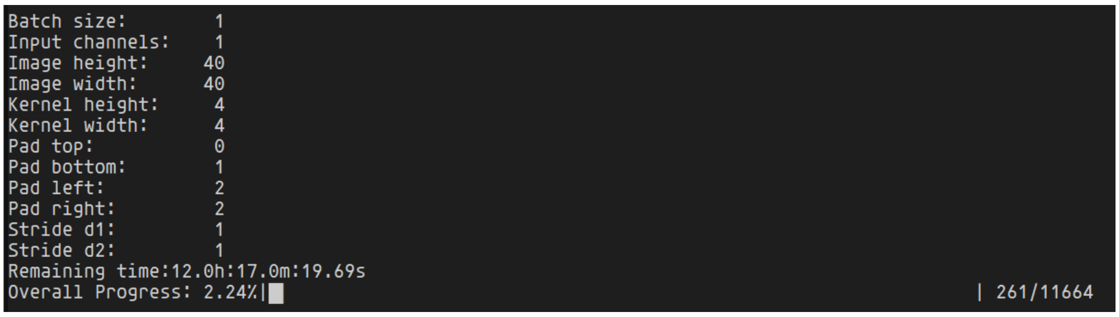
The `launchTest` method integrates these debugging capabilities to *execute test applications*, gather the serial output, and parse the results to evaluate verification outcomes.

Execution Time Estimation

VerifHeep includes a set of functions dedicated to performance estimation, which is particularly useful during verification involving multiple iterations. The `chronoStart`, `chronoStop`, and `chronoExecutionEst` methods assist in determining the average execution time of test iterations and estimating the remaining time for the entire test.

By providing accurate timing information, these methods help developers optimize and manage test durations effectively.

2.6.4 Developing a Verification Environment for the Im2col SPC using VerifHEEP



```
Batch size:        1
Input channels:    1
Image height:     40
Image width:      40
Kernel height:     4
Kernel width:      4
Pad top:           0
Pad bottom:        1
Pad left:          2
Pad right:         2
Stride d1:         1
Stride d2:         1
Remaining time:12.0h:17.0m:19.69s
Overall Progress: 2.24%|█                                                        | 261/11664
```

Figure 2.27: Screenshot of the Im2col SPC verification environment interface, built using VerifHEEP.

This use case demonstrates the power of the VerifHeep tool in developing a verification environment for the Im2col SPC functionality. It involves running *thousands of iterations*, each with different parameter configurations.

This approach allows for an extensive verification of the Im2col function under various conditions. Each iteration of the loop modifies parameters such as the input tensor dimensions, padding, kernel size, and strides to comprehensively validate the performance and correctness of the Im2col SPC across different scenarios.

The test application developed for this environment assesses three different implementations of the Im2col function. First, it performs the Im2col operation using only the CPU, executing the C code version implemented by XuanTie, as shown in **2.11**. Next, it runs the same algorithm using the new DMA engine, as described in section **2.5.3**. Finally, the test uses the specialized Im2col SPC hardware described in chapter **4.2**.

In each iteration, the process begins by generating an input dataset using the `genInputDataset` function. This dataset is tailored to the specific parameters for that iteration, ensuring that each configuration is tested thoroughly.

The golden result is then computed using the `genGoldenResult` function, which takes the generated input dataset and passes it through a reference `im2col_function` implemented in PyTorch. This function performs the Im2col transformation, providing the expected output for comparison during verification.

After generating the dataset and golden result, the environment connects to the PYNQ-Z2 board using GDB, and the test is executed via the `launchTest` function. The execution involves running the Im2col function on the hardware, with the results collected through serial communication.

The serial output is then parsed and categorized into different lists based on the type of test outcome, such as CPU, DMA 2D, or SPC. Each iteration logs information about the cycles and parameter configurations, allowing for detailed performance analysis.

The graphical aspect of this verification environment is managed using the `curses` and `tqdm` libraries. The `curses` library provides an interactive console interface that displays the current parameters being tested, while `tqdm` offers a progress bar indicating the overall progress of the verification process.

This combination creates a user-friendly interface that helps users monitor the progress of thousands of iterations in real-time, providing clear visibility into the ongoing verification. Figure **2.27** presents a screenshot of the graphical interface used in the im2vol verification process.

Finally, the results from each iteration are stored in three separate lists based on the type of test performed, i.e. CPU, DMA 2D, or SPC. These categorized results are then written to a text file for further analysis. Specifically, this environment has been used to obtain the results discussed in the next chapter.

Chapter 3

Results

3.1 Analysis of Performance and Area Overhead of the new DMA System

3.1.1 Test Set-Up

In order to gain enough insights to evaluate the performance of the new DMA system, a tailored VerifHEEP-based environment has been developed that runs *example_dma_2d*, an application developed to test different features of the introduced DMA system.

Each of these tests measured the DMA performance by carrying out the same operation on the CPU. The CPU result was used both to verify the correctness of the DMA and to obtain a cycle count, enabling a performance comparison between the DMA and the CPU.

Specifically, the following tests were performed:

- **Testing copy and padding of an NxM matrix using HALs**: This test extracts a matrix of size NxM, applies optional padding, and copies it to a destination matrix using hardware abstraction layers (HALs). This allows for a detailed evaluation of the data movement and padding mechanisms handled by the DMA.

- **Testing copy and padding of an NxM matrix using direct register configuration**: This test bypasses HALs and instead directly utilizes DMA register operations for copying and padding an NxM matrix. This test focuses on maximizing performance while sacrificing certain safety checks, allowing for a maximum performance.

During each iteration of the test, the input datasets were generated using

VerifHEEP's methods, as described in **2.6.3**, by varying both the extracted matrix dimensions and the padding applied. This approach ensured that a diverse dataset was used for each iteration, enhancing the potential and robustness of the DMA verification.

Additionally, area and power results were obtained via a *16nm synthesis* on the *HEEPatia* project, which, as mentioned in section **2.6.2**, will include the new DMA system.

