Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Programa de Maestría en Electrónica

Design of a library of generic accelerators of DNN-based inference algorithms for low-end FPGAs

para optar por el título de

***Magister Scientiae* en Ingeniería Electrónica**

énfasis en Sistemas Empotrados

con el grado académico de

Maestría

Luis Gerardo León Vega

8 de diciembre del 2022

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Luis León Vega

Luis Gerardo León Vega

Cartago, 8 de diciembre del 2022

Céd: 7-0233-0194

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Proyecto de Graduación
Tesis de Maestría
Tribunal Evaluador

Tesis de maestría defendida ante el presente Tribunal Evaluador como requisito para optar por el grado académico de maestría, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

[illegible]

Dr.-Ing. Carlos Adrián Salazar García
Profesor Lector

[illegible]

Dr.-Ing. Laura Cabrera Quirós
Profesora Lectora

[illegible]

Dr.-Ing. Juan José Montero Rodríguez
Revisor Externo

[illegible]

Dr-Ing. Jorge Castro-Godínez
Director de tesis

Los miembros de este Tribunal dan fe de que la presente tesis de maestría ha sido aprobada y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 8 de diciembre del 2022

Resumen

El bajo consumo de potencia, recursos computacionales escasos y los pocos grados de libertad para la optimización limitan la implementación de soluciones para la inferencia de aprendizaje profundo en el *edge*. La computación aproximada y la síntesis de modelos de alto nivel en C++ resultan prometedoras para el diseño de aceleradores genéricos especializables. Este trabajo propone un marco de trabajo de código abierto con librerías incluidas para la generación y evaluación automática de elementos de procesamiento (PE, *Processing Element*) vectorizados y aceleradores personalizables para la multiplicación-adición de matrices y para la convolución con tamaño de operandos, longitud y tipo de dato, y operandos aritméticos adaptables, usando síntesis de alto nivel desde descripciones en C++ genérico. A través de una exploración del espacio de diseño (DSE, *Design Space Exploration*) que varía la longitud del dato de 4 a 16 bits, los tamaños de operando de 2 a 8 elementos y los filtros desde 3×3 hasta 7×7, se evalúa el escalamiento del consumo de recursos, ciclos de reloj, eficiencia de diseño y la distribución del error, presentando una vista comprensible de cómo los parámetros afectan las implementaciones genéricas. La multiplicación-adición de matrices presenta un compromiso entre *granularidad vs eficiencia*, donde PEs grandes con longitudes de datos cortas son favorecidas por la eficiencia de diseño. La configuración más idónea es un acelerador con un único PE de 2×2, requiriendo anchos de dato de 16 bits con 4 bits de parte entera para mantener el error de 20%, logrando 9 GOP/s con 3.2% de eficiencia en una ZYNQ XC7Z020. En la convolución, se presenta la implementación de dos algoritmos: la convolución espacial y Winograd. La convolución espacial es mejor en términos de desempeño, mientras que Winograd en términos de consumo de recursos y tolerancia a los errores, requiriendo no menos de 4 bits para obtener 28 dB de PSNR con 10% de error medio. Finalmente, esta contribución puede ser adoptada en otros proyectos diferentes de redes neuronales dada la versatilidad de la programación genérica realizada en C++ y parametrización del diseño.

Palabras clave: computación aproximada, aprendizaje automático, redes neuronales, aceleración por hardware, inferencia, matriz de puertas programables.

Abstract

Low-power consumption, scarce computational resources, and reduced degrees of freedom for optimisation limit the implementation of deep learning inference solutions at the edge. Approximate computing and the synthesis from high-level C++ models report promising techniques for designing specialisable generic accelerators. This research proposes an open-source framework with built-in libraries for the automatic generation and evaluation of vector processing elements (PEs) and customisable accelerators for matrix multiplication-addition and convolution, with adaptable operand size, data bit-width, datatype, and arithmetic operands, using generic C++ high-level synthesis. Through the design space exploration (DSE) that varies the data bit-width from 4 to 16 bits, the operand sizes from 2 to 8, and the kernels from 3×3 to 7×7, this work evaluates the resource consumption scaling, clocks-to-solution, design efficiency, and error distribution, presenting a comprehensive view of how the parameters affect the properties of the generic implementations. The matrix multiplication-addition presents a trade-off between *granularity vs efficiency*, where the design efficiency favours large PEs with short data widths. The most suitable configuration was a single-PE accelerator with 2×2 operands, requiring 16-bit data width with a 4-bit integer part to keep the error below 20%, achieving 9 GOP/s with 3.2% efficiency in a ZYNQ XC7Z020. Regarding the convolution PEs, this document shows the implementation of two algorithms: a window-based spatial convolution and Winograd. The spatial convolution is better in terms of performance, whereas, the Winograd in terms of resource consumption and error tolerance, requires no less than 4 bits to get 28 dB PNSR values and 10% of mean error. Finally, this contribution can be adopted in other projects different from neural networks because of the versatility of the generic programming performed in C++ and design parameterisation.

Keywords: approximate computing, machine learning, neural networks, hardware acceleration, inference, field programmable gate arrays.

*to my beloved mother Roxana Vega Quesada,
family, and friends who supported me all the time during this
research and encouraged me to pursue my dreams.*

Acknowledgements

First, I want to thank my family, and friends for their unconditional support during my career and academic goals.

Moreover, this thesis was possible thanks to the contribution of the following students:

- Eduardo Salazar-Villalobos: matrix multipliers.
- Alejandro Rodríguez-Figueroa: convolution PEs.
- Fabricio Elizondo-Fernández: activation functions.
- Esteban Campos-Granados: drivers for the accelerators.
- Erick Obregon-Fonseca: acceleration daemon.
- David Cordero-Chavarría: proof-of-concepts and TensorFlow examples.

Also, to the following institutions:

- Instituto Tecnológico de Costa Rica: for their research scholarship grants for post-graduate students (2022).
- RidgeRun Embedded Solutions LLC: for their scholarships for collaborators (2021) and tools.
- eXact Lab S.R.L: for their scholarship programme for students.
- Ministero dell'Università e della Ricerca: for their Programma Operativo Nazionale (PON) scholarships.

Finally, special thanks to Dr.-Ing. Jorge Castro-Godínez, for his mentoring and support during the development of this research.

Luis Gerardo León Vega

Cartago, 8 de diciembre del 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural networks have become one of the last decade's most important machine learning methods. They outstand against classical statistical methods regarding solutions and computational performance in complex problems [1]. Nowadays, deep learning inference (DLI) is present in many trending applications across the computational spectrum: from smartphones and the Internet of Things (IoT), to computers, servers, and virtual assistants. Most current applications are cloud-based, where each device samples data from its inputs and uploads them to the cloud to perform the inference task, downloading only the final results [2]. However, there are applications such as self-driving cars, autonomous robots, satellites, and safety systems for which cloud-based inference is unsuitable due to connectivity issues, availability, privacy, reliability, and latency. This makes the cloud inadmissible to these critical applications and leads to the need for researching on intelligent *edge-based* solutions that are independent of network connections and self-contained.

Systems at the edge commonly have limited computational resources and constrained energy consumption, making them restrictive for most of the current DLI models [2]–[4]. Moreover, the evolution of deep learning (DL) at the edge tends to be steady regarding the computation power vs energy consumption ratio [2].

There are opportunities to address the edge-based inference on several fronts from these issues. Some alternatives rely on Graphics Processing Units (GPUs) (suffering a supply crisis since 2021), edge networks, or sophisticated microprocessors. Nevertheless, Field Programmable Gate Arrays (FPGAs) show a promising potential for balanced power consumption and algorithm computation power for inference tasks due to the capability of tailoring the hardware to a specific DLI model without wasting resources.

Taking into consideration the challenges presented above, this thesis focuses on:

- leveraging Artificial Intelligence (AI) to low-end FPGAs for a better exploitation of their low-consumption capabilities;
- development of generic and easy to customise IP Cores for DL with flexibility for optimisations and modularity for tuning the resource consumption;

- automation of design exploration tasks; and
- quantitative analysis amongst implementations for sets of selected parameters.

Based on the above, this work explores the automatic generation of vector processing elements (PEs) for matrix multiplication-addition, image convolution, and activation functions with adaptable operand size, data bit-width, datatype, and arithmetic operands. It focuses on using **generic programming** in standard C++ and **High-Level Synthesis** to implement PEs for modular accelerators in combination with approximate computing (AxC) for reducing the resource consumption footprint in exchange for results accuracy. AxC plus generic programming make it easier to exploit the error resilience of the applications, exchanging numerical errors to get less energy consumption and small designs, increasing the suitability of low-end FPGAs for DLI acceleration.

The main contribution of this research is an open-source library of DL PEs for DLI, where each PE is characterised by its parameters for rough error and resource evaluations without the need for running simulations that can take long runtimes.

1.1 Alternatives for DLI

The research community and the industry have addressed edge inference from several perspectives, from CPU optimisations and special SIMD instructions to hardware accelerators. Most of the hardware accelerators target small models, using non-floating-point numerical representations, concentrating the efforts on model optimisation to run gracefully on the vast diversity of edge devices.

One of the most popular vendors is NVIDIA, with the Jetson family within the market. Their embedded platforms combine a multi-core ARM microprocessor, integrated GPU with shared memory, and some accelerators like the NVIDIA Deep Learning Accelerator (NVDLA) and the Video Image Compositor (VIC). One of the solutions is the Jetson Nano, the most basic system offered by NVIDIA, consuming between $5W$ and $10W$ [5], equipped with a Maxwell GPU. The Jetson Xavier is the most powerful solution, equipped with a Volta GPU and a couple of NVDLA units, consuming up to $30W$ [6]. Although not mentioned, there are more alternatives from NVIDIA in between.

There are Application-specific Accelerators (ASA) for performing DLI. Google has presented two different architectures of Tensor Processing Units (TPU); one for Cloud applications and another for Edge. In terms of performance, the Cloud TPU [7] shows a peak performance of 92 TOP/s, having an energy efficiency of 1.23 TOP/s/W, while the Edge TPU presents a 2 Watts chip running at 2 TOP/s/W [8]. In contrast to CPUs and GPUs, both solutions are more energy efficient. An Intel Xeon Platinum 8260 CPU roughly reaches 0.18 TOP/s with an efficiency of 0.011 TOP/s/W [9]. In contrast, GPUs are more competitive in terms of performance. The NVIDIA Ampere can reach up to

1248 TOP/s with an efficiency of 3.12 TOP/s/W. The main difference relies on the consumption, where the Ampere GPU consumes up to 400 W [10], which is unsuitable for most edge applications.

1.2 FPGA-based DLI Accelerator

FPGAs are becoming an option due to their efficiency in computation per unit of energy. The essential advantage of these platforms over embedded systems is their high performance and flexibility, offering a better trade-off between computation power and energy consumption [11] than GPU-based System-on-Chips (SoCs). In the case of the ZYNQ 7000 family, they have a maximum power consumption of $5W$ and the Xilinx K60, up to $7.6W$, with better performance than the Jetson Nano [12], [13].

However, most solutions target high-end[1] FPGAs such as the Xilinx UltraScale+, Alveo, Kintex, and Virtex and their equivalents in other vendors. It leaves aside proper support for low-end FPGAs like the Artix-7, which consumes up to $1.6W$[2] [14]. In AI, Xilinx offers the Vitis AI software development kit (SDK), and Intel provides OpenVINO and OneAPI [15]. Most of them employ closed-source generic IP Cores such as the Deep Learning Processor (DPU) from Xilinx [16], closing the opportunity for tuning according to the error resiliency of the application.

The research community is interested in offloading inference on FPGA because of its low power consumption. There are attempts to synthesise models such as LeNet-5, VGG16, and You Only Look Once (YOLO) using both tool-assisted HDL and HLS. Some alternatives synthesise entire models on Vivado HLS, performing the entire inference within the FPGA, resulting in a 4.7x speedup on a Zybo 100Mhz with respect to an Intel Core i5 4590 3.3GHz in single-precision floating-point (FP32) without applying any quantisation [17]. Besides, there are case studies of CNN optimisation on FPGA. Some of these implementations manipulate the numerical representation using fixed-point numerical representation [18] increasing the performance compared to using FP32. An object detector based on the YOLO model managed to run at 1.88 TOP/s at 200MHz, favouring performance at a low clock speed. In this case, the FPGA acceleration offers 17.6 to 29.4 times less energy consumption than CPU/GPU processing for equivalent workloads. It shows the potential of FPGAs in this field in terms of speed and computation efficiency.

A recent survey highlights the importance of FPGA research for Internet-of-Things (IoT) since GPUs and CPUs are unsuitable due to energy consumption constraints [19]. Most contributions target model compression and approximation, taking advantage of the inaccurate nature of neural networks. Some proposals include non-linear quantisations [20], accelerator-aware and filter pruning [21], [22], sparse algebra acceleration, and knowledge distillation [23].

[1]FPGAs with more than 100K logic cells.
[2]According to Xilinx Vivado 2018.2 and Xilinx Power Estimator.

Successful frameworks within the scientific community have significantly impacted the DLI on FPGAs. At CERN, the `hls4ml` framework [24] implements a workflow that receives the model in either TensorFlow or PyTorch. Then, it performs an HLS conversion and creates a project that allows the user to tune the design: establishing optimisation goals and adjusting the numerical precision and the reuse factor. `hls4ml` provides compile-time design tuning, allowing design synthesis thanks to using C++ in HLS. The designs generated by `hls4ml` are singletons tailored to the model, synthesising the units to execute the entire model into the FPGA, requiring extensive resources in the target FPGAs. FINN is another framework that synthesises high-performance accelerators [25], taking into account the dataflow style (similar to `hls4ml`) and running the inference aid by the PYNQ framework [26]. Unlike `hls4ml`, FINN proposes operation-cost functions to get near-optimal implementations, performing an automated design exploration. Besides, it splits the design into several IP cores instead of using a singleton. FINN also reduces resource consumption by modifying the folding factor (how much the hardware units are recycled). It gives a certain degree of tuning for low-end FPGAs. However, both frameworks still need to be more flexible to explore approximation opportunities since their main goal is to ease the DLI on FPGAs through Python and not in the DLI design optimisation.

1.3 Contributions

The work in this thesis presents the following **novel contributions**:

- an open-source accelerators library for DL inference computation: the accelerators are implemented on templated C++11 and parameterised on the datatypes, numerical precision, arithmetic operators, operand sizes, and the number of processing elements;
- a design space exploration (DSE) framework for evaluating each configuration while tuning the accelerator's parameters, characterising the numerical error and resource consumption for each solution obtained from each set of parameters;
- a novel figure of merit to quantify the efficiency of the designs in terms of computational performance and resources consumption;
- a behavioural evaluation for each accelerator from the library, analysing the resource consumption and numerical error impact when changing the parameters. Thus, a complete library analysis is also given together with the source code;
- a comparison of peak performance of one of the most promising design solutions amongst the Artix-7 XC7A50T, the ZYNQ-7000 7Z020, and the Xilinx Kria K60;

This work focuses on developing accelerators for general matrix multiplication-addition and convolutions. Nevertheless, there are contributions in approximate PEs for activation functions and approximate computing.

This document is structured as follows: Chapter 2 starts with the current solutions given by the industry and academia to address the DLI and the background knowledge required for this work. Chapter 3 starts with the first level of abstraction, illustrating the process of developing and evaluating PEs. Chapter 4 continues with the second level of abstraction, illustrating the process for template accelerators. Chapter 5 explains the framework to generate both PEs and DLI accelerators. Chapter 6 shows the Design Space Exploration (DSE) results and an inter-FPGA evaluation. Finally, Chapter 7 concludes this work.

Chapter 2

Background and Related Work

2.1 Deep Neural Networks (DNNs)

Neural networks have become one of the last decade's most important machine learning methods. They are competent against classic statistical methods in solutions and computational performance [1]. McCulloc and Pitts started with the initial foundation of an artificial neuron (also known as perceptron) based on the human brain [27]. An artificial neuron can be seen as a linear combination of multiple stimuli feeding an activation function, which is often non-linear:

$$y = \sigma(\mathbf{w} \odot \mathbf{x} + b) \tag{2.1}$$

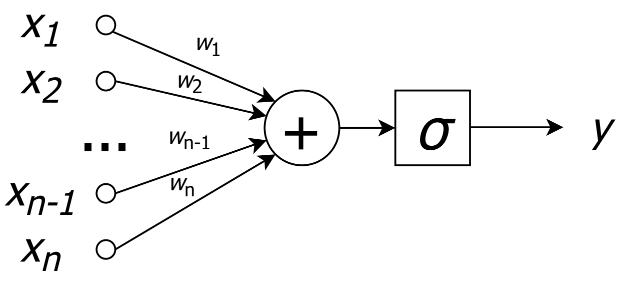
where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is the parameters' vector (also known as weights), b is the bias of the neuron, $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function, and y is the output. $\odot$ is the dot-product between two vectors. Figure 2.1 (a) illustrates how a perceptron is graphically represented.

Rosenblatt trained a perceptron to solve linear regression problems in 1958 [28]. Later, Minsky and Papert (1969) will spot that a single perceptron cannot deal with the XOR function [29]. It motivated the exploration of multi-perceptron layers, also introduced by Rosenblatt.

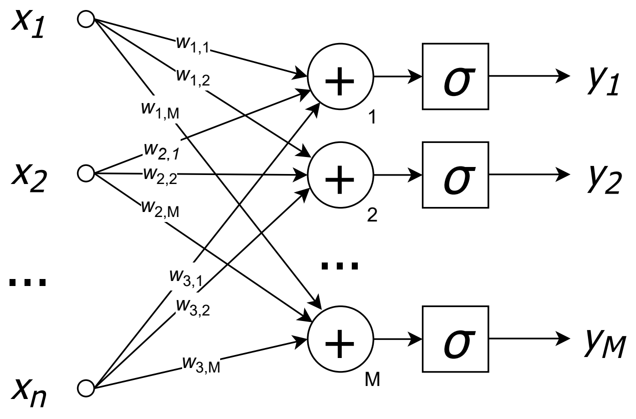
A multi-perceptron layer can be mathematically defined as

$$\mathbf{y} = \sigma(\mathbf{x}\mathbf{W} + \mathbf{b}) \tag{2.2}$$

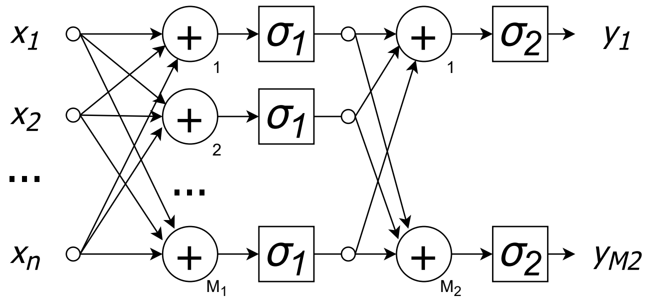
where now $\mathbf{y}$ and $\mathbf{b}$ are column vectors of the same size as the number of perceptrons, and $\mathbf{W}$ is a matrix whose columns are the weights of the perceptrons. σ is now a function that receives a vector $\sigma : \mathbb{R}^M \to \mathbb{R}^M$. It can be graphically represented as in Figure 2.1 (b). Then, the multilayer perceptrons arrived by cascading multiperceptron layers as in Figure 2.1 (c). In this former case, a network with multiple layers is also named as *deep*



(a)



(b)



(c)

Figure 2.1: Graphical representation of a neural network. **(a)** single perceptron, **(b)** multi-perceptron layer, and **(c)** multi-layer perceptron

neural network.

2.1.1 Common Operations in DNNs

Deep Neural Networks perform various operations that lead to their capacity to learn. Some layers contain parameters that adjust the network to learn a model given the data, such as the Fully Connected Layers (FCL) and the Convolutional Layers. The first can be seen as a linear combination of the inputs weighted by the parameters. In contrast, the Convolutional networks can be seen as filters trained to activate according to a feature.

This section briefly describes the operations taken into consideration within this work.

2.1.2 Fully Connected Layers

FCLs, often called Dense Layers, are the most common layers in DNNs [30]. Each perceptron is single-ended, producing a single output. An FCL is defined in (2.2). For concurrence, it is possible to define a matrix of inputs that leads to a matrix of outputs, as Equation (2.3) shows, where more than one sample (x vector) is processed at a time per FCL, leading to an operation over a batch of inputs. Then, the vectors become matrices containing n inputs, m neurons, and t samples [31].

$$\begin{bmatrix} y_0^{(0)} & y_1^{(0)} & \cdots & y_{m-1}^{(0)} \\ y_0^{(1)} & y_1^{(1)} & \cdots & y_{m-1}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ y_0^{(t-1)} & y_1^{(t-1)} & \cdots & y_{m-1}^{(t-1)} \end{bmatrix} = \begin{bmatrix} x_0^{(0)} & x_1^{(0)} & \cdots & x_{n-1}^{(0)} \\ x_0^{(1)} & x_1^{(1)} & \cdots & x_{n-1}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(t-1)} & x_1^{(t-1)} & \cdots & x_{n-1}^{(t-1)} \end{bmatrix} \begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,m-1} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ W_{n-1,0} & W_{n-1,1} & \cdots & W_{n-1,m-1} \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \cdots & b_{m-1} \\ b_0 & b_1 & \cdots & b_{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_0 & b_1 & \cdots & b_{m-1} \end{bmatrix} \tag{2.3}$$

However, compacting multiple time samples within a single matrix-matrix operation leads to an exchange between latency and computational speed. Latency, in this context, is the time required for the first sample t_0 to appear in the output. Increasing the number of samples to t will lead to a latency of t until a valid result. To exemplify the effect of the latency in a practical system, consider a camera feed at 30 frames-per-second, so each frame arrives in a 33 ms period. If the network requires $t = 16$ samples, the first frame inference will appear after $528ms$ in the output. In low-latency systems, the latency makes this simplification unsuitable [32].

Typical hardware accelerators such as the TPU and the GPU require the samples to be submitted in batches for efficiency reasons. The Google TPUv2 requires $N = 40000$ samples to reach an adequate inference time per image. In contrast, an NVIDIA K80 GPU typically requires $N = 8192$ samples for a convolutional network applied to an MNIST dataset in a classification task [33].

There are systems based on FPGAs that prioritise the latency, such as FINN from Xilinx, that performs the inference using a dataflow approach [25].

2.1.3 Convolutional Networks

Convolutional neural networks (CNNs) are a type of neural network widely adopted for pattern recognition in computer vision. CNNs deal with the computational complexity of computing dense neural networks for images due to their number of inputs (pixels). Their inspiration comes from the research on the visual cortex of a cat [34]. It describes how the neurons activate according to a stimulus coming from a screen. Then, the research was applied to pattern recognition of shift and deformation variant objects [35] and ML by using LeNet for classifying hand-written digits [36], introducing the concept of convolutional layers to artificial neural networks.

The relevance in neural networks relies on the compression of the trainable parameters, allowing NNs to learn features and compress them in feature maps (also known as convolution kernels) activated by a determined pattern when screening an entire image. The convolution operation between an image and a kernel is described by a Hadamard product and a sum reduction as an inner product in 2D per output pixel

$$Y_{i,j} = \sum_{n=-\kappa}^{\kappa} \sum_{m=-\kappa}^{\kappa} X_{i+n,j+m} K_{m,n} \tag{2.4}$$

where Y_{ij} is the pixel at the i-th row and j-th column of the output $\mathbf{Y}$, $X_{i,j}$ is the pixel at the i-th row and j-th column of the input $\mathbf{X}$, and $K_{i,j}$ is the pixel at i-th row and j-th of the feature map $\mathbf{K}$ of size $(2\kappa+1) \times (2\kappa+1)$. The former equation describes the convolution in the spatial domain.

There are other ways to compute the convolution by using *domain transformations*. The most known is the Discrete Fourier Transformation (DFT), which transforms the operands to the frequency domain and the convolution operation becomes a Hadamard product [37]. However, the FT is computationally expensive and involves complex numbers. Another method is the Winograd convolution [38], proposed by Shmuel Winograd in 1980. It pursues the same idea of domain transformations but involves real numbers only. The whole Winograd transformation, convolution and inverse transformation can be described as

$$\mathbf{Y}' = \mathbf{A}^T((\mathbf{B}^T\mathbf{X}'\mathbf{B}) \odot (\mathbf{G}\mathbf{K}\mathbf{G}^T))\mathbf{A} \tag{2.5}$$

where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{G}$ are the Winograd transformation matrices for transforming the output, the input window, and the kernel, respectively. $\mathbf{X}'$ is a window within the input matrix $\mathbf{X}$, and $\mathbf{Y}'$ is a window within the output matrix $\mathbf{Y}$. Winograd is an attractive method since it reduces the number of multiplications of the computation of convolutions. The matrices for the transformation of a 3x3 kernel and 2x2 output are

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \tag{2.6}$$

In this document, the PEs will be based on this Winograd configuration, given that 3×3 is one of the most common configurations in convolutional neural networks [39], [40]. Kernels larger than 11×11 are often rare and spatial methods are not as efficient as the FFT, according to the OpenCV library [41].

2.1.4 Activation Functions

Activation functions play an essential role within neural networks. They break the linearity within the neural networks, allowing them to avoid weight collapse. Without the activation functions, an FCL with weight matrices $\mathbf{W_0}, \mathbf{W_1}, \ldots, \mathbf{W_n}$ collapse into a single matrix $\mathbf{W} = \mathbf{W_0}\mathbf{W_1}\ldots\mathbf{W_n}$. They also introduce the ability to classify points with non-linear mappings of the data, making the embedded points linearly separable [30]. The activation functions must be first-order differentiable for training when doing the backpropagation. The most common activation functions are *softmax*, *Rectified Linear Unit*, *tanh*, *logistic*, and *arctan*.

Softmax and Logistic

The *softmax* function is a version of the logistic function used when having non-binary classifiers. It is often placed at the end of the classifiers as an activation function to extract the probabilities of each output class in a neural network, particularly after a fully-connected layer (FCL) [42]. A typical example is a LeNet5 model on the MNIST dataset [36].

The softmax function is defined as

$$\Phi(\mathbf{v})_i = \frac{e^{v_i}}{\sum_{j=1}^{k} e^{v_j}} \tag{2.7}$$

where v_i is the i-th element of the input vector $\mathbf{v}$ and k is the number of elements of the vector [30]. It involves the computation of the exponential function in a specific domain $S \subset \mathbb{R}$. The domain S can be determined according to the input and output domains of the FCL preceding the softmax function.

While *softmax* is used for multi-class classifiers and involves a vector as an input, the *logistic* activation function is equivalent for scalar inputs under the assumption that the labels are binary (two classes).

Rectified Linear Unit (ReLU)

ReLU is one of the most common activation functions used in the hidden layers of NNs. Its principle is to suppress negatives from the output vectors, leveraging the negatives to the weights of each layer and avoiding gradient vanishing [30]. The definition of a ReLU is

$$\Phi(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2.8}$$

where x is a scalar (single input).

There are variants of ReLU to include the negatives but with a penalty. For instance, the *leaky ReLU* penalises the negatives by a factor of 10,

$$\Phi(x) = \begin{cases} 0.1x & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2.9}$$

whereas the Exponential Linear Unit (eLU) adds an exponential function for $x \leq 0$

$$\Phi(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{2.10}$$

where α is a parametric constant.

Hyperbolic Tangent (tanh)

The tanh is another popular activation function. It allows the saturation of the outputs to be in a range between -1 and 1, where the derivative becomes smaller when reaching higher values and suppressing the growth of the weights. In the case of small values, the derivative is higher and robust on quick changes. Moreover, its use is commonly seen in binary classifiers [30].

Arctangent ($\tan^{-1}$)

The *arctangent* ($\tan^{-1}(x)$) can be seen as a logistic activation with a broader range ($[-\pi/2, \pi/2]$) [43]. However, it provides faster backpropagation than the logistic regression due to the operations: the exponentials are slower than the second-order polynomials.

The graphical representation of each activation function can be seen in Fig 2.2.

The advance in the work of optimising activation functions is presented in Appendix C.

2.1.5 Criteria for Choosing Activation Functions

The criteria for selecting activation functions rely on the kind of layer, the number of classes and empirical measurements. Some of these criteria are listed below [43]:

- ReLU functions are intended for hidden layers.
- Training speed (backpropagation speed) due to the activation derivatives.
- Gradient vanishing: the gradient sinks to zero.
- Number of classes: multi-class (softmax), binary (symmetric in Y axis).

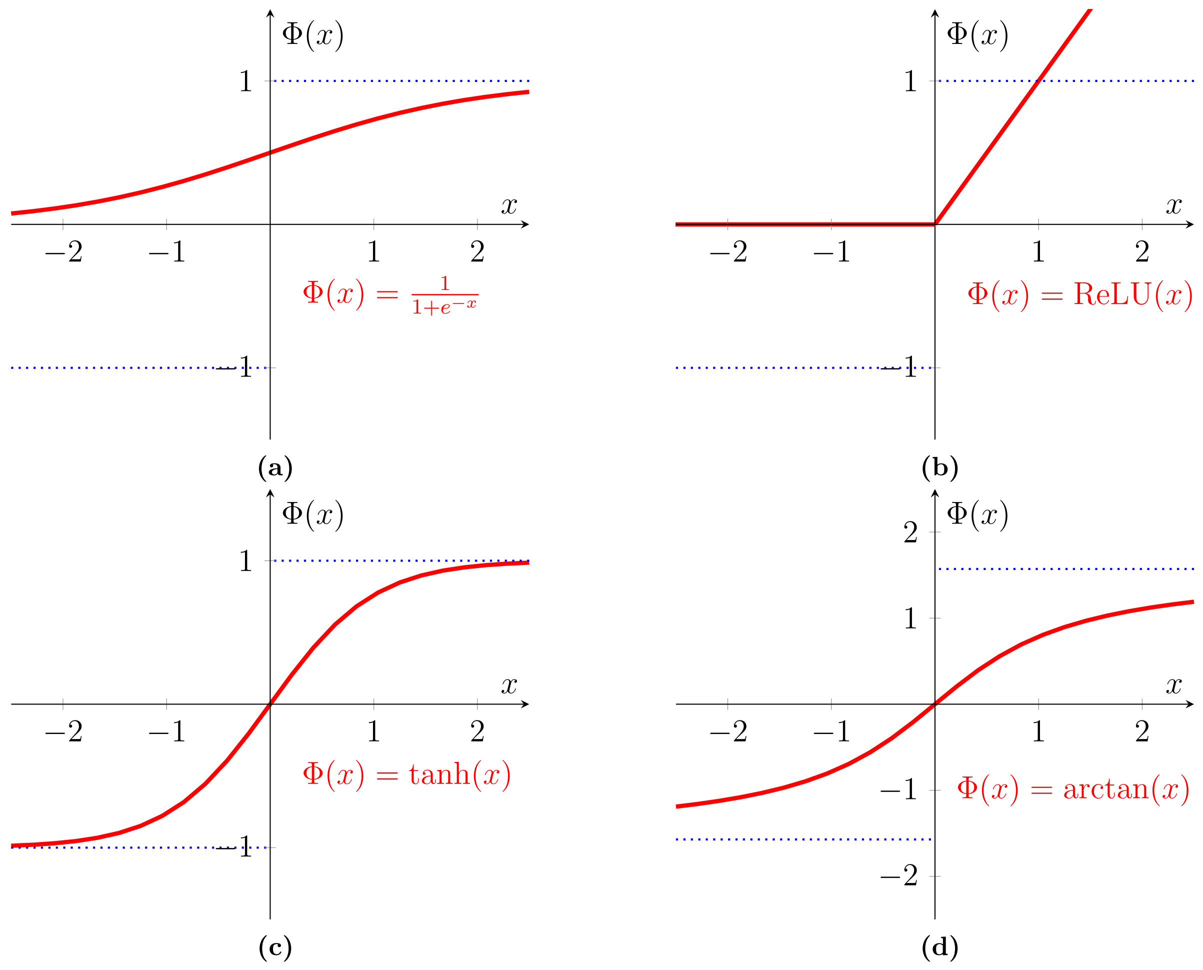


Figure 2.2: Graphical representation of the activation functions.