3.1.2 Performance Improvement with respect to CPU Routines

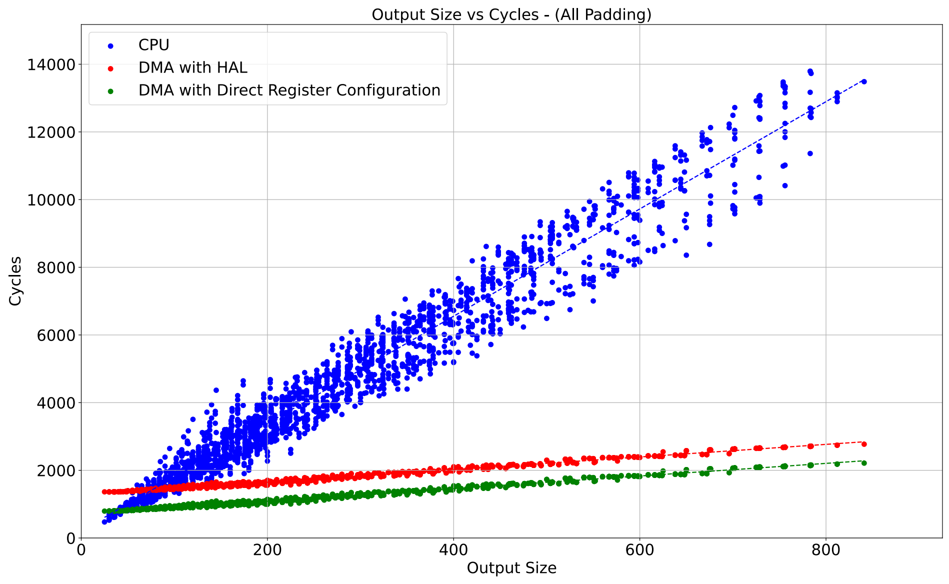


Figure 3.1: Performance comparison of a matrix copy operation between the CPU and DMA implementations. The y-axis shows the number of cycles required to perform the copy, while the x-axis represents the size of the output matrix, including the applied zero padding. The comparison highlights the efficiency of the DMA over the CPU, especially for larger matrix sizes.

The performance of the DMA was evaluated by comparing a matrix copy operation performed in three different ways: by the CPU, by the DMA using Hardware Abstraction Layers (HALs), and by the DMA using direct register writes.

Figure **3.1** presents the comprehensive results, with the CPU data in blue, DMA with HAL in red, and DMA with direct register configurations in green.

From the analysis of these results, four key observations can be made.

- The *DMA* clearly *outperforms* the CPU for larger transactions, showcasing its advantage in data transfer tasks.
- The use of HAL introduces an expected *overhead* due to validation and integrity checks. This overhead is constant rather than relative, meaning that for smaller transactions, the impact is more significant, whereas for larger transactions, it becomes less noticeable.
- For very small transactions, the *CPU* actually *outperforms* the DMA. This is because the constant configuration overhead of the DMA cannot be overcome for such small data sizes, making the CPU the faster option in these cases.
- There is significant *variation* in the CPU results, which is largely attributable to the effect of padding.

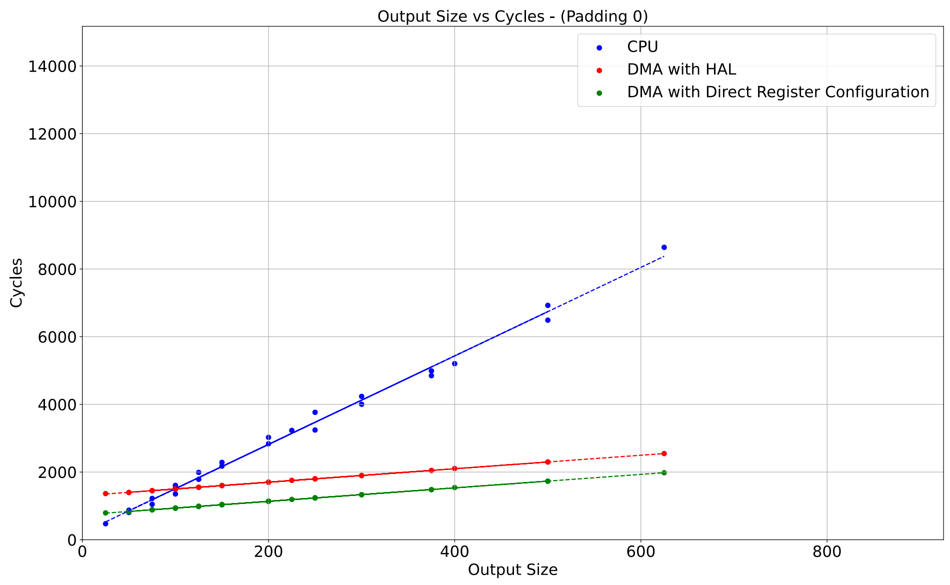


Figure 3.2: Performance comparison of a matrix copy operation without padding.

The padding phenomenon increases the loop size of the C code linearly, whereas for the DMA, it only involves configuring one of the registers. Whether or not padding is present, the configuration remains constant, resulting in no additional overhead for the DMA.