- Number of layers.
- Model target: regression or classification.

For quantisation purposes, the most interesting functions are those which are bounded in co-domain. For instance, the $\tanh(x)$ goes from -1 to 1, making it suitable for normalised numerical representations.

2.1.6 Common Optimisations in DNNs

Model size and computation are always a concern for running DLI. The model size impacts the memory footprint and the area of the execution units utilised for the model operations, like matrix multiplications and convolutions. It also impacts the bandwidth required for moving the model parameters between a host and an accelerator back and forth [19]. To overcome these issues, the most common optimisation is quantisation, which changes the data type from floating-point to fixed-point or integers. Modern GPUs support 4-bit data representations [10], leading to reductions from 32-bit to 4-bit, reducing the model size

Table 2.1: Popular data representations within the classical DLI processors

Data type	Number of Bits	Resolution	Maximum
`Float32`	32	1×10^{38}	3×10^{38}
`Float16`	16	5.96×10^{-8}	65504
`BFloat16`	16	1×10^{-38}	3×10^{38}
`Int16`	16	3×10^{-5}	0.9999695
`Int8`	8	7.8×10^{-3}	0.9921875
`Int4`	4	1.25×10^{-1}	0.875
`Int1`	1	0	1

by eight times. Some examples of data quantisations and representations to compress the models are summarised in Table 2.1 [19], [44], [45]

According to Table 2.1, most of the datatypes are a power of 2. As the number of bits decreases and how the bits are distributed within the representation, the resolution and maximum values allowed also decrease. Whereas the IEEE standard defines `FloatXX`, the `BFloat16` is a non-standard representation that tries to get the same resolution as the IEEE single-precision floating-point (`Float32`) [45]. This is beneficial in terms of model accuracy because the representation error is minimal. However, changing the data representation to the others may lead to accuracy degradation, which can be improved using quantisation-aware training techniques [46].

On the other hand, some neurons and weights do not contribute significantly to the model accuracy, so another technique is pruning weights and neurons [46]. Since pruning removes weights from the model, it forces the model to be sparse and represented by a key-value pair. Thus, some accelerators may be unsuitable for this technique or suffer performance degradation.

2.1.7 Open Challenges in DNNs

The Deep Learning field constantly evolves, introducing new tools for increasing its power and applications. Beginning from predicting non-linear functions up to Natural Language Processing, Generative Adversarial Networks and Computer Vision applications [30]. These applications rely on the capability of DNNs to automatically adjust their parameters and fit the model to a dataset in almost a black box fashion.

Training the networks is challenging: dealing with convergence speed and quality, overfitting, brain damage, vanishing gradients, and others. Moreover, the challenges also go to the hardware: how to increase the training speed. Apart from the training challenges, DLI is also challenging, and some are inherited from training. Enumerating some of these challenges, it is possible to find:

- Network accuracy degradation caused by quantisation: sometimes cured by quantisation-aware training.
- Energy efficiency: reduce the energy footprint of the model computations.
- Performance: speed up the inference process and increase the inference rate.
- Device capabilities: how to run the model in a well-performing fashion while trading with energy efficiency.

From the model's perspective, the model size is often a source of optimisation. The previous section explained that the most common optimisations are quantisation, which tends to change the data type and reduce the bit-width, leading to reductions from 32-bit to 4-bit, reducing the model size by a factor of 8. Besides, some neurons and weights contribute little to the model accuracy, so another technique is pruning weights and neurons.

Integrating hardware capable of executing quantised and sparse models is another field of study at the hardware level. NVIDIA Ampere architecture integrates Tensor Core units capable of running models in `INT4` (integer of 4-bits) and sparse models [10]. In this case, the hardware design has to take into consideration computation units tailored to low bit-width models, alternative ways of computing operations (i.e. Winograd for convolutions), frequency optimisation methods, data communication and access patterns, and multi-PE architectures with loop tiling (or array partitioning) [46].

Despite the optimisations described above promising a good speed-up, the models continue growing in space and complexity, adding more parameters and computation within the layers. At the hardware level, there are significant challenges to continue pushing the hardware beyond its current capabilities [19]:

- Von Neumann Bottleneck: massive parallelism can reduce the computation time; however, the data access becomes the bottleneck when extracting it from memory. In this case, computation units are faster than the memory, and it limits the inference rate.
- Sparse AI toolchains for hardware accelerators: despite the efforts from Google and Facebook on their frameworks (TensorFlow and PyTorch, respectively), there is no unified framework to program the AI accelerators.
- AI at the edge: running DLI at the edge has some limitations because of hardware restrictions. Some challenges derived from running inference at the edge are:
  - Co-optimisation between the model optimisation and the hardware intended to run the model.

- Hardware with a lower energy footprint, going beyond the classical techniques, i.e. using analog computation.
- Training-on-device: most edge devices are only capable of running inference; however, learning at the edge can open better opportunities in application evolution.

This work will address the inference from low-power AI inference at the edge.

2.2 Classical Architectures for Running DLI

This section presents an overview of the current architectures used for processing and accelerating DLI, including typical systems such as microprocessors, GPUs, and ASICs. At the end of the section, the analysed architectures will be concentrated in a table summarising common characteristics and novelties.

2.2.1 Central Processing Unit (CPU)

In most systems, the CPU will be the chosen unit to perform the inference tasks without accelerators. However, CPUs are usually equipped with vector units that speed up the processing of many operations that a DLI task involves.

Intel equips their processors by adding SIMD instructions that trigger vector execution units under the hood. They are often the Streaming SIMD Extensions (SSE), Advanced Vector Extensions SIMD instructions (AVX), and Advanced Matrix Extension (AMX) [47]–[49]. ARM, another microprocessor architecture designer, proposes NEON and Scalable Vector Extensions (SVE) under the same idea of vector units but supporting unaligned memory and variable vector size [50], [51].

From the vector mentioned above, AVX-512 and AMX are intended for DL acceleration. AVX-512 presents VNNI [48], which performs a dot product between two vectors and adds a third vector as a bias:

$$y = \mathbf{a} \cdot \mathbf{b} + c \tag{2.11}$$

where $\mathbf{a}, \mathbf{b}, c$ and y can be represented in (un)signed integers (8, 16, and 32 bits), and 32-bit IEEE-compliant floating-point. The number of entries of the vectors $\mathbf{a}$ and $\mathbf{b}$ will vary depending on the datatype, considering that each vector register has 256 bits. Thus, for 8-bit operads, it can process up to 32 vector entries and for 32-bit operands up to 8 entries. Moreover, VNNI handles the overflow by doubling the result register. If $\mathbf{a}$ and $\mathbf{b}$ are represented in 8-bit integers, the results y will be a 16-bit integer.

In terms of performance, CPU processing is known to be the less-performing unit of operations per energy consumed. Thermal design power (TDP) is often used to determine how much the CPU can consume and dissipate. For an Intel Xeon Platinum 8260L @ 2.40GHz, the TDP is 165 Watts [9]. To estimate the performance of a processor, it is possible to use:

$$P_{\text{cpu,peak}} = f_{\text{base}} \times n_{\text{cores}} \times n_{\text{ops/core,entry,clock}} \times n_{ventries} \tag{2.12}$$

where the peak performance P_{peak} is given by the product of the base frequency f_{base}, the number of physical cores n_{cores}, and the number of operations per clock cycle, entry and core $n_{\text{ops/core,entry,clock}}$, and the number of entries that fit in a 256-bit vector $n_{ventries}$. In the case of the frequency, previous research has demonstrated that the clock is incapable of running at the base frequency when enabling the AVX-512 units [52], leading to underperformance. However, neglecting the clock slowdown, the performance of the Xeon 8260L for 8-bit operands is roughly

$$\begin{aligned} P_{\text{cpu,peak}} &= f_{\text{base}} \times n_{\text{cores}} \times n_{\text{ops/core,entry,clock}} \times n_{ventries} \\ &= 2.40\text{GHz} \times 10^9 \times 24 \times (2\text{ops}/2\text{cycles}) \times 32 \\ &= 1843\text{GOP/s} \end{aligned}$$

where AVX performs two operations (addition + product) in two clock cycles. It leads to an energy-aware performance (P_{peak}/TDP) of

$$P_{\text{E,cpu,peak}} = \frac{1843\text{GOP/s}}{165W} = 11\frac{\text{GOP/s}}{W} \tag{2.13}$$

AMX presents a more interesting approach to computate, allowing operating over matrices and using internal fused multiply-add operations:

$$\mathbf{C}_i = \mathbf{AB} + \mathbf{C}_{i-1} \tag{2.14}$$

where **A**, **B**, and C are matrices and each matrix unit performs a matrix multiplication-accumulation.

It is still unclear how many clocks AMX will take to complete the operations and the processors equipped with this vector extension, making (2.12) unsuitable to be applied. In terms of the operand size, the registers are based on 64-byte tiles with 16 rows each. Having 8-bit integers as entries, it would be possible to operate 16×8 matrices at maximum.

On the other hand, NEON and SVE have variable-size registers (from 128 to 2048 bits), allowing unaligned memory transfers and giving more flexibility. They are intended to address various HPC workloads, including operations required for DL. It has been reported that SVE achieves a 3x speedup compared to a non-accelerated approach [53]. However, there need to be details about their implementation for a good estimation in terms of performance.

2.2.2 Graphics Processing Unit (GPU)

The GPUs are one of the most popular DL training and inference accelerators, managing to achieve significant speedups compared to CPUs [54].

In the case of NVIDIA, the most popular GPU vendor proposes a type of execution unit for computing matrix operations, called *Tensor Cores*, introduced in its Volta architecture [55]. A tensor core combines the matrix multiplication and the addition, leading to a fused multiply-add unit:

$$\mathbf{D} = \mathbf{A}\mathbf{B} + \mathbf{C} \tag{2.15}$$

where the operands are likely to be 16-bit or 32-bit floating-point operands. In newer architectures, like Ampere, it also supports integers of 4 and 8 bits [56]. Moreover, each matrix has a size of 4×4.

Moreover, tensor cores group the matrix operations in batches for concurrent execution, increasing the parallelism. In the Volta architecture, the final tensor (batch of matrices) has a size of $4 \times 4 \times 4$. However, it can operate over greater matrices thanks to the *Warp-level Matrix Multiply and Accumulate* (WMMA) [57].

Tensor cores are available in a stream multiprocessor (SM). The Tesla V100 has 640 tensor core units, leading to a theoretical performance of 125 TOP/s. It can be estimated by using

$$P_{\text{gpu,peak}} = f_{base} \times n_{\text{tcores}} \times n_{ops} \times \frac{32}{b_{\text{data}}} \tag{2.16}$$

where the peak performance of the GPU is given by the product of the base clock frequency, the number of tensor cores, the number of operations per tensor core, and the proportion of the 32-bits that the numerical representation requires. For the Tesla V100, it leads to

$$\begin{aligned} P_{\text{gpu,peak}} &= 1.530\text{GHz} \times 640 \times (4 \times 4 \times 4) \times \frac{32}{16} \\ &= 125337.6\text{GOP/s} \end{aligned} \tag{2.17}$$

where the data used is a 16-bit floating-point representation (b_{data}).

In terms of power consumption and energy-aware performance, the TDP for the Tesla V100 is 300 Watts [55]. It leads to a performance of

$$\frac{125337.6\text{GOP/s}}{300W} = 417.8\frac{\text{GOP/s}}{W} \tag{2.18}$$

which is 37x more efficient than the Xeon CPU presented before.

2.2.3 Application-Specific Accelerators (ASA)

GPUs are often unsuitable because of their power consumption, and edge solutions require a more compact device for energy-constraint environments. Google has developed the Tensor Processing Unit (TPU), whose first version arrived in 2015, oriented to servers. Google Coral, instead, targets edge solutions [58], [59].

Server TPUs are mainly composed of a matrix multiply-addition unit of 256×256 operators, allowing them to operate over 8-bit integer numbers and achieve up to 128K operations (multiply + add) per clock cycle [58]. The novelty of TPUs relies on the systolic array of elements, broadcasting the operands of the matrix similar to a data stream.

To estimate the performance of a TPU, it is possible to determine it as the product of the clock frequency and the number of operations:

$$P_{\text{tpu,peak}} = f_{base} \times n_{rows} \times n_{cols} \times 2 \tag{2.19}$$

where the number of operations is the product of the number of columns, rows, and the two operations: multiply and add. Thus, for a TPU running at 700 MHz, it results in

$$\begin{aligned} P_{\text{tpu,peak}} &= f_{base} \times n_{rows} \times n_{cols} \times 2 \\ &= 700\text{MHz} \times 256 \times 256 \times 2 \\ &= 92000000\text{MOP/s} = 92\text{TOP/s} \end{aligned} \tag{2.20}$$

In terms of energy-aware performance, the TPU has a TDP of 75W [7], leading to

$$\frac{92\text{TOP/s}}{75W} = 1.23\frac{\text{TOP/s}}{W} \tag{2.21}$$

which is almost 3x more efficient than a GPU.

Nevertheless, the Google Coral achieves 2 TOP/s per Watt, achieving even better performance than the server version [59].

Other accelerators such as the Ethos-78 from ARM, the NVIDIA DLA, and the Xilinx DPU are examples of ASAs that are promising in terms of energy-aware performance [16], [60], [61].

2.2.4 Taxonomy

Table 2.2 condensates the information presented in the previous architectures. It is possible to compare to notice that the GPUs tend to stay at 1 TOP/s/W of energy-aware performance, while the CPU is the worst-performing device for executing DL operations. The TPU presents the best benefit in terms of energy and performance.

The state-of-the-art research does not specify the matrix size for the Tensor Cores in the case of the Turing and Ampere architectures. However, it is possible to assume that it is a multiple of 4 since the Tensor Core in the Volta Architecture works on 4×4 matrices. The energy-aware performance P_E is computed assuming 8-bit operands (16-bits if not supported). The data is taken from [7], [9], [48], [49], [51], [53], [55], [56], [61].

Table 2.2: Taxonomy of the researched architectures

Device	Architecture or Extension	Operative Frequency	Datatype	OPS	Matrix or Vector	Size Matrix/Vector	TDP (W) [$\text{P}_E(OP/s/W)$]	Especialización de la arquitectura
TPU	Version 1	700 MHz	`int8`	92 TOPS/s	Dense matrix	256×256	75 [1.23T]	Matrix Multiplier Convolution
Intel Xeon Platinum 8260L Processor	VNNI	2.40 GHz	`int8` `int16` `int32` `fp32`	Equation 2.12	Vector	256 / b_{data}	165 [11 G]	Dot product
N/A	AMX	N/A	`int8` `bf16`	N/A	N/A	N/A	N/A	Multipurpose FMA units Dot product
N/A	SVE	N/A	N/A	N/A	Vector	N/A	N/A	Variable Vector Size
Tesla V100	Volta Tensor Core	1530 MHz	`fp16` `fp32`	125 Tensor TFLOPS	Dense Matrix	4×4	300 [417.8 G]	Matrix multiply by Tensor Cores
Quadro RTX 6000	Turing Tensor Core	1455 MHz	`fp16` `int8` `int4`	130.5 Tensor TFLOPS with `fp` 261 Tensor TOPS with `int8` 522 Tensor TOPS with `int4`	Dense and sparse matrix	Flexible	260 [1.00 T]	Matrix multiply by Tensor Cores
NVIDIA RTX A6000	Ampere Tensor Core	1800 MHz	`fp16` `fp32` `bf16` `tf32` `int8` `int4`	154.8 Tensor TFLOPS with `fp` 77.4 Tensor TFLOPS with `tf32` 309.7 Tensor TOPS with `int8` 619.3 Tensor TOPS with `int4`	Dense and sparse matrix	Flexible	300 [1 T]	Matrix multiply by Tensor Cores
Ethos-78	NPU	1 GHz	`int8` `int16`	1-10 TOPS/s	Dense and sparse matrix	8×8	N/A	Winograd

2.3 Approximate Computing

Most AI applications involve complex algorithms and high computational power, requiring power-hungry platforms due to the clock speed and the number of computing units [62]. However, some applications, such as IoT and mobile devices, cannot afford traditional computing units i.e. GPUs because of the power/energy budget and the available computing resources.

Some applications can be approximated due to their error resilience [63], sacrificing results precision to reduce the need of complex arithmetic units, such as FPUs (Floating Point Units). This paradigm is called *Approximate Computing* (AC) [64].

The primary constraint to implementing an application using the AC paradigm is the resilience to the error in its results. Given their inherent inaccuracy, probabilistic applications are often good candidates for approximation, such as Machine Learning based on DNNs [62]. However, having error resilience is not sufficient. Some applications can be more suitable to be approximated than others, depending on the relevance of their results in the following steps and how critical they are.

Generally, AC applications include parameters to adjust the degree of approximation, i.e. the numeric precision of a given stage in a baseline application. The adjustment iterates in adjusting and evaluating the impact, verifying if the results are still suitable for the given application [62]. Shafique et al in [62] present this procedure in their paper, applying approximate computing to Machine Learning applications. As they explain, the *quality knobs* (named like that due to the quality change in the precision results) give control over the *accuracy-power trade-off*.

Before continuing the analysis of approximate computing and their impact in the application development field, it is required to study their possible ways of implementation and some cases of study which have put into practice this paradigm.

2.3.1 Methods of Implementation

Approximate computing approaches modifications on the hardware or the software [63]. The suitability is determined by the flexibility of the application and the hardware restrictions [65].

Software-like Approximations

These approximations involve changes in the algorithm and data types to introduce adjustable knobs, allowing reductions in the computational/memory requirements of a given application software application in exchange for accuracy. Some ways to implement are listed below:

- Loop perforation
- Numerical representation
- Approximate formulas or functions

Loop perforation is a technique applied to code loops to reduce the computation workload and time to have the results available to read. Besides, it can reduce energy consumption and enhance performance [66].

The main idea of loop perforation is to skip one or more iterations depending on the impact on the results, letting the possibility of dynamically determining whether an iteration can be skipped or not. To demonstrate that, consider the following exact form of a *for loop* in C++:

```
for(int i = 0; i < b; i++) {
    // Some instructions
    // ...
}
```

The proposal presented in [66] suggest that a *naïve* loop perforation may be

```
for(int i = 0; i < b; i+=n) {
    // Some instructions
    // ...
}
```

where $n-1, n>0$ is the number of cycles skipped consecutively.

Furthermore, it is possible to dynamically determine the number of cycles that can be skipped in a row, depending on the program status. It proposes a change at the compiler level with *LLVM* that makes the naive loop perforation automatic.

Numerical representation is another way to introduce approximations in an application. It involves changing the data type to represent a number, e.g., truncating a floating-point number and making it an integer. Changing the numeric representation from floating-point to fixed-point in a DNN can reduce weight storage by up to 36% and the multiplier power required by up to 50%, sacrificing precision in their results by changing the numeric representation [67]. There are also face recognition applications that have reported about 62.49% of energy saving. In practice, this change is often known as *quantisation* in Data Science and is widely used in production applications.

To exemplify this change, consider the following snippet:

```
// Number represented in floating point
float floatingN = 0.95863445262;

// To fixed point given by 16 bit signed integer
int16 fixedN = 0.95863445262 * 32768;

// Result: 31412. Now, the number change to: 0,958618164

// Error:
float error = 0.95863445262 - fixedN / 32768;
error /= 0.95863445262;

// Resulting error: 0.002%
```

According to the example, the 16-bit fixed-point representation leads to numeric errors below of 0.01%, considering that the maximum result deviation is given by $\sigma_r = 1/2^{15}$.

Approximate formulas or functions are a way to simplify a function computation using a less computationally exhausting expression to avoid using complex operators that can delay results delivery, e.g. using FPUs. Besides, some numerical and logic techniques simplify and speed up the computation of complex functions.

In the field of DLI, exponential-based activation functions are often slower in prediction and backpropagation due to how the exponential is computed. A typical implementation in the standard C library uses an iterative algorithm to have a general function that works in the entire domain. However, in DLI, by controlling the range of the inputs and outputs of each layer, the domain can be restricted in a range such that the exponential function can be defined only in a range with Look-up Tables (LUTs) and linear interpolation or the Taylor Series [68] (See Appendix C).

Other alternatives are code analysis engines, which make machine code generation smarter and data-responsive, e.g. *approximate-aware programming languages* [69]. A simple example of optimisation at code level can be:

```
/*
  Dealing with multiplications
*/
// Multiplying by a 2^n factor
int number1 = 5;

// Operation
int result1 = 5 * 16;
int result2 = 5 << 4;

// Both results will be 80
```

Despite being an exact example, it can be extrapolated to approximate cases. Consider this other case:

```
/*
  Dealing with multiplications
*/
// Define a const given by a formula
const float factor = 4.05;
const int equivalent_shift = 2;

// The variable is stored in int32 format
uint8 number1 = 5;

// Operation
uint16 result = (uint16) number1;
result = number1 << equivalent_shift;

// Error: 1.23% in the result
```

In the example presented above, the factor can be approximated to a binary shift, neglecting the fractional part of the constant *factor*. The error is kept near 1.24% from the range from 0 to 255 allowed by the *int8* data type.

In applications where this error is acceptable, there is room for the compiler to perform this change. This approximation method can avoid the multiple clock cycles required by the FPU to perform the multiplication [70] and only need one cycle to perform the multiplication using binary shifts, saving time and speeding up the execution of an application.

Hardware Approximation

Hardware can also be the subject of approximation. DLI requires matrix multipliers, convolutions and activation functions to run. The PEs in charge of performing these operations can integrate software-like approximations like the ones described before.

A model can be compressed to save space through quantisation and pruning (*numerical approximations*). Approximate computing can offer more than space savings, including saving energy when combining these DL optimisations with hardware approximations. Therefore, the PEs can integrate these optimisations to get an extra benefit in energy and latency.

Likewise, *function approximation* is another approximation that can be useful for computation. Domain-specific functions can work in DLI because of its resilience to errors. The hardware can implement functions like `exp(x)` according to the numerical range. For instance, if the numerical representation is a normalised fixed-point number ($[-1, 1[$), the function can be defined in a limited domain. Thus, first-order Taylor Series [71] and a

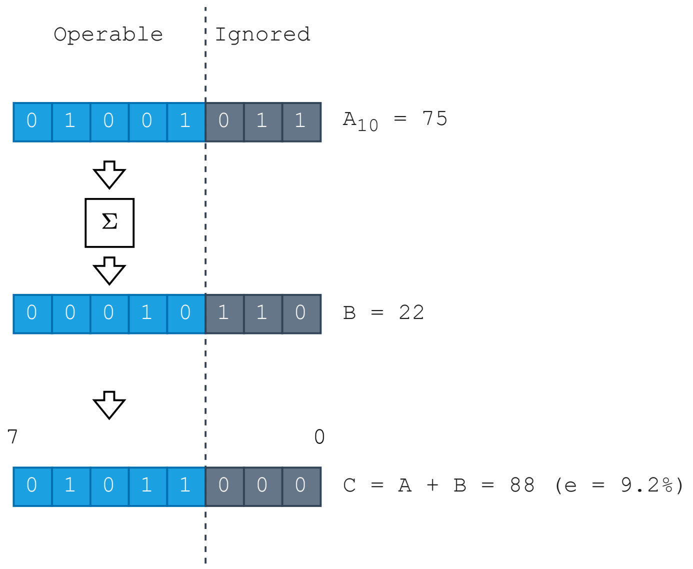


Figure 2.3: Adder with LSB drop. The last three bits are ignored and set to zero. There are other possibilities such as bypassing one of the operands or set a constant number

piece-wise linear interpolation [72] are good candidates with lower logic complexity.

From another perspective, *approximate logic* is another topic to inspect within DLI. One of the possibilities is to *drop the less significant bits* from the operation, operating only in the upper part of the operands [73], [74] (see Figure 2.3). Within the same idea, dropping the LSBs does not only apply to arithmetic units but to memory and other hardware.

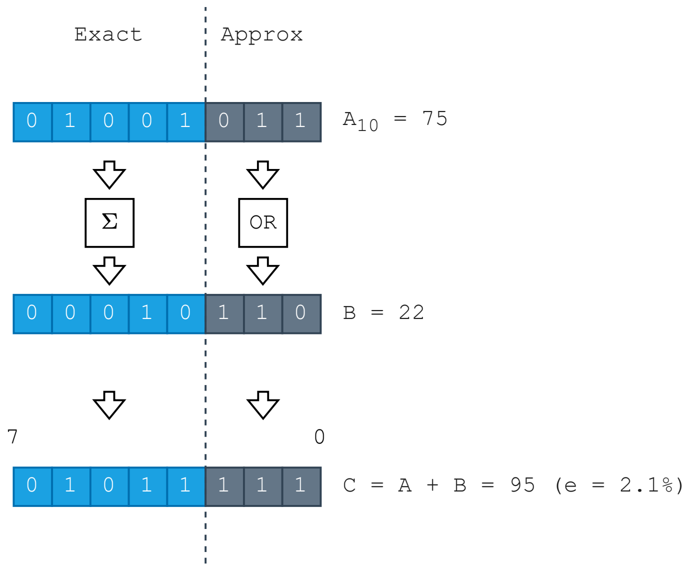


Figure 2.4: Approximate adder with OR in LSB. The last three bits are operated by a bitwise OR. There are other possibilities such as using a XOR or a half-adder.

Other approximations consist of computing parts of the operands with approximate hardware and others with exact hardware. From the last approximation, instead of dropping the LSBs, they can be operated using OR or XOR gates [75], [76], as it is illustrated in

Figure 2.4.

There are other hardware approximations at a deeper level; however, they are not part of the scope of this work. Parallel work has derived the results in Appendix D.

2.4 FPGA Implementation Workflow

The classical way of working with FPGAs is using Hardware Description Languages (HDL) such as Verilog and VHDL. It involves describing the hardware in Register Transfer Logic (RTL). Then, this description in HDL is simulated to verify the behaviour of the hardware implementation. After, the implementation is synthesised (or adapted) to the platform and its capabilities. There is another simulation for verifying the synthesised design, which is still in RTL. Once the post-synthesis simulation is finished, the design implementation in the FPGA happens, placing the logic, configuring the logic cells and routing. After having the design implementation ready, the process concludes in the FPGA programming [77].

However, describing the hardware using HDL is complex and can be a source of errors due to the number of lines to write to implement the design logic. High-level Synthesis (HLS) has come as an alternative to reduce the work of implementing complex designs in FPGAs. The most popular HLS tools are the Xilinx Vivado HLS/Vitis [78], and Intel Quartus HLS [79], which use untimed C++ code that focuses on the functionality and the implementation details are passed through directives within the code or in a configuration file [80]. Other alternatives, such as Chisel, an extension of Scala to describe hardware with the Object Oriented Programming paradigm, make it easier to recycle and parameterise the design [81].

2.4.1 Architecture Topologies

This section aims to explore the capabilities of HLS in the hardware description. This work will limit its scope to explore the capabilities of Vivado HLS using C++ code as the input.

One of the exciting capabilities of HLS is that a single piece of C++ code can lead to various design implementations controlled by compiler directives (a.k.a. as pragmas). In other words, it is possible to explore several designs using the same code and perform a design space exploration with directives. This document will refer to each of these possible designs as *design solutions*.

Table 2.3 shows the directives supported by Vivado HLS 2018.2. In this context, *cores* are FPGA resources or RTL library components that implement operations such as division, multiplication, modulus, and others. *Functions*, on the other hand, refer to C++ functions that behave as hardware blocks, designs or modules. *Data containers* are often `structs` in C++ and *loops* as for-loops (while-loops are not fully supported in HLS).

Table 2.3: Vivado HLS 2018.2 design directives [82]

Directive	**Description**	**Scope**
`ALLOCATION`	Specifies the limit of a resource or function (force HW sharing)	Functions, cores
`ARRAY MAP`	Combines small arrays into a single array for memory recycling	Arrays
`ARRAY PARTITION`	Partitions large arrays into several small arrays.	Arrays
`ARRAY RESHAPE`	Combines the elements of an array in larger words to improve memory access.	Arrays
`DATA PACK`	Combines the elements of a struct into a single word to improve access.	Data containers
`DATAFLOW`	Enables task-level concurrency. It makes functions and loops work concurrently	Loops, functions
`DEPENDENCE`	Provides a hint about dependencies. It improves loop pipelining and lower intervals	Loops
`INLINE`	Inlines logic, avoiding inter-function communication. It reduces latency and logic.	Functions
`INTERFACE`	Dictates how ports must be implemented in the RTL	Function arguments and return
`LATENCY`	Specify the constraints in terms of latency	Function, loops
`LOOP FLATTEN`	Allows collapsing nested-loops into a single loop with enhanced latency	Loops
`LOOP MERGE`	Combines consecutive loops for enhanced latency and resource sharing	Loops
`OCCURRENCE`	Specify the occurrence of a code fragment within a loop	Loops, functions
`PIPELINE`	Reduces initialisation interval by allowing the inner functions to execute concurrently	Loops, functions
`STREAM`	Specifies that an array must be implemented as a FIFO	Arrays
`TOP`	Declares a function as the top module	Functions
`UNROLL`	Unroll loops to create independent execution units or paths	Loops

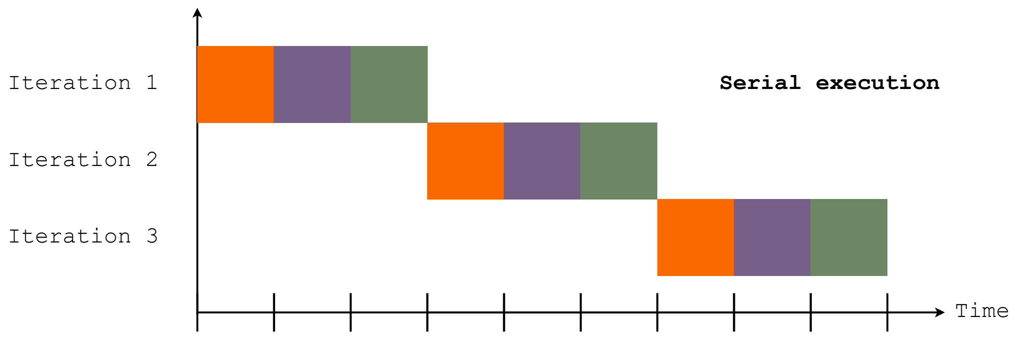


Figure 2.5: Serial architecture illustration. The new iteration executes until the previous one has finished

Considering the directives in Table 2.3, HLS can create design solutions by combining two or more directives, allowing users to explore several architectures in less time. It is essential to mention that these directives are disabled by default if not specified.

Some of the architecture topologies are specified below.

Serial Design

It is the case when most directives are disabled or absent, particularly PIPELINE and `DATAFLOW`. The modules in this architecture are serialised, i.e. a module can be executed until the preceding one has completed its work. Figure 2.5 illustrates through a timing diagram how this architecture behaves. To achieve that, please, consider the following code:

```
void MyFunction (int a, int b, int c inc &d) {
  int temp1;
  int temp2;

  module1(a, b, temp1);
  module2(a, c, temp2);
  module3(temp1, temp2, d);
}
```

Without any directive, this code behaves asFigure 2.5 illustrates, where `module2` starts to execute until `module1` finishes. The same is for `module3`, which requires `module2` completed before starting.