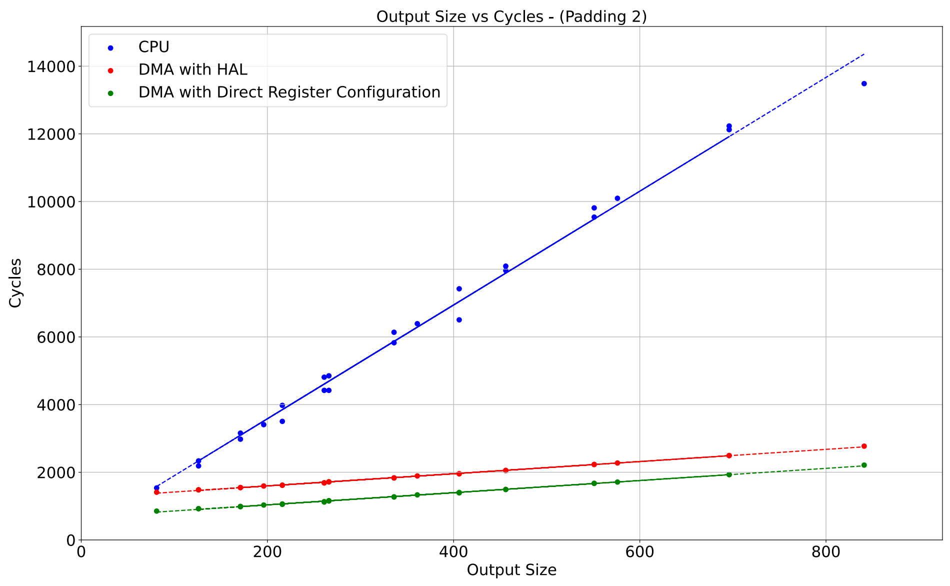


Figure 3.3: Performance comparison of a matrix copy operation with a 2-wide padding applied to every border of the input matrix.

Figures **3.2** and **3.3** illustrate the performance comparison when filtering tests without padding and with a 2-wide padding applied to every border of the input matrix, respectively.

Without padding, the DMA without HAL achieves a *performance improvement* of ***4.4 times*** compared to the CPU. With a 2-wide padding, the performance increase rises to ***6.1 times***, emphasizing the efficiency of the DMA in handling complex memory configurations.

3.1.3 Area Occupation Analysis

Let's start from analysing the area footprint of a single DMA channel, embedded in HEEPatia and synthetised on 16nm. As shown in Table **3.5**, the largest unit is the configuration registers.

This is understandable, since the DMA channel has a considerable number of parameters and configuration options, with a total of 26 parameters. Some of these are single-bit, such as the transaction and window IFRs, while others are 32-bit, like the source, destination, and address pointers.

The second largest contributor to the area footprint is the inter-stage *FIFOs*. Overall, the units with strong memory components comprise most of the area footprint, which is to be expected, as there are no large combinatory units present. The finite state machines (FSMs) constitute just 24% of the area of a single DMA channel.

To compare the area footprint of the new DMA channel to the previous DMA design, a synthesis using Xilinx Vivado was necessary. This is because the HEEPatia project originally began with the new DMA channel, making it technically impossible to perform a 16nm synthesis of both versions simultaneously.

Table **3.2** reports the differences between the two versions of the DMA, considering a single channel.

Table 3.1: Area footprint of a single DMA channel

Unit	Area Contribution (%)
Configuration Registers	40
Inter-stage FIFOs	36
FSMs	24

Table 3.2: Comparison between the new and old DMA channel

	New DMA	Old DMA
LUTs	1080	620
Registers	978	725

During these tests, the new DMA demonstrated performance improvements of approximately **53%** in the best-case scenario. Notably, the performance *gains scale linearly* with the matrix size, meaning that larger matrices yield even greater performance benefits.

Given these improvements, the increase in area footprint is well justified, as it leads to substantial gains in both functionality and efficiency.

3.2 Analysis of Performance, Power Consumption and Area Overhead of the Im2col SPC

The following sections present and analyze the results of a comprehensive verification test based on the environment described in Section **2.6.4**. Area and power figures were obtained through a 16nm technology synthesis conducted as part of the HEEPatia project.

3.2.1 Performance Improvement with respect to CPU-based Routines

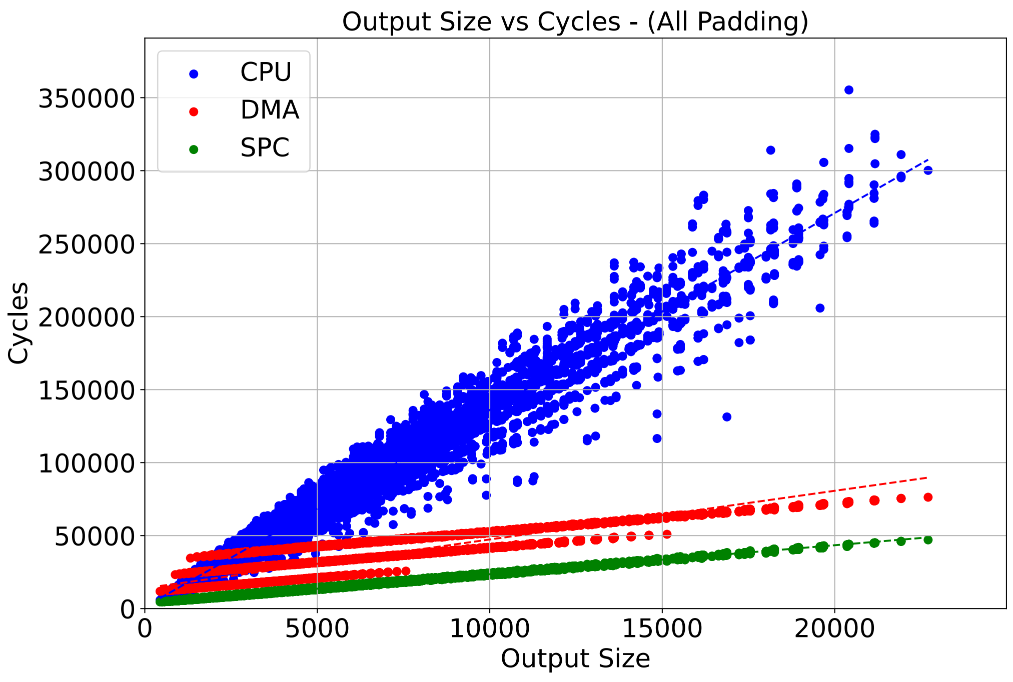


Figure 3.4: Performance comparison of an *im2col* reshaping transformation, performed using the XuanTie C implementation, its DMA mapped version and finally using the *im2col SPC*. The y-axis shows the number of cycles required to perform the operation, while the x-axis represents the size of the output matrix, including the applied zero padding. The comparison highlights the efficiency of the DMA over the CPU, especially for larger matrix sizes, and the additional advantage of the im2col SPC.

To evaluate the performance advantage of the im2col SPC, it was compared against the CPU-based routines, specifically the XuanTie C implementation and its DMA-mapped version, as explained in Section **2.6.4**. Figure **3.4** illustrates the results of the verification run, where thousands of tests were conducted, each with varying configurations and input data.

As expected, the results closely resemble the performance of a DMA channel during a matrix copy. This is because the im2col operation, at its core, involves a series of matrix copies, which becomes evident when the algorithm is mapped onto the new DMA.

The im2col SPC improves upon the DMA-programmed CPU routine by parallelizing the computation of parameters and programming the DMA channel with minimal overhead. As a result, it achieves a significant performance boost of up to ***60%***. However, the increase is not extraordinary, which aligns with expectations.

One key aspect of the im2col SPC, not immediately apparent from the performance analysis but clear from the power analysis discussed later, is that the CPU remains *completely inactive* while the im2col SPC is running.

As outlined in previous chapters, this allows the CPU to either process the transformed data or enter a low-power state, thereby enhancing energy efficiency. This is a substantial improvement, complementing the notable performance gains offered by the im2col SPC.

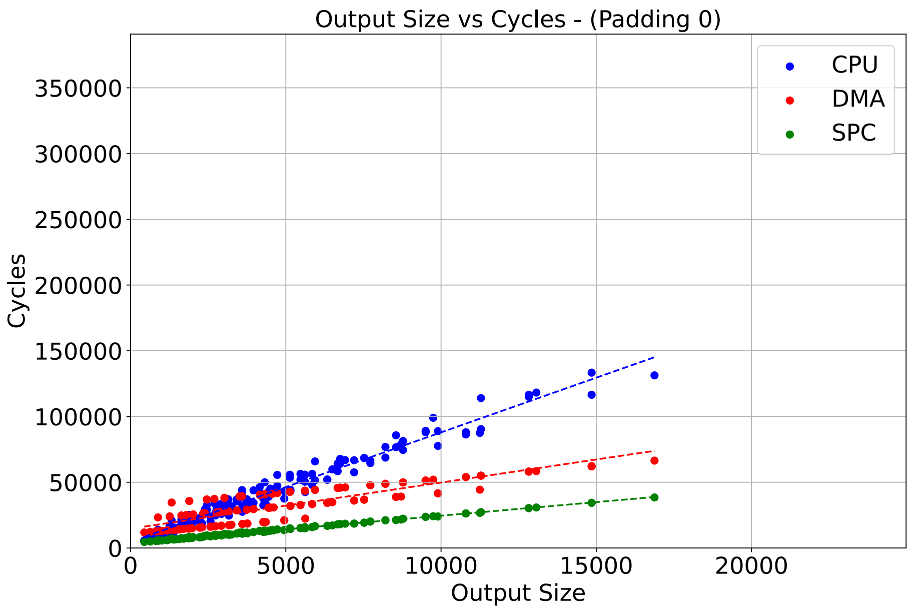


Figure 3.5: Performance comparison of an im2col reshaping operation without padding

Once again, there is a significant difference in performance depending on whether padding is involved, similar to the results observed in the DMA performance analysis. Figures **3.5** and **3.6** present the filtered results of the verification run, showing that when a padding of size 2 is applied to all sides, the im2col SPC performs **6.1x** better than the CPU. In contrast, without padding, the performance gain is **4.1x**. It is very important to emphasize once again that these *gains scale linearly* with the size of the test.

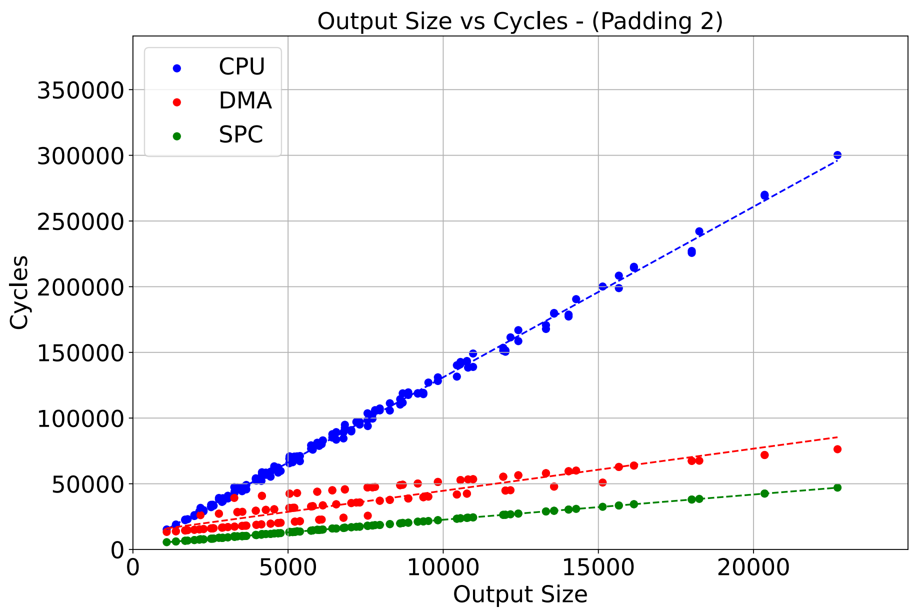


Figure 3.6: Performance comparison of an im2col reshaping operation with a 2-wide padding applied to every edge of the input tensor

3.2.2 Power Consumption Analysis

To perform a comprehensive analysis of the introduced im2col SPC accelerator, a power analysis was conducted using a *16nm technology*, embedding the SPC within the *HEEPatia* project. Three tests were carried out, comparing the SPC against the XuanTie C implementation and its DMA-mapped version, as detailed in Section **2.6.4**. The input tensor was generated using the *VerifHEEP* methods, with the following parameters:

- IH: 27
- IW: 27
- CH: 3
- BATCH: 1
- FH: 3
- FW: 3
- TOP_PAD: 2

- BOT_PAD: 2
- LEFT_PAD: 2
- RIGHT_PAD: 2
- STRIDE_D1: 1
- STRIDE_D2: 1

These parameters were set to produce an output matrix close to the size of a single RAM bank, i.e., 32kB. These tests were performed without low-power optimizations, meaning the power manager was not used to disable peripherals when not required.