In this document, this type of design is referred to as *baseline*.

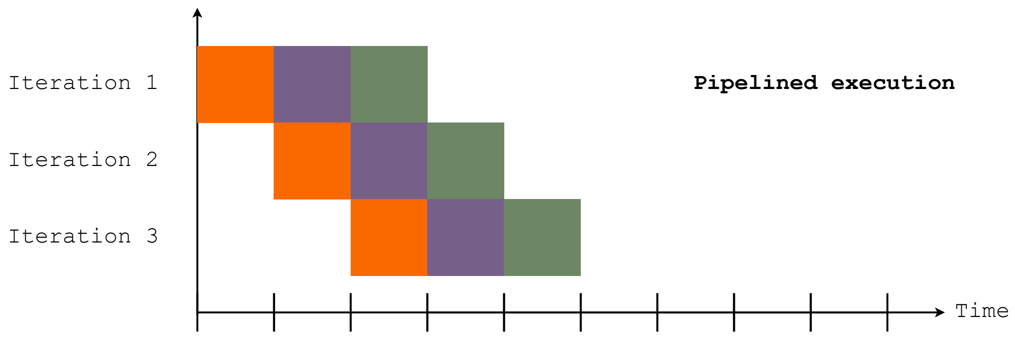


Figure 2.6: Pipeline architecture illustration. The new iteration does not require the preceding one to start. It just needs that the first unit finishes the current iteration to begin.

Pipeline Design

The pipeline design is one of the most promising design architectures. They are popular in multi-processors [83] since they enable better hardware utilisation. Each module is constantly executing in this architecture and does not have execution bubbles (or inactivity).

For a pipeline execution can be achieved by modifying the code presented in the **Serial design** and adding the `PIPELINE` directive:

```
void MyFunction (int a, int b, int c, inc &d) {
#pragma HLS pipeline
   int temp1;
   int temp2;

   module1(a, b, temp1);
   module2(a, c, temp2);
   module3(temp1, temp2, d);
}
```

The code presented above will behave as illustrated in Figure 2.6. The modules do not need to wait until their predecessors finish to execute. Instead, the modules wait until finishing an iteration of themselves.

Furthermore, pipeline designs may imply adding additional optimisations to achieve better performance. For instance, a for-loop pipeline often implies unrolling inner loops to reduce latency (done automatically). For unrolling, the memory access must be concurrent to avoid bottlenecks and high latencies. Consequently, the `ARRAY PARTITION` directive should be used.

On the other hand, false dependencies could prevent the synthesiser from pipelining a code block. These are often removed by explicitly telling it that the dependency does not exist, adding the `DEPENDENCE` directive [82].

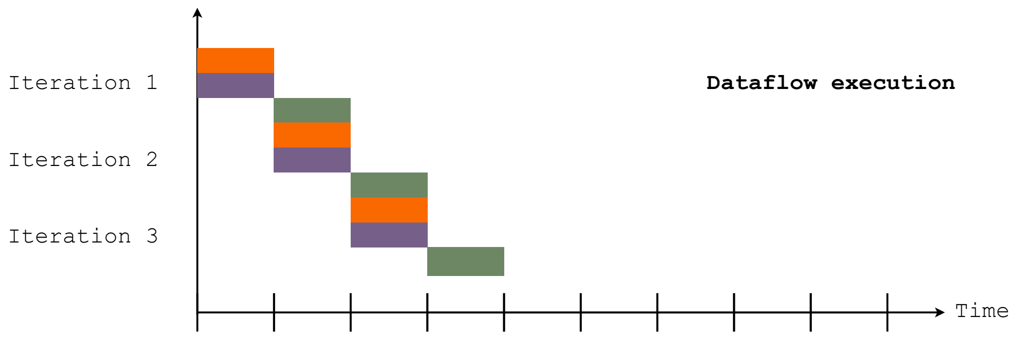


Figure 2.7: Dataflow architecture illustration. In this case, module 1 (orange) and module 2 (purple) can execute concurrently. However, module 3 (green) requires both modules to have their result ready to start.

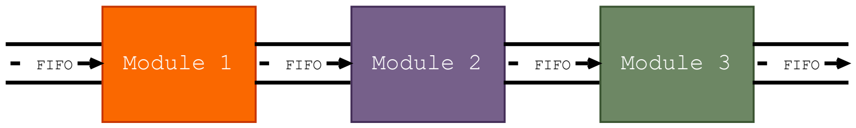


Figure 2.8: The dataflow design implies adding FIFO interfaces to communicate each module with its successor. It ensures efficient data transmission with low latency.

Dataflow Design

The *dataflow design* is a data-driven architecture that focuses on how the data flows within the design. The idea of dataflow is the following: if one datum is ready in module 1, module 2 can take it to start the processing. The dataflow design is also called task-level parallelism since it focuses on the task at the data level.

Differently from the pipeline design, the time division amongst the modules is more diffused. Modules do not require their predecessors to finish an iteration, as presented in Figure 2.7. At the implementation level, the synthesiser places FIFO interfaces amongst the subsequent modules to pass data once they have finished processing, as illustrated in Figure 2.8 [82].

The change is similar to the pipeline design at the code level. Instead of placing `PIPELINE`, the code requires the `DATAFLOW` directive, as illustrated in Figure 2.7.

```
void MyFunction (int a, int b, int c, inc &d) {
#pragma HLS dataflow
  int temp1;
  int temp2;

  module1(a, b, temp1);
  module2(a, c, temp2);
  module3(temp1, temp2, d);
}
```

However, some factors prevent the synthesiser from implementing a dataflow design:

- The variables, arrays or streams cannot be static.
- The data must flow in a forward fashion without feedback.
- There should not be data dependencies: read by one iteration and written by another.

Dataflow designs are often known to be low latency due to data-centrism and do not need data batches to be efficient [82].

Array Partitioning

Despite *array partitioning* is not a design architecture, it is relevant to understand how massive parallelism is possible in FPGAs. Please, consider the following example:

```
void MyVectorProcessor (int a[N], int b[N], int c[N]) {
  for (int i = 0; i < N; ++i) {
#pragma HLS unroll
    c[i] = a[i] * b[i];
  }
}
```

In this case, `a`, `b`, and `c` are arrays with `N` elements. By default, the synthesis tool implements arrays as single-port memories, which are limited to delivering one datum per cycle and will restrict the synthesiser to parallelise the for-loop. Depending on the implementation, the for-loop cannot be unrolled because of the memory access.

Array partitioning is a tool to fragment arrays into smaller arrays (even arrays of a single element). In the example presented above, full unrolling is possible when fragmenting the arrays into single-register elements (arrays of one element), as follows:

```
void MyVectorProcessor (int a[N], int b[N], int c[N]) {
#pragma HLS array_partition variable=a dim=0 complete
#pragma HLS array_partition variable=b dim=0 complete
#pragma HLS array_partition variable=c dim=0 complete

  for (int i = 0; i < N; ++i) {
#pragma HLS unroll
    c[i] = a[i] * b[i];
  }
}
```

The `ARRAY PARTITION` directive specifies the variable, the dimension to partition and the type of partition [82]:

- Block: divides a large array into small contiguous arrays.
- Cyclic: divides a large array into small interleaved arrays.
- Complete: divides a large array into registers.

In the example, partitioning completely leads to a solution with N multipliers, operating the entire loop in a single cycle.

2.5 FPGA-specific Work

DLI has not been limited to CPUs, GPUs or ASAs. There is research on executing DLI on FPGAs and ways to improve the hardware implementation on these devices.

2.5.1 General Optimisations

DNNs are generally resilient to numerical errors, depending on the stage and their applications' criticality. There is an emphasis on the approximation opportunities to optimise and compress neural networks accepting some errors and inaccuracies, assuming that those errors are noise [22]. This fact opens a variety of opportunities for approximate computing in non-critical applications.

Some classical optimisation techniques are widely applied independently from the accelerator architecture but take into account the numerical capabilities. Models are classically trained in FP32 representation, which consumes 32 bits. Nevertheless, these models are numerically over-dimensioned and can be represented using 8-bit integer numbers, which consume less area, are faster, and consume four times less memory. This technique is called *quantisation* [23] and implies two consequences: (1) memory and (2) power consumption.

There are proposals on non-conventional quantisation based on base-2 logarithm, which compressed even better the weights [20]. The authors used 3-bit fixed-point numbers to represent the weights and defined the product (2.22) and additions (2.23) as binary operations. For validation, they compare their work using AlexNet and VGG16, leading to a 1.4% accuracy loss concerning FP32.

$$w^T x \simeq \sum_{i=1}^{n} \mathrm{BitShift}(1, \tilde{w}_i + \tilde{x}_i) \tag{2.22}$$

$$\tilde{s}_2 \simeq \max(\tilde{p}_1, \tilde{p}_2) + \mathrm{BitShift}(1, -|\tilde{p}_1 - \tilde{p}_2|) \tag{2.23}$$

During the training, models also tend to have weights that contribute little to the final output. These weights can be removed from the model by using *pruning*, which consists

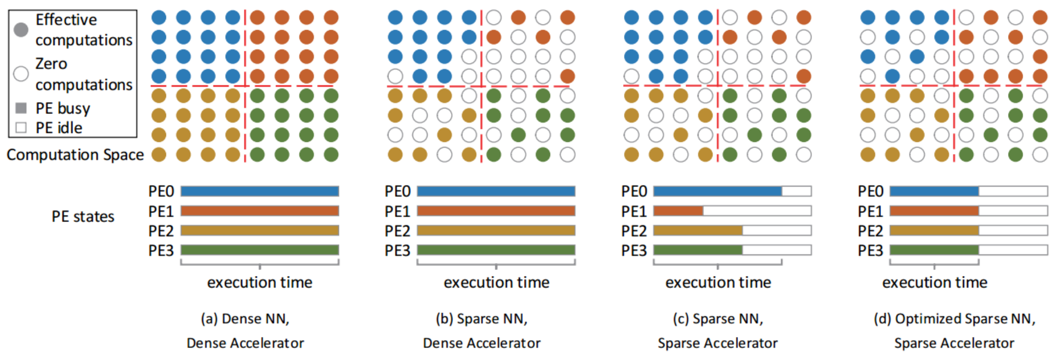

Effective computations
Zero computations
PE busy
PE idle
Computation Space
PE states
PE0
PE1
PE2
PE3
execution time
(a) Dense NN, Dense Accelerator
(b) Sparse NN, Dense Accelerator
(c) Sparse NN, Sparse Accelerator
(d) Optimized Sparse NN, Sparse Accelerator

Figure 2.9: Execution time comparison between dense and sparse accelerators while running several configurations of pruned networks. Balancing the load between the PEs can achieve better utilisation of the resources. Taken from [21].

of identifying those weights which are usually zero-ed or are below a threshold. Chang, Pan, et al. propose extending this technique to prune filters from a CNN and remove negligible weights from the filter [22]. However, pruning without control can lead to resource under-utilisation in some accelerators. Li & Louri analyse this issue and propose an accelerator-aware pruning technique in [21]. They explain that having unstructured pruning makes the accelerator take more clock cycles than needed. Hence, finding a balance can lower the time needed for computation, making the pruning aware of the model's accelerator. Likewise, they study their solution with SCNN, which models pruned CNNs using sparse matrices, which are more efficient in representing highly pruned models. Figure 2.9 compares several cases while running dense and sparse accelerators on unstructured pruning and an accelerator-aware one.

2.5.2 Arbitrary Precision

Most of the methods apply to models which are mapped to FPGA. However, there are more exploitable optimisation opportunities thanks to the hardware representation's flexibility. If the workflow uses HLS, it can perform optimisations at the operations level, such as pipelining and memory rearrangement. Moreover, there are more opportunities to enhance the already presented techniques.

HLS also offers arbitrary precision number representation. Thus, the quantisation is no longer limited to FP32 or 8-bit integer representation. It leads to a broader range of representations that can be tested to achieve better results regarding accuracy loss and power consumption. The quantisation is also not limited to being only linear. For example, the 3-bit logarithmic representation [20] can be represented better on FPGAs than on CPUs, where there is a waste of resources since the minimum register length is 8 bits. Thus, leveraging this optimisation to FPGA can achieve less area occupation and, therefore, less power consumption, leading to more parallelism opportunities.

Taking advantage of this capability, Froehlich et al. in [84] introduce arbitrary precision to LeFlow [85], a tool that joins: Google XLA, to compile the model from Python to LLVM intermediate representation, and LegUp [86], to synthesise C code to RTL.

2.5.3 Tools for Converting Models into HLS

Following the work by Froehlich, there is research on reducing the complexity of the DLI implementation from the user's perspective. This type of contribution often receives a deep learning model, analyses it, and starts a compilation process to determine the layers, the quantisation layer adaptors, and others until getting an FPGA design. Apart from work by Froehlich, two popular frameworks work in this fashion for scientific research: hls4ml [24] and FINN [26]. These contributions are explained in the following sessions.

One of the key advantages of using FPGAs as hardware accelerators is hardware reconfigurability. The developers can easily modify the accelerator's microarchitecture by importing IP cores, which can perform common operations such as multiplications and additions to more complex tasks such as convolutions and dot products. Froehlich et al. have also explored this area in their research. Figure 2.10 shows the workflow followed by ASNet. After the model compilation from Python to LLVM IR using XLA, ASNet proposes a layer to introduce approximations using numerical precision changes and inject hardware units, such as custom adders and multipliers. After that, the approximated LLVM IR artefact is synthesised into Verilog using a modified version of LegUp, which supports custom IP Cores' introduction. Froehlich obtained a reduction of 24.66% in logic units while increasing the maximum frequency by 35.6% with a negligible accuracy loss ($< 2\%$).

2.5.4 Computation Rearrangement

Colleman, Verhelst & Member worked on rearranging the computation of the CNN mapping on FPGAs by exploiting the spatial parallelisation [87]. An intuitive approach for a multi-layer neural network is to compute the outputs layer-per-layer and execute the next layer computation until completing the previous outputs. However, this approach can lead to high I/O consumption and communication. Colleman et al. propose a *deep-first* alternative, which computes the outputs as their inputs are ready, without waiting for the whole layer to be ready. They explain that there are multiple strategies for computing the CNN using this alternative, such as line-based or pixel-based (see Figure 2.11). The most effective strategy is the line-based, where a line of pixels is transmitted for computation. The results regarding the I/O reduction between the host and the accelerator, parallelisation opportunities, and area footprint are promising. Likewise, the authors obtained 695 GOP/s with 78-93% accuracy on a ZYNQ Ultrascale+.

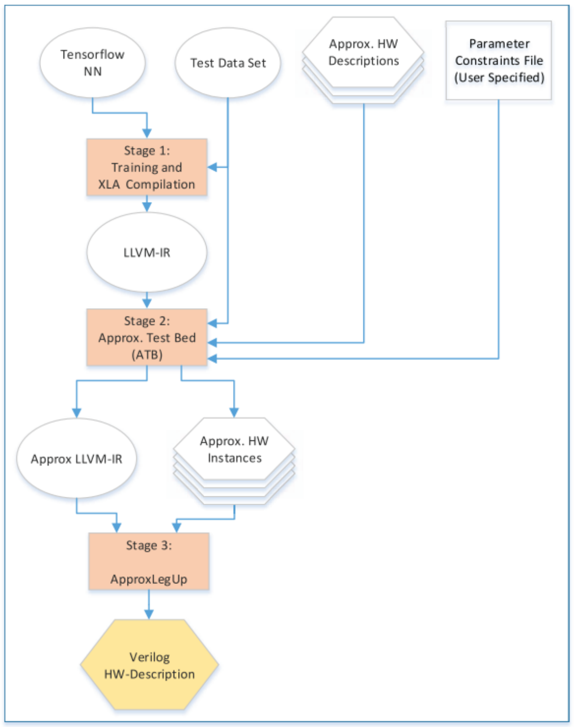


Figure 2.10: ASNet workflow for compiling and synthesising models from TensorFlow to HDL described in Verilog. ASNet takes the artifacts generated by XLA and starts the approximations injection by varying numerical precisions and replacing execution units. Taken from [84].