The results of the power consumption simulations are shown in table **3.3** and figure **3.7**. Notably, the total power consumption among the CPU, DMA system and im2col SPC remained consistent across all tests, with a slight *3.88% reduction* when using the im2col SPC.

The DMA consumed slightly more power in the DMA-mapped test compared to the CPU test, due to cyclic register configurations. A similar observation can be made for the SPC, though it consumed slightly less power as it optimizes register configurations, avoiding redundant register rewrites, unlike the HAL used in the DMA-mapped test.

CPU power consumption decreased in the DMA-mapped test and further in the SPC test, as expected, since the CPU had to execute fewer instructions and perform less intensive operations.

As anticipated, the SPC power consumption remained stable in the first two cases but increased slightly in the final test due to register configuration and logic execution.

Test	DMA [W]	CPU [W]	im2col SPC [W]	Total [W]
XuanTie C	1.78e-04	3.81e-04	2.70e-04	8.29e-04
DMA-mapped	1.82e-04	3.55e-04	2.70e-04	8.07e-04
im2col SPC	1.80e-04	3.42e-04	2.76e-04	7.98e-04

Table 3.3: Power consumption of different units and their total contribution across different tests

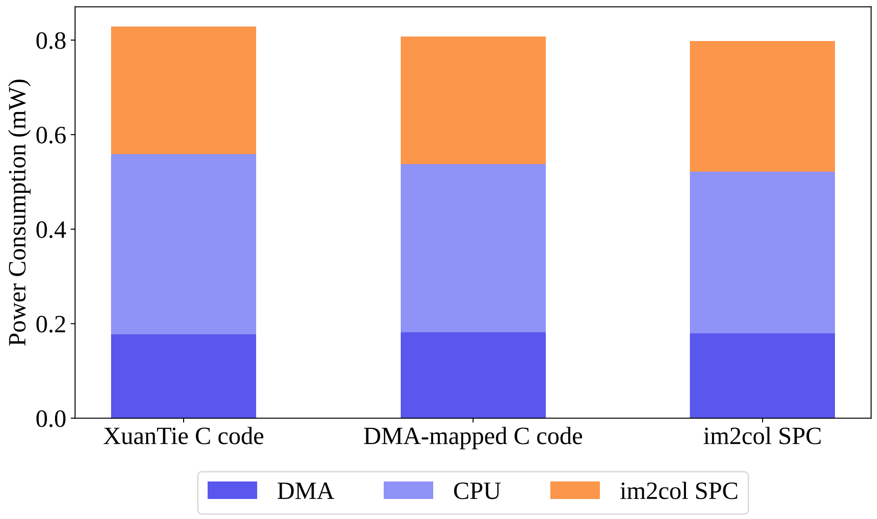


Figure 3.7: Graphical representation of the power consumption contributions for each type of test

This is a highly positive outcome, as it demonstrates that the performance gains from the im2col SPC and DMA-mapped approaches are achieved without increasing power consumption. Furthermore, these improvements translate directly into a *significant enhancement* in *energy efficiency*, which is a critical metric for low-power edge-computing applications.

Specifically, across the three tests, the energy efficiency improvements amount to **4.31x** for the DMA-mapped im2col implementation and **7.11x** for the im2col SPC, as illustrated in table **3.4**.

	XuanTie C	**DMA-mapped**	**im2col SPC**
Runtime [s]	3.22e-03	7.68e-4	4.71e-4
Total power consumption [mW]	0.829	0.807	0.789
Total energy consumption [J]	2.67e-6	6.20e-7	3.76e-7
Energy efficency gain	1x	4.31x	7.11x

Table 3.4: Estimation of energy efficency gains, considering the 100 MHz frequency used in the test

3.2.3 Area Occupation Analysis

Let's analyze the area footprint of the various components that make up the im2col SPC. The largest contributor to the area footprint is the DMA-interface FIFO, which is used to decouple the computation of DMA channel parameters from their

Table 3.5: Area footprint of the im2col SPC

Unit	**Area Contribution (%)**
Configuration Registers	13
DMA-interface FIFOs	36
Parameter FSM	24
AOPB Interface FSM	2

actual configuration.

This is expected, given the substantial number of parameters involved. The parameter FSM accounts for 24% of the total area footprint of the SPC, while the AOPB interface FSM contributes only 2%. Configuration registers represent 13% of the total area.

In relation to the total area of HEEPatia, the im2col SPC occupies only 0.43%. Given the significant performance and efficiency gains, alongside the minimal area footprint, the inclusion of this unit in the HEEPatia chip is well justified.

Chapter 4

Future Improvements and Conclusions

This section introduces concepts and strategies aimed at improving the designs and solutions presented in this thesis. While their implementation was beyond the scope and timeline of this work, their integration could offer significant enhancements to the challenges and critical issues related to the deployment of Edge AI models on X-HEEP.

4.1 Optimizing Memory Accesses for Non-Word Datatypes

As explained in Section **2.3.2**, X-HEEP memory is word-addressable. This means that the last two bits of the addresses are discarded, and every non-word memory access still results in a word access.

While this is not an issue for word datatype operations, it can lead to multiple accesses to the same memory location for other data types.

Consider an application that programs the DMA to copy an $N \times M$ matrix from location A to location B using half-words for both input and output. In the worst-case scenario, this could result in a waste of $(N \times M)/2$ memory accesses to read the data.

For repetitive patterns, this issue can be mitigated by the application developer, who could opt to use word accesses, reducing the time and memory accesses needed to move a half-word matrix by half. For byte datatype operations, $(N \times M) \times 3/4$ of memory accesses would be redundant due to accessing the same location, as shown in figure **2.14**.

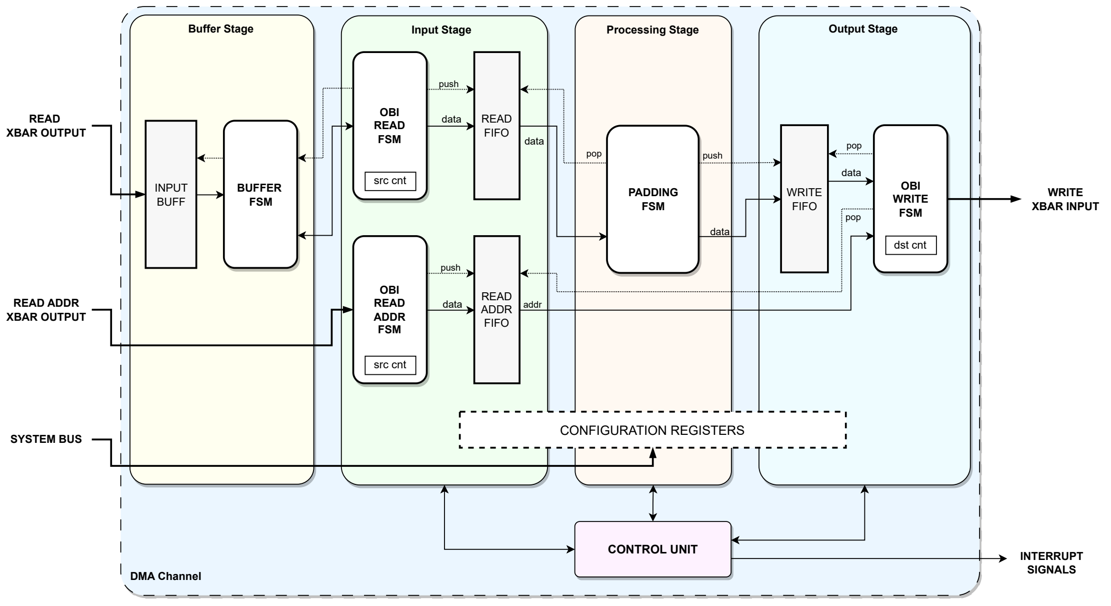


Figure 4.1: Updated DMA channel scheme with a new stage, the Buffer Stage, which implements the data buffer optimization for non-word transactions

However, this optimization is *application-specific*, whereas a hardware solution could transparently reduce memory accesses for the application developer. One of the simplest and most effective hardware approaches would involve a *32-bit buffer* to store read data.

A check on the next read address's 30 most significant bits (MSBs) could determine if the data is already in the buffer, and if not, a new memory read operation would begin. This feature could be easily introduced in the DMA channel structure by adding another stage, the *Buffer Stage*, before the *Input Stage*, as shown in figure **4.1**.

While increasing the buffer size is possible, it introduces additional complexity. With multiple buffers, logic would be required to fetch the correct buffer, handle removal, and track available spots.

The *tradeoff* between reduced memory accesses and the increased complexity of the DMA channel's Input Stage should be carefully evaluated. Reducing memory accesses would lower memory power consumption and decrease switching activity on the system bus, but this could be offset by the complexity needed to maintain such a system.

Nonetheless, optimizing memory access would be highly beneficial, as memory operations are often the bottleneck, as discussed throughout this thesis.

4.2 Further Optimizing Convolution Computation Using 3D DMA and Stream Accelerators

The current implementation of the `im2col` function utilizes a DMA system to produce one row at a time of the output matrix. This approach leverages the two-dimensional capabilities of the DMA introduced in this thesis: each row of the reshaped output is obtained by copying elements of the input tensor that overlap with the filter as it slides over the input, processing one channel at a time and one element of the filter per row.

However, by extending the DMA with 3D transactions and considering, for simplicity, a batch size of one, the DMA could transition across channels within a single transaction. This capability would allow it to write multiple rows of the output matrix simultaneously, specifically, three rows per transaction when dealing with three channels.

Furthermore, utilizing a 3D DMA would enable a modification of the `im2col` algorithm to write one column of the output matrix at a time instead of one row.
In this version of the `im2col` algorithm, each column is formed by copying the elements covered by the filter at a fixed spatial position—specifically, the same height (H) and width (W) offsets, across all channels. The elements are copied left to right and top to bottom within the filter window, but the filter remains stationary at this spatial location while moving across channels.
After collecting the necessary elements from all channels for this position, the filter is then moved to the next spatial location according to the stride, and the process repeats. A conventional 2D DMA can only copy a fraction $1/C_h$ of the column because it is limited to copying elements within a single channel, i.e. it cannot transition across channels while maintaining the same H and W offsets. In contrast, a 3D DMA can perform this operation across all channels in a single transaction, effectively copying the entire column corresponding to that fixed spatial position.

This alternative method of executing `im2col` can be exploited not merely to reduce the `im2col` transformation time but to accelerate the entire convolution process.
As explained in Section , convolution computed using `im2col` is performed as a matrix multiplication: **Filters** × **Input**. Each individual multiplication is essentially a vector dot product between the unrolled filter and a single column of the reshaped input.

The vector dot product itself is a *multiply-accumulate* (MAC) operation, involving the accumulation of the products of corresponding elements from the filter vector and the input column.