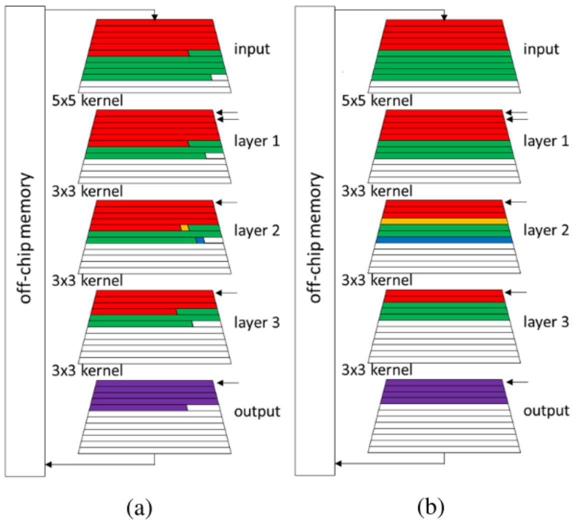


Figure 2.11: (a) Pixel-, (b) line-based deep-first strategies. Each big trapezoid represents a feature layer. Each small trapezoid represents all the channels from one line of activation data of one layer. The colours represent the state: red - out from memory, green - in memory, orange - removed from memory, blue - being computed, purple - exported. The arrow means that the line is padded. Taken from [87].

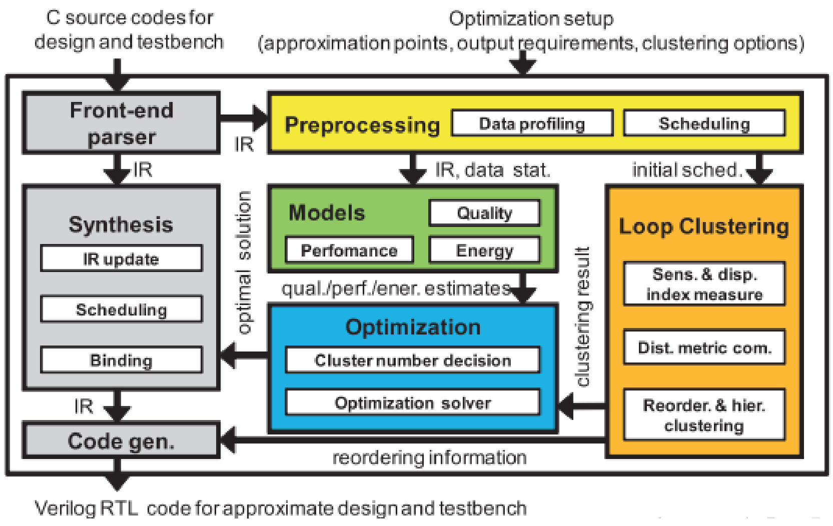


Figure 2.12: Example of an AHLS toolchain for ML. Differently from a conventional HLS, an AHLS introduces approximation points and requirements to the optimisation setup. Additionally, it has an approximation-aware optimiser. Taken from [88].

2.5.5 Approximate High-Level Synthesis roadmap

Zervakis et al. present the challenges and vision for approximate computing for machine learning [88]. They indicate that MAC operations spend 99% of the energy in Deep Neural Networks. It means that approximating and optimising hardware can significantly impact the energy consumption of DNNs, making them both lighter and less power*consuming. They have highlighted that Approximate High-Level Synthesis (HLS) can help achieve promising results in tailoring models to FPGAs for more outstanding performances per Watt. Some tasks which an AHLS toolchain can perform:

- Neural network pruning
- Approximation-aware loop optimisations
- Approximate units and techniques selection while evaluating their impact on the error
- Approximation libraries utilisation, with pre-characterised components in terms of delay, error, and energy models
- Quantisation by novel techniques like log-based discretisation
- Reconfigurable optimisations

Figure 2.12 offers an overview of an AHLS toolchain, which does not differ significantly from a typical HLS-based workflow. The key differences are in the optimisation setup, where the approximation constraints are passed to the toolchain and the incorporation of error evaluation during the design optimisation.

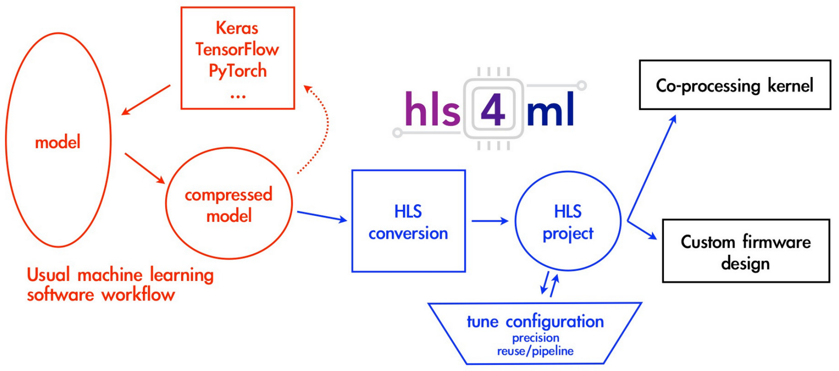


Figure 2.13: *hls4ml* workflow. The process begins with a model which is later compressed. Once it is optimal, the model is converted to hardware through HLS and implemented in the target FPGA. Taken from [89].

2.6 FPGA popular frameworks for DLI

This section presents *hls4ml* and FINN, the most popular machine learning frameworks to perform inference in FPGA devices. CERN supports the first, and the second is supported by academia and AMD/Xilinx (one of the big FPGA vendors).

2.6.1 *hls4ml*

hls4ml framework [89] implements a workflow that receives the model in either TensorFlow or PyTorch. Then, it performs an HLS conversion and creates a project that allows the user to tune the design: establishing optimisation goals and adjusting the numerical precision and the reuse factor. *hls4ml* provides compile-time design tuning, allowing the design synthesis thanks to the use of C++ in HLS, as presented in Figure 2.13. Optimal performance in *hls4ml* is determined by:

- Size and compression of the model: allowing reducing resources.
- Precision: the numerical precision will determine the model's error and how accurate the DLI will be.
- Dataflow and resource reuse: the user can tune the trade-off between resource reuse and performance. Reusing resources imply lower area consumption but more inference time. Instead, investing in more area leads to lower inference time.
- Quantisation-aware training: although it is not part of the library, quantisation-aware training helps to fit the model to the quantisation backend. In this case, *hls4ml* uses QKeras for this step.

Concerning hardware consumption, *hls4ml* employs the *reuse factor* to indicate the replication of the hardware. Figure 2.14 illustrates how the hardware synthesis behaves when tuning the reuse factor.

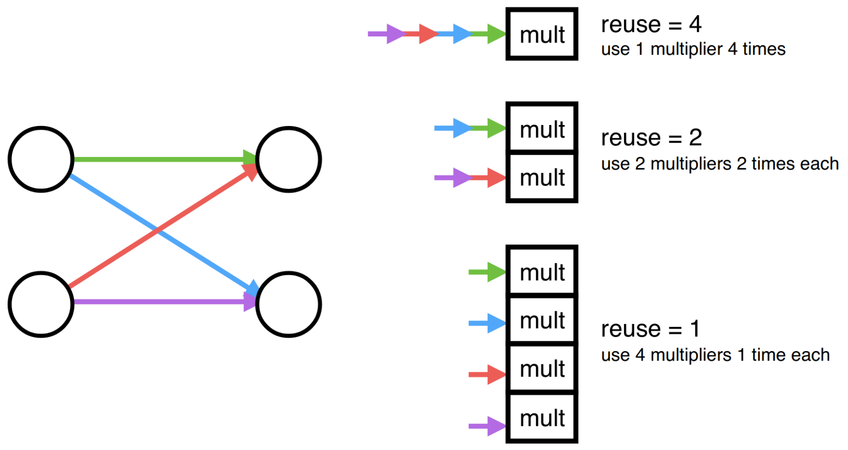


Figure 2.14: *hls4ml* reuse factor effect on the resource usage and execution of the model. Taken from [89].

hls4ml, in its version v0.6.0, supports TensorFlow/Keras in at multi-layer perceptron and convolutional layers [90]. PyTorch and ONNX only have support for multi-layer perceptron layers.

At the HLS level, the implementation is often a dataflow, where each layer is connected to the following. *hls4ml* provides the following example [90]:

```
layer2_t layer2_out[N_LAYER_2];
#pragma HLS ARRAY_PARTITION variable=layer2_out complete dim=0
nnet::dense_latency<configs...>(input_1, layer2_out, w2, b2);

layer3_t layer3_out[N_LAYER_2];
#pragma HLS ARRAY_PARTITION variable=layer3_out complete dim=0
nnet::relu<configs...>(layer2_out, layer3_out);

layer4_t layer4_out[N_LAYER_4];
#pragma HLS ARRAY_PARTITION variable=layer4_out complete dim=0
nnet::dense_latency<configs...>(layer3_out, layer4_out, w4, b4);

nnet::sigmoid<configs...>(layer4_out, layer5_out);
```

The example shows two dense layers (multi-perceptron layer) optimised for latency, an activation function (ReLU) and a sigmoid. This example illustrates how the entire model is implemented into a single accelerator within the FPGA.

In terms of support, *hls4ml* supports the production of synthesisable designs in Intel and Xilinx-based FPGAs through Quartus and Vivado HLS backends.

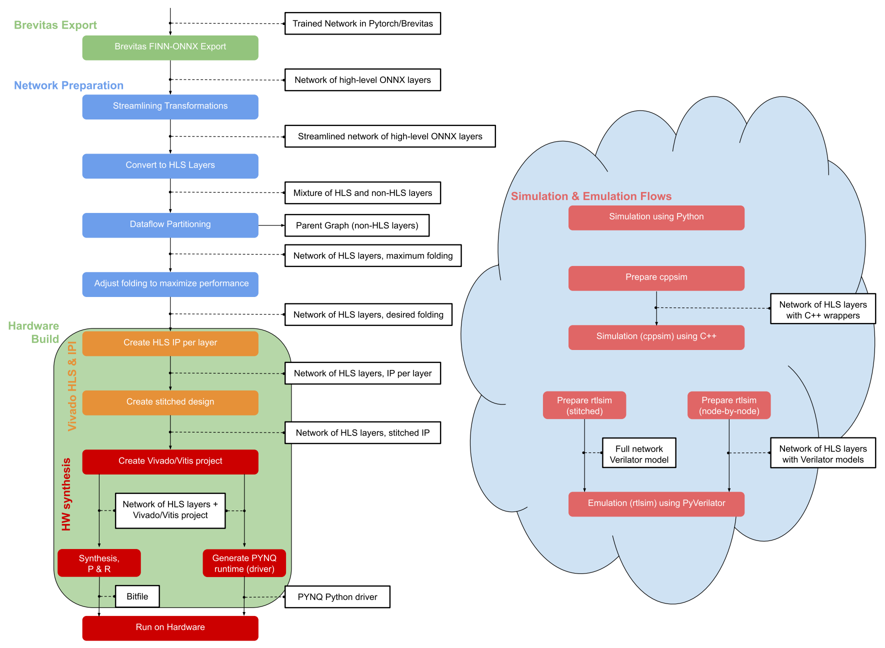

Brevitas Export
Brevitas FINN-ONNX Export
Trained Network in Pytorch/Brevitas
Network Preparation
Network of high-level ONNX layers
Streamlining Transformations
Streamlined network of high-level ONNX layers
Convert to HLS Layers
Mixture of HLS and non-HLS layers
Dataflow Partitioning
Parent Graph (non-HLS layers)
Network of HLS layers, maximum folding
Adjust folding to maximize performance
Network of HLS layers, desired folding
Hardware Build
Vivado HLS & IPI
Create HLS IP per layer
Network of HLS layers, IP per layer
Create stitched design
Network of HLS layers, stitched IP
HW synthesis
Create Vivado/Vitis project
Network of HLS layers + Vivado/Vitis project
Synthesis, P & R
Generate PYNQ runtime (driver)
Bitfile
PYNQ Python driver
Run on Hardware
Simulation & Emulation Flows
Simulation using Python
Prepare cppsim
Network of HLS layers with C++ wrappers
Simulation (cppsim) using C++
Prepare rtlsim (stitched)
Prepare rtlsim (node-by-node)
Full network Verilator model
Network of HLS layers with Verilator models
Emulation (rtlsim) using PyVerilator

Figure 2.15: FINN design flow. The model is exported in a custom ONNX that includes additional layers to support arbitrary quantisation. Then, FINN starts to build an HLS IP per layer with maximum folding for later tuning by the user. After having a tuned design, the IPs are stitched to make a datapath with all the layers. The hardware execution can be based on PYNQ or standalone. Taken from [25].

2.6.2 FINN

FINN is a machine learning framework for DLI of quantised models on FPGAs supported by AMD/Xilinx. Like *hls4ml*, it takes a model from Python and produces an HLS design capable of running on Xilinx FPGAs. However, FINN, different from *hls4ml*, is focused on PyTorch as the main machine learning framework. Moreover, instead of generating a single accelerator, FINN adopts a more granular approach, generating a single accelerator per layer. Each layer can be more optimised than others since the reuse factor (a.k.a. folding factor) applies per layer [25].

This difference between the two frameworks is crucial for leveraging DLI to FPGAs with limited resources. The granular approach adopted by FINN allows tuning the expensive layers and trying higher folding factors for resource recycling.

Figure 2.15 shows the FINN data flow. At the model level, the description and the quantisation are performed thanks to Brevitas, a framework compatible with PyTorch. Then, a series of HLS IPs are generated per layer, favouring a granular optimisation by tuning the folding factor to get a beneficial resource-performance trade. To adjust the folding, the values for PE and SIMD can be increased to increase the performance.

The resulting IPs are connected in a daisy chain fashion to get a dataflow path and exported to either a hardware bit file or a PYNQ project [26], making the whole process user-friendly through Python.

FINN also supports software simulation of the whole datapath thanks to the HLS simulation. It is helpful for design verification and when calculating the folding factor of each IP and seeing how the final results behave in terms of performance and resource consumption. Besides, the simulation can happen at the Python level, HLS code and Verilog, offering a complete verification of the design in its several stages.

2.6.3 Challenges and Opportunities

Both solutions were conceived for high-end FPGAs and cloud facilities, making them lack granularity in the design and limiting the possibility of implementing designs in low-end FPGAs. However, the optimisations are still closed for research in optimisation tasks and approximate computing. FINN addresses this issue by splitting the accelerator into IPs per layer.

Another challenge is the integration of approximate units. FINN and *hls4ml* support fixed-point arithmetic, but their execution elements are still exact and do not integrate any approximation beyond quantisations.

Hence, both frameworks are powerful and popular in DLI on FPGAs. They still have some open challenges and opportunities to improve. For instance:

- Granularity: both approaches implement the units of the whole model into the FPGA. There is no chance to distribute the computation between the FPGA and a host CPU for a hybrid execution.
- Approximate computing: both frameworks are exact in their computations. It opens the chance to experiment with approximate computing techniques.
- Model size: the designs produced by both frameworks can still be unsuitable for low-end FPGAs like the Artix-7 in its most minor configurations. The combination of the first two challenges can lead to an improvement in this issue.
- Opening to other applications: both frameworks are highly tailored to DLI and restrict their usage in other applications such as Linear Algebra. A more generic approach could satisfy the DLI and other fields as well.

Chapter 3

Design of Customisable Processing Elements

This chapter explains the PE design process for matrix operations (including matrix multiplication and addition) and convolution. The design includes the analysis of the mathematical operation to resolve, possible architectures, and the differences with other alternatives.