This presents an advantageous opportunity: with a 3D DMA available, the `im2col` operation can be performed column by column. Consequently, instead of storing the reshaped input for later use, each column can be directly utilized to compute the convolution output, processing one vector product at a time.

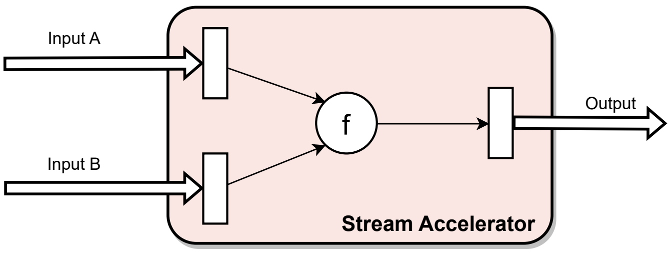


Figure 4.2: General scheme of a double input, single output stream accelerator

To implement this computation, a *stream accelerator* can be employed, a simple unit composed of two inputs and one output, each connected to FIFOs (First-In, First-Out buffers), as shown in giure **4.2**. One of the inputs, *Input A*, receives data from a DMA channel (e.g., Channel 0) programmed by a modified `im2col` SPC that implements `im2col` in a column-wise fashion, as describe before. This channel writes the reshaped input column element by element into Input A of the stream accelerator.

The second input, *Input B*, is supplied by another DMA channel (e.g., Channel 1), configured by the CPU at the beginning of the computation to copy the filter elements, which have been reshaped using `im2col` beforehand since the filters are constant. The third dimension of the DMA, now accessible in this extended DMA system, is utilized to reset the filter copy process after it has been used to compute the product with a single input column.

The stream accelerator processes the data by performing element-wise multiplication and accumulating the results until a predefined counter reaches zero. This counter is initialized with the number of products to accumulate, which is the size of the `im2col` column, given by $F_w \times F_h \times C_h \times B$, where F_w and F_h are the filter width and height, C_h is the number of channels, and B is the batch size. This parameter is set in a configuration register before computation begins.

Once the counter reaches zero, indicating that all necessary products have been accumulated, a single output value of the convolution is ready. A third DMA channel (e.g., Channel 2) is then used to transfer this result from the stream accelerator. The stream accelerator regulates the writing process using an additional trigger connected to this DMA channel. This setup is represented in figure **4.3**.

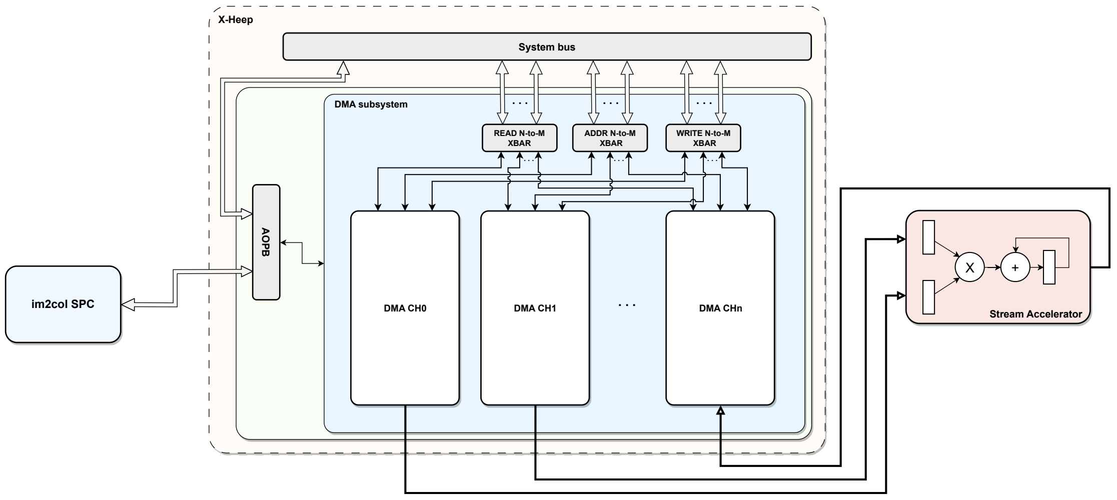


Figure 4.3: Scheme of a stream accelerator integrated in *X-HEEP* along with the *im2col SPC*

This synchronization is necessary because the output becomes available only after $F_w \times F_h \times C_h \times B$ accumulations, requiring the output DMA channel to wait for data readiness, similar to how a DMA waits for a peripheral device like SPI to signal that data is ready.

The ingenuity of this solution lies in the fact that, within the same time required to produce the reshaped input tensor, the convolution result is simultaneously computed and available.

This approach offers two significant advantages:

- **Zero Memory Overhead:** The largest issue with `im2col` reshaping is the substantial memory overhead required to store the expanded input tensor.

  In the current implementation, this issue can be mitigated through tiling techniques: by computing `im2col` row by row, it's possible to perform the convolution channel by channel, thereby reducing the memory needed to buffer the reshaped input by a factor of $1/C_h$.

  However, with the proposed solution, the reshaped input does not need to be buffered at all. The only memory overhead is the minimal storage required for the FIFOs in the stream accelerator.

- **Greatly Enhanced Performance:** There is a significant performance improvement over traditional methods of computing convolution using `im2col` techniques. By processing data directly as it is transferred, the proposed method reduces computational latency tenfold.

Moreover, the stream accelerator, as described, introduces *minimal area overhead*, making it a highly efficient and impressive enhancement to the convolution computation pipeline.

4.3 Conclusion

This thesis presents several enhancements to the *X-HEEP* microcontroller, addressing critical challenges in data transfer and manipulation, a major limitation in the application of edge computing applications, such as Edge AI.

The central innovation is the *advanced DMA system*, which, through its multiple independent channels, efficiently manages various data streams, allowing the CPU to either process data or enter low-power modes. The 2D transaction capabilities, including zero padding, transposition, and sign extension, further enhance performance and efficiency for data-intensive tasks.