3.1 Design Goals

From the challenges exposed in Section 2.6.3, the PE is crucial to addressing the accelerator's granularity, approximation support and implementation size. The PE, in this work, is a small computing unit in charge of calculating operations for a small matrix, i.e., matrix multiply-add, convolution, and activation functions. Thus, the size of the PE will determine how many PEs can fit into an accelerator given an FPGA fabric. Moreover, the PE provides the approximation capability of the accelerator.

The design goals set for the design of the PEs are:

- **Easy data type swapping**: the PEs shall allow the support of multiple data types, i.e., floating-point, fixed-point, or integer. This support can be done through a C++ template parameter.
- **Configurable operand size**: varying the input/output size makes it possible to adjust the PE size and define how many PEs can fit into an accelerator.
- **Stackable blocks**: the accelerator shall support placing one or more PEs seamlessly. It means that adding more PEs shall not be problematic or require additional developer logic.

Table 3.1: Comparison between the most popular FPGA ML libraries and this work - PE level

Criterion	**hls4ml**	**FINN**	**FAL** (this work)
Easy data type swapping	✓	✓	✓
Configurable operand size	Accel	Accel	✓
Stackable blocks	Accel	✓	✓
Open for approximations	✗	✗	✓
Replaceable	Accel	Accel	✓

- **Open for approximations at the operator level**: approximate adders, multipliers and non-linear functions can be supported by the PE. Thus, by modifying a parameter will be possible to replace an exact adder with an approximate one.
- **Replaceable**: different implementations of the PE shall have a standard interface allowing changing the PE at the accelerator's level.

From the goals listed above, the design is focused on offering degrees of freedom for modifying the PE size in exchange for numerical errors (approximations) and exploring PE implementations. Moreover, this research is interested in the ease of use for creating a vector accelerator and replacing the unit for design space exploration.

This work's approach is more comprehensive than most popular alternatives, such as *hls4ml* and FINN, regarding granularity, design control of the resource consumption, and approximation capability. Table 3.1 illustrates the design goals accomplishment amongst the most popular FPGA ML frameworks and this work. *Accel* accomplishment refers that the support is only available at the accelerator level. This level of abstraction implies the computation of a whole layer, being less granular for dealing with the model size unless altering the *reuse* (or *folding*) factor.

Another interesting feature of accomplishing the design goals stated in this section is the possibility of reusing the PEs for purposes other than DLI.

3.2 Generic Matrix Multiply-Addition (GEMMA)

GEMMA is one of the most common operations in DLI, and it is widely used in dense layers (or multi-perceptron layers) [42]. For instance, in the case of the LeNet-5 [36], it is composed of three dense layers that compute GEMMAs of

- $A : \mathbb{R}^{256\times b}, B : \mathbb{R}^{120\times 256}, C : \mathbb{R}^{120\times b}$
- $A : \mathbb{R}^{120\times b}, B : \mathbb{R}^{84\times 120}, C : \mathbb{R}^{84\times b}$

- $A : \mathbb{R}^{84\times b}, B : \mathbb{R}^{10\times 84}, C : \mathbb{R}^{10\times b}$

where A, B, and C are the matrix operands such that $D = A \times B + C$ composes the GEMMA operation. In total, the LeNet-5 model has 44426 parameters where 41854 participate in multiply-adds computed in matrix multiply-add.

In this work, the goal is to define a generic matrix multiply-add whose arithmetic operators (i.e., multipliers and adders) can be replaced by approximated versions that use fixed-point numbers with changeable bit-lengths, with a parameterised number of rows and columns. A generic matrix multiply-add can be represented as

$$Y_{ij} = f(\mathcal{S}_{k=1}^{N}\{\mathcal{S}\{\mathcal{M}\{X_{ik}, W_{kj}\}, B_k\}\}) \tag{3.1}$$

where $\mathcal{S}\{\cdot\}$ is a single adder operator, $\mathcal{M}\{\cdot\}$ is a single multiplier, $f(\cdot)$ is the activation function, and N is the number of columns of the input matrix and the rows of the weights matrix. A naïve C++ implementation of the matrix multiply-add can be:

Algorithm 1 Baseline Matrix Multiply-Add, `+` and `*` are functors

```
rows:
  for(int n = 0; n < N; n++) {
cols:
    for(int m = 0; m < N; m++) {
accumulation:
      d[n][m] = c[n][m];
      for(int k = 0; k < N; k++) {
        d[n][m] += a[k][m] * b[n][k];
      }
    }
  }
```

where there are three for-loops for addressing the rows, columns and accumulation. The input matrices are stored in the 2-D arrays a, b, c and the output is the d array. From this naïve approach, it is possible to derive multiple architectures:

- *baseline*, a serial implementation that may require at least N^3 clock cycles;
- *pipeline*, which pipelines the rows and unrolls the inner loops, requiring partitioning a and b completely, and c and d in its columns (utilising N memory blocks), consuming $N + 2$ cycles; and
- *single-cycle*, which performs the operations in a single-cycle by unrolling all loops.

There are other possible designs but this work only takes the mentioned above for simplicity.

On the other hand, typical multiplier implementations save the result in a register bigger than the operands to avoid overflow issues [83]. The output register is twice the width of the two input operands, while the accumulation is at least one bit bigger. This work's multipliers and adders keep the output operands' width consistent with the data throughput and reduce area consumption. It is similar to the method used in lossy applications [91], which saves area by dropping the bits. The proposal to manage the overflow is based on scaling two of the multiply-add operands as

$$d' = \frac{ab}{2N} + \frac{c}{2N} \tag{3.2}$$

where a, b, c are the multiply-add operands, $d = 2Nd'$ is the output, and N is the number of rows considering a $N \times N$ squared matrix. One caveat of scaling with respect to the number of rows (or columns) is that the numerical error of the operation becomes dependent on the matrix size. The following sections demonstrate how the matrix size impacts the overall error.

Considering the characteristics of the matrix multiply-add described during this section, the construction is similar to the one presented in Algorithm 2. The PE requires a datatype definition and the approximate operators (even though, it falls back to the exact version if not defined). The data type is defined as a fixed-point of $W = 16$ bits with $I = 1$ bit of integer. The operand size is $R \times C = 2 \times 2$. The arithmetic operators are an approximated version, dropping the $D = 4$ LSBs of the operands. Once all these elements have been defined, the PE is specialised and can be utilised after its construction.

3.3 Window-based Convolution

Convolutions are popular in image processing with neural networks [30]. In the case of LeNet-5, it has two convolutions with kernels of 5×5, responsible for 2572 parameters out of 44426 total. Moreover, following the matrix multiply-addition, the convolution engine is parameterised in the input and output dimensions, the datatype, and the arithmetic operators, making it possible to use approximate computing techniques to better use the resources compared to using standard datatypes and exact arithmetic.

The computation of a single pixel using spatial convolution is represented as:

$$Y_{ij} = \mathcal{S}^{\kappa}_{n=-\kappa}\{\mathcal{S}^{\kappa}_{m=-\kappa}\{\mathcal{M}\{X_{i+n,j+m}, K_{mn}\}\}\} \tag{3.3}$$

where K_{mn} is the m-th row and n-th column of the kernel $\mathbf{K}$, and $\kappa = \lfloor N_K/2 \rfloor$ is defined as the half of the kernel size $N_K = 1, 3, \ldots, 2k-1$. For instance, for a kernel size $N_K = 3$, the indices based on κ would be from -1 to 1.

Algorithm 2 Matrix multiply-add C++ description

```
/* Matrix dimensions */
static constexpr int R = 2;
static constexpr int C = 2;

/* Define data types */
static constexpr int I = 1;
static constexpr int W = 16;
static constexpr int D = 4;
using DataType = ap_fixed<W, I>;

/* Define approximate operators */
using Multiplier =
    axc::arithmetic::lsbdrop::Multiply<DataType, W, I, D>;
using Adder =
    axc::arithmetic::lsbdrop::Add<DataType, W, I, D>;

/* Specialise the GEMMA PE */
auto engine =
    ama::hw::operators::MatrixMultiplyAdd<DataType, R, C,
        Adder, Multiplier>{};

/* Use the PE over matrices MA, MB, MC, MD */
engine.Execute(MA, MB, MC, MD);
```

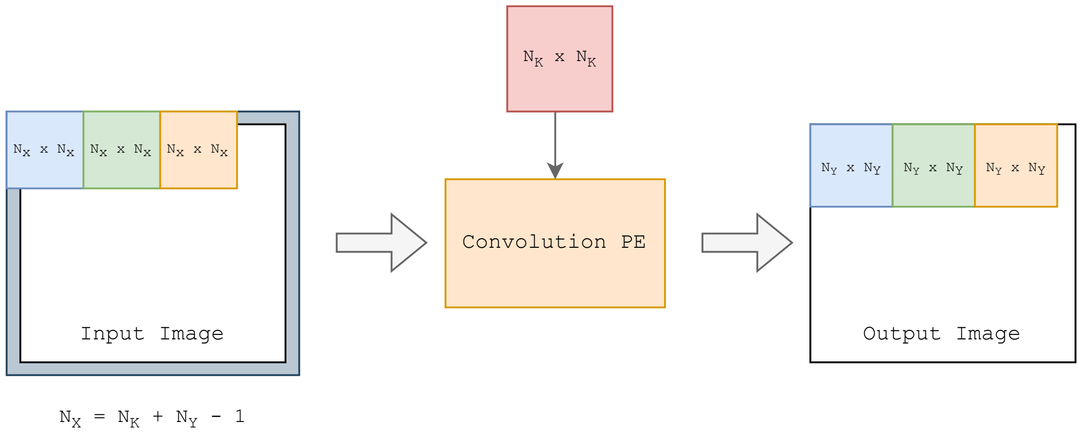


Figure 3.1: Window convolution principle. The convolution PE computes windows of the image, leading to greater parallelism at the PE level.

For adding more computing power to the PEs, each unit computes a *window* of pixels instead of a single pixel, as Figure 3.1 illustrates. The output window size is also configurable, increasing the parallelism at the PE level and allowing the computation of multiple pixels while multiple inputs arrive at the PE. From now on, the window-based convolution based on Equation (3.3) will be called *Window-Based Spatial Convolution*. A naïve implementation of the window-based convolution in the space domain can be represented as:

Algorithm 3 Baseline Spatial Convolution

```
output_rows:
  for(int oy = 0; oy < NY; oy++) {
output_cols:
    for(int ox = 0; ox < NY; ox++) {
      y[oy][ox] = 0;
kernel_rows:
      for (int n = 0; n < NK; n++) {
kernel_cols:
        for (int m = 0; m < NK; m++) {
          y[oy][ox] += x[oy + n - NK / 2][ox + m - NK / 2] * k[n][m];
        }
      }
    }
  }
```

where the output matrix is `y` of size $N_Y \times N_Y$, the kernel matrix is `k` of size $N_K \times N_K$, and the input matrix is `x` of size $N_X \times N_X$, with $N_X = N_K + N_Y - 1$. Both N_K and N_Y are parameters of a PE.

The algorithm for the convolution consists of an element-wise product followed by the sum of the products, leading to the pixel computation loops and the output window computation. Based on these two parts of the spatial convolution algorithm, the following implementations are possible:

- *baseline*, a serial implementation that may require at least $N_K^2 + N_Y^2$ clock cycles, N_K^2 for the products, and N_Y^2 for exploring the output matrix;
- *serial output + unrolled kernel loops*, an attractive architecture can be composed by the merge of the output loops serially and the unrolling of the kernel loops, leading to a single execution unit for the pixel computation and requiring partitioning k (the kernel) and x (the input) completely. It consumes, at least, N_Y^2 clock cycles; and
- *single-cycle*, which performs the operations in a single-cycle by unrolling all loops, requiring all arrays to be partitioned completely.

This work also involves the Winograd convolution implementation, illustrated in Figure 3.2. In this case, the Winograd PE targets a $N_K = 3 \times 3$ kernel and a 2×2 output

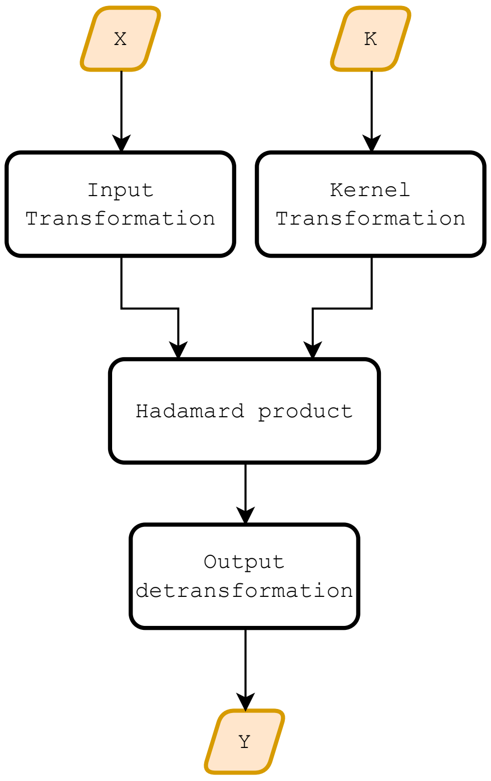


Figure 3.2: Winograd convolution algorithm. Similar to other domain transformations, it requires the inputs to be transformed and the output to be transformed back to space.

window ($N_Y \times N_Y$ for simplicity), making it more specific than the former convolution technique. The Winograd operations are implemented discretely without loops (loop unrolling + algebraic simplifications). For bigger kernels, it would imply using a for-loop-based matrix multiplication function for computing the transformation, given that discretising the operations becomes unreadable and impractical. Moreover, the intermediate results are stored in matrices whose entries occupy twice the bits of the input/output matrix entries.

Given that the algorithm has multiple stages, it is a good candidate for pipelining and dataflow. Thus, the possible implementations are:

- *baseline*, a serial implementation that may require at least 4 clock cycles when having all the inputs and outputs partitioned completely. Otherwise, it may require more than $N_X^2 + 4$ clock cycles to access the inputs and perform the processing;
- *pipeline*, where each algorithm step is performed in about one clock cycle. It may require partitioning all the inputs and outputs for one-clock cycle access and streaming for continuous processing; and,
- *single-cycle*, which performs the operations in a single cycle by inlining all the functions and partitioning the inputs and outputs completely in single registers.

Like the GEMMA PE, the convolution can process arbitrary fixed-point datatypes, adjusting the operand size and utilising approximate arithmetic operators. For example, Algorithm 4 describes a Winograd convolution that operates a matrix of $O = 2$ and a kernel of $K = 3$. The data type is a $W = 16$ fixed-point with $I = 1$ integer bits. The operands drop the $D = 4$ LSBs. Once defined all these requirements, the PE is specialised, constructed and used.

Algorithm 4 Convolution C++ description

```
/* Matrix dimensions */
static constexpr int O = 2;
static constexpr int K = 3;

/* Define data types */
static constexpr int I = 1;
static constexpr int W = 16;
static constexpr int D = 4;
using DataType = ap_fixed<W, I>;

/* Define approximate operators */
using Multiplier =
    axc::arithmetic::lsbdrop::Multiply<DataType, W, I, D>;
using Adder =
    axc::arithmetic::lsbdrop::Add<DataType, W, I, D>;

/* Specialise the GEMMA PE */
auto engine = ama::hw::convolvers::Winograd<DataType, K, O,
     Adder, Multiplier>{};

/* Use the PE over image IX with kernel IK with output OY */
engine.Execute(IX, IK, OY);
```

3.4 Vectorisation Capability

Addressing the challenge of granularity implies answering the question of how to scale the PEs to large accelerators. Vectorisation is one of the alternatives to concatenate several PEs to increase the processing power while keeping the execution time constant. It also implies spending resources in the same proportion as the processing power. It achieves the goal of **Stackable Blocks** set at the beginning of this chapter.