The *Always-On Peripheral Bus* expands the potential for accelerator developers by enabling direct access to Always-On peripherals, such as the DMA subsystem. This feature enables highly efficient data transfer and manipulation operations while minimizing the area footprint on the accelerator side, all without adding complexity to the system bus.

The *im2col SPC* was introduced as an accelerator that utilizes the *AOPB* interface to fully exploit the novel DMA system, offering significant performance improvements over CPU-based routines while reducing power consumption. This approach achieves both a notable efficiency gain, up to **7.1x**, in the most critical step of GEMM-based convolution computations. Remarkably, these results were achieved with minimal area overhead, consuming only 0.43% of the total area in the *HEEPatia* project.

This demonstrates that the DMA-based accelerator approach proposed in this thesis not only delivers substantial improvements in performance and efficiency, but also shows that these gains can be realized with small, highly efficient accelerators. This is particularly valuable for small, low-power microcontrollers like *X-HEEP*, which operate within stringent area constraints, making it a key contribution to advancing the development of more powerful and efficient Edge AI applications.

All the designs introduced in this thesis, including the *DMA* system, *AOPB*, and *im2col SPC*, will be integrated into the *HEEPatia* SoC, a 16nm silicon implementation of *X-HEEP* designed for low-power edge-AI applications. Consequently, a thorough verification and testing campaign was required. To expedite and automate this process, while also assessing the performance of the *Unit Under Test*

(UUT), *VerifHEEP* was developed. This Python library supports the creation of a complete verification environment for *X-HEEP*, including model building and synthesis, dataset and golden results generation, FPGA board connection, and runtime estimation.

Thanks to *VerifHEEP*, both the *DMA* system and *im2col SPC* have been successfully verified through tens of thousands of unique tests on both simulation platforms and FPGA targets.

An important achievement of this thesis is the publication of all modifications to *X-HEEP* on its open-source GitHub repository. The project gained significant traction, with numerous institutions around the world using it as a foundation for developing their own accelerators and contributing to its ongoing evolution.

A key objective of this thesis was to ensure that all components were designed to be easily understood, utilized, extended, and adapted by future developers. To support this, extensive documentation has been created for both the *DMA* subsystem and the *VerifHEEP* library, along with thoroughly commented HALs and examples.

The impact of this work is already evident, as several DMA-based projects utilizing these new features have started development at both *ESL* and Polito's *VLSI* group even before the thesis's completion. This early adoption underscores the significance of these contributions in addressing the data movement bottleneck and advancing research in Edge AI.

Bibliography

[1] Andrew Moore. *Interview on Artificial Intelligence.* Interview by Forbes, Carnegie Mellon University. Dean of Computer Science at Carnegie Mellon University. 2017 (cit. on p. 1).

[2] John McCarthy. *Dartmouth Conference.* 1971 Turing Prize winner. 1956 (cit. on p. 1).

[3] William of Ockham. *Philosophical Writings: A Selection.* Ed. by Philotheus Boehner. Occam's Razor is often summarized as "entities should not be multiplied beyond necessity." Indianapolis: Hackett Publishing Company, 1990 (cit. on p. 4).

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems.* Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012 (cit. on p. 11).

[5] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going Deeper with Convolutions». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2015, pp. 1–9 (cit. on p. 16).

[6] Muhammad Usman, Siddique Latif, and Junaid Qadir. «Using deep autoencoders for facial expression recognition». In: *2017 13th International Conference on Emerging Technologies (ICET).* 2017, pp. 1–6. DOI: 10.1109/ICET.2017.8281753 (cit. on p. 19).

[7] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. «Learning agile and dynamic motor skills for legged robots». In: *Science Robotics* 4.26 (2019), eaau5872. DOI: 10.1126/scirobotics.aau5872. URL: https://www.science.org/doi/abs/10.1126/scirobotics.aau5872 (cit. on p. 20).

[8] NVIDIA Corporation. *Deep Learning Performance Documentation: Convolutional Layers* (cit. on p. 26).

[9] XuanTie. *im2col.c.* `https://github.com/XUANTIE-RV/csi-nn2/blob/main/source/reference/im2col.c` (cit. on pp. 27, 67).

[10] Michele Caon, Clément Choné, Pasquale Davide Schiavone, Alexandre Levisse, Guido Masera, Maurizio Martina, and David Atienza. «Scalable and RISC-V Programmable Near-Memory Computing Architectures for Edge Nodes». In: *arXiv preprint arXiv:2406.14263* (2024). URL: `https://arxiv.org/abs/2406.14263` (cit. on p. 29).

[11] 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.* 2024. arXiv: `2401.05548 [cs.AR]`. URL: `https://arxiv.org/abs/2401.05548` (cit. on p. 30).

[12] Alireza Amirshahi, Maedeh H. Toosi, Siamak Mohammadi, Stefano Albini, Pasquale Davide Schiavone, Giovanni Ansaloni, Amir Aminifar, and David Atienza. «MetaWearS: A Shortcut in Wearable Systems Lifecycle with Only a Few Shots». In: *Proceedings of the 2024 ACM/IEEE International Symposium on Wearable Computers (ISWC).* ACM/IEEE. 2024 (cit. on p. 31).

[13] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Alexandre Levisse, Miguel Peón-Quirós, and David Atienza. «HEEPocrates: An Ultra-Low-Power RISC-V Microcontroller for Edge-Computing Healthcare Applications». In: *EUROPRACTICE User Stories on Prototyped Designs* (2023), pp. 38–39 (cit. on p. 32).

[14] PULP Platform. *Open Bus Interface (OBI).* `https://github.com/pulp-platform/obi`. Accessed: [Insert date of access here]. 2023 (cit. on p. 36).

[15] Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. *SPOTS: An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-Matrix Multiplication.* Tech. rep. DCS-TR-756. Technical Report. Rutgers Department of Computer Science, 2021. URL: `https://arxiv.org/abs/2107.13386v2` (cit. on p. 65).