This work presents a vectorisation wrapper class that allows replicating PEs within an accelerator. It allows concurrent execution that requires more resources but preserves the

PE's execution time, i.e., having a single PE can lead to an execution time of 10 clocks. In contrast, five PEs also lead to 10 clocks, preserving the execution time and providing five times more performance at the computation level. It is similar to the functionality of the vector units in a central processing unit, giving a similar to Single-Instruction Multiple-Data (SIMD) functionality [83]. From the related work, FINN has a similar capability [25].

The implementation is through template recursion, as presented in Algorithm 10 (Appendix A). In this case, the algorithm presents the example for the convolution wrapper, where the template parameters are `N` (the current iteration of the recursion), `NT` (total number of instances), and `ENGINE` (PE specialisation ready to construct). The wrapper receives an array of rows multiplicated by the number of instances (lines 6 and 11). Then, the rows are grouped and distributed amongst the PE instances (lines 19 and 20) for execution in an iteration-constructed PE instance (line 17). Later, the recursion continues by reconstructing the wrapper until getting `N = 0`.

Algorithm 5 Vectorisation template use case: Matrix Multiply-Add

```
GEMMA(const A, const B, const C, D)
    # select a PE core for $R \times C$ float matrices
    pe = operators::MatrixMultiplyAdd<Float16, R, C>;
    # vectorise pe with $N$ PEs
    Vectorise<N, N, pe>::Execute(A, B, C, D);
```

The vector accelerator presented in Algorithm 5 shows the case for a multiply-add PE specialised in 16-bit floating-point $R \times C$ matrices, which is replicated N times. Figure 3.3 shows how the design looks after the synthesis. The data is assumed as vectorised, and the matrices are distributed to the multiple PEs. In this case, the matrices are represented in planar row-major and copied to a register bank within the vector unit.

Algorithm 6 Vectorisation template use case: convolution

```
CONV(const IN, const KERNEL, OUT)
    # select a Winograd PE core for a $N_K^2$ kernel and $N_Y^2$ output matrix
    pe = convolvers::Winograd<FxP<16,1>, N_K, N_Y>{};
    # vectorise pe with $N$ PEs
    Vectorise<N, N, pe>::Execute(IN, KERNEL, OUT);
```

For the convolution case, Algorithm 6 shows the vectorisation of a Winograd PE specialised in 16-bit fixed-point with 1-bit integer number representation for $N_K \times N_K$ kernels and $N_O \times N_O$ output matrices. Figure 3.4 illustrates how the accelerator looks when containing a vector convolution. The system feeds the vector using windows ($N_X \times N_X$, where $N_X = N_K + N_O - 1$) which do not share rows. This is conveniently designed given that the memory layout in the C language is row-major, and the addressing of

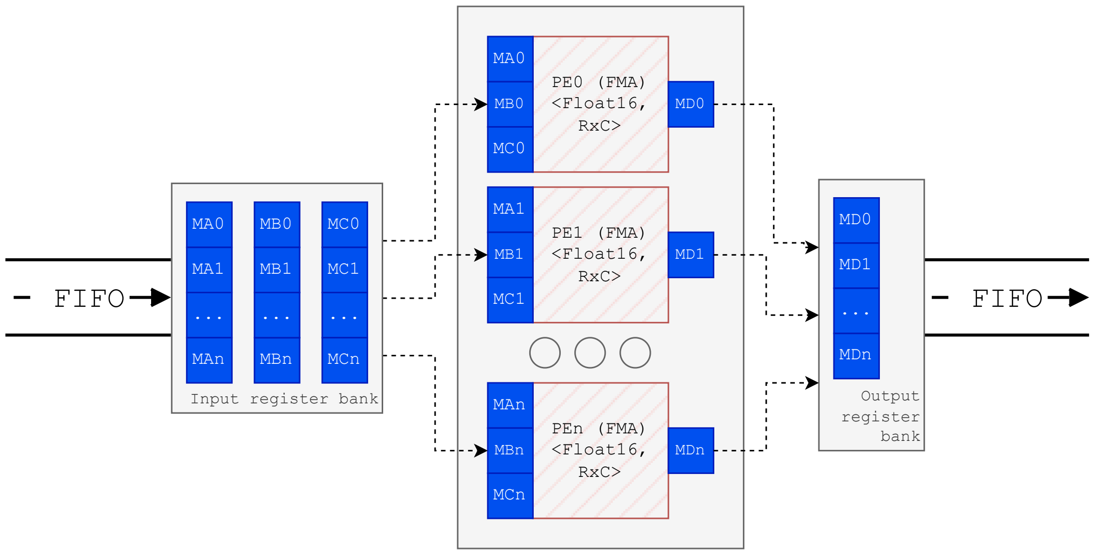


Figure 3.3: Vectorisation block diagram for the matrix multiply-add. The wrapper replicates the PE N times and distributes the vector of data to multiple instances. For this purpose, it is assumed that the matrices are stored in row-major, the interface to the host is through a FIFO-like protocol, and the data are stored in a register bank for concurrent access (complete array partitioning).

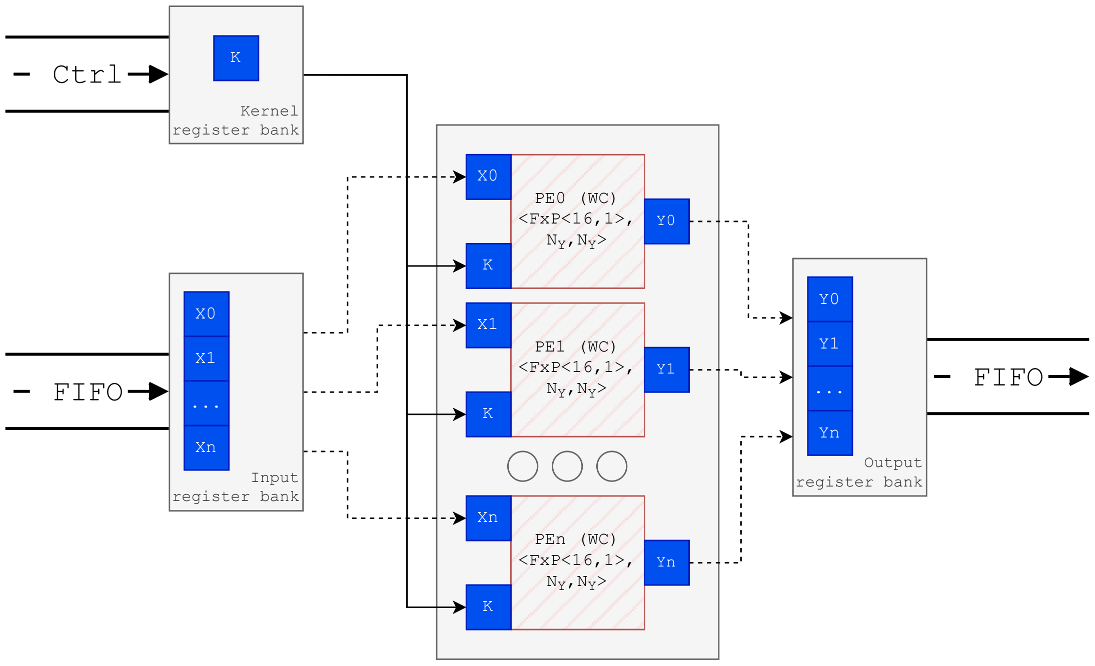

Ctrl
K
Kernel register bank
FIFO
X0
X1
...
Xn
Input register bank
PE0 (WC) <FxP<16,1>, N_Y,N_Y>
PE1 (WC) <FxP<16,1>, N_Y,N_Y>
PEn (WC) <FxP<16,1>, N_Y,N_Y>
Y0
Y1
...
Yn
Output register bank
FIFO

Figure 3.4: Vectorisation block diagram for a convolution. The wrapper replicates the PE and distributes the vector of data to multiple instances. In the convolution case, all the PE instances share the kernel and convolve distinct parts of the image. For this purpose, it is assumed that the images are stored in row-major, the interface to the host is through a FIFO-like protocol, and the data are stored in a register bank for concurrent access (complete array partitioning).

two-dimensional arrays takes the rows as the outer dimension. So, the data distribution within the accelerator can be distributed as

$$\mathbf{X}^{(p)} = \mathbf{X}[N_X \times p][N_X] \tag{3.4}$$

where $\mathbf{X}^{(p)}$ is the submatrix sent to the p-th PE, and N_X is the number of rows and columns of the submatrix. This way, the columns are contiguous in memory, and the copies can be vectorised.

3.5 Summary

A complete algorithm for an accelerator can highlight the achievement of the design goals stated in the beginning of this chapter. Recalling Algorithm 4, it is possible to configure a convolution PE by specifying the data type, the arithmetic operators, and the operands' size. It proved the achievement of the three first design goals from the list presented above.

Algorithm 11 (Appendix B) shows an example of a complete execution unit with replacement and vectorisation. Lines 1-9, and 15-18 are the same as Algorithm 4, where the data type, the operands' size and the arithmetic operators are defined. These lines prove the achievement of **easy data type swapping**, **configurable operand size**, and **open for approximations at the operator level**. Lines 21-27 shows how to replace PE implementations. Through a compilation-time flag, it is possible to choose between a Winograd or a Spatial convolution. If the flag `USE_WINOGRAD` is defined, the Winograd PE will be synthesised, otherwise the Spatial will. This is heavily exploded by the framework for design space exploration. It makes this work achieve the **replaceable** goal. Line 12 specifies the number of cores wanted in the vector unit. In this case, the final unit will have 4 PEs. Lines 31-32 use this constant to replicate the engine (or PE specialisation) from Lines 21-27 4 times using Algorithm 10 and execute the convolution. The operands, differently from Algorithm 4, have an input size of $4N \times 4$ (recalling that $N_X = N_Y + N_K - 1$), kernel size of 3×3 and output size of $2N \times 2$, which are distributed as illustrated in Figure 3.4. With this, the **stackcable blocks** goal is achieved, completing the design goals proposed in the beginning of this chapter.

The summary of techniques utilised for accomplishing these goals is presented below:

- **Easy data type swapping**: addressed by setting the datatype through C++ templates (as a parameter).
- **Configurable operand size**: also addressed by setting the operands' size through C++ templates (as a parameter).
- **Open for approximations at the operator level**: passing the operands as functors, including the exponentials, allows approximation opening within the PEs.

- **Stackable blocks**: the vectorisation wrapper allows stacking multiple PE instances through template recursion. It is implemented as a template class that wraps a PE specialisation.
- **Replaceable**: keeping the interface in terms of size, memory requirements and protocol helps to achieve replacement of each PE when possible.

Chapter 4

Design of Template Accelerators

This chapter explains the accelerator design process, the next level of abstraction after the PE design process. This process involves the integration of the PEs, communication protocols and interfaces to the host (the ARM microprocessor in the case of an FPGA-based SoC), the explored architectures and their analysis, and the differences with other approaches.

4.1 Design Goals

Similarly to Chapter 3, the development of the accelerators looks forward to addressing the challenges exposed in Section 2.6.3. So far, the PE design considers the **Approximate Computing** opening and the **Granularity** principle, being modular, stackable and replaceable. However, the model characteristics and the opening for other applications still need to be addressed.

In this chapter, the design goals of the accelerators are:

- **Template structure**: the design shall behave like a template. It integrates both a static part and a replaceable one. The static part is in charge of communication, control and caches. In contrast, the replaceable part integrates the execution units based on PEs, making it easy to swap between implementations.
- **Standard interface**: the accelerators must work with a generic driver. The data transmission interface and the control interface shall be similar for all the templates, including possible intersections in control registers.
- **Configurable**: the framework shall adjust the static part of the template according to the operands (data type and size). Moreover, the framework shall configure the dynamic part in terms of the target PE implementation and the number of units required, i.e., choosing between Winograd or Spatial PEs in a convolution accelerator.

Table 4.1: Comparison between the most popular FPGA ML libraries and this work - accelerator level

Criterion	**hls4ml**	**FINN**	**FAL** (this work)
Template structure	✓	✓	✓
Standard interface	✓	✓	✓
Configurable	Partial	Partial	✓
Simulable	✓	✓	✓
Acceleration granularity	✗	✗	✓

- **Simulable**: the accelerator shall be entirely simulated at software level, allowing simulations of applications and DLI.
- **Acceleration granularity**: placing multiple small accelerators rather than a singleton shall be possible, allowing communication-execution overlapping.

Unlike the PE design, the accelerator design is a template with some degrees of freedom: the PE implementation and the number of PEs. Moreover, it focuses on the capacity of simulating the design for a software evaluation, avoiding going through the FPGA workflow from the HLS synthesis up to the bitstream generation, accelerating the DSE process and offering a fast tool to evaluate the designs.

Thanks to High-Level Synthesis and C++, this work can use templates to create generic implementations, parameterised and adapted according to the design requirements. *hls4ml* and FINN already leverage these advantages on Xilinx-based platforms. Another interesting feature of these frameworks is the simulation capability. FINN, in particular, can simulate each stage of the development flow, acquiring results regarding error (or model accuracy) and resource consumption (in implementation stages).

Moreover, this work's approach is versatile in terms of the application's domain. An accelerator is sufficiently generic to run applications other than DLI. For instance, a GEMMA accelerator would benefit Approximate Linear Algebra applications. In addition, it is possible to have multiple accelerators with the same specialisation (or parameters) within the FPGA, permitting overlapping between communication and execution through work balancing. In other words, having two or more accelerators allows one to run either accelerator at different times and balance the loads, dealing with the communication bottlenecks, i.e., if one accelerator is executing, another one can use the communication channel. This leads to another level of granularity at the accelerator level not found in frameworks such as FINN and *hls4ml*.

Another difference with existing frameworks is the optimisation capability. As proposed by this work, the user can optimise the accelerator and PEs through directives, adding a new degree of freedom to the DSE for creating different design architectures and implementations. In FINN and *hls4ml*, this is not possible, given that they have ”flashed” the optimisations within the code.

Table 4.1 summarises the design goals in the most relevant frameworks for DLI on FPGA and this work. Common goals include the **template structure**, **standard interface**, and **simulability**. However, **acceleration granularity** is not fully addressed and the *configurability*, where FINN and *hls4ml* present a *partial* configurability. Both frameworks offer two or more alternatives to the user, mainly focused on resource-saving or latency in the case of *hls4ml* and the folding factor in the case of FINN. However, since there is no solid concept of PE, the number of execution units is unavailable.

By accomplishing all the goals stated above, the acceleration units can be tailored, DSE-friendly, and general-purpose. Moreover, the analysis will be limited to the matrix accelerator (focused on GEMMA) and the convolution accelerator. Activation functions are assumed as integrated into these units.

4.2 Accelerator Template

Two of the design goals are that the accelerator shall behave **template structure** and shall have a **standard interface**. This work proposes a series of macros to describe the accelerators without going into implementation details. The macros declare capabilities, check parameters, and define the accelerator regions. Also, it implements the FIFO-like structures for the AXI4-Stream port and the register implementation for the AXI4-Lite control port.

Algorithm 7 reflects the macros utilised to describe a matrix accelerator. There are, in particular, three kinds of macros:

- DEF_SET_*_CAPABILITIES() declares the capabilities of the accelerator.
- DEF_SET_*_PARAMS_CHECK() performs the logic to check the runtime parameters with respect to the capabilities of the accelerator.
- BEGIN_DEF_TOP_*_FUNCTION() and END_DEF_TOP_*_FUNCTION() defines the implementation of the accelerator. It includes array declarations and connects the static and dynamic modules.

* means that it can be either `MATRIX` or `CONV` for a matrix or a convolution accelerator, respectively.

The first two kinds expand to functions that provide the accelerator’s capabilities and check the parameters set by the user at the execution time, providing runtime **configurability**. The third kind expands to the top module, which defines the accelerator as a whole. The top module receives all the accelerator’s variables and parameters since the implementation can use them. Moreover, it also invokes the capabilities and parameters check functions, redirecting some of the variables to these functions. For example, in Algorithm 7, the variables `en_accumulation`, `rows`, and `columns` are redirected to the parameters check. The template implements these variables at the implementation level as registers accessible through the AXI4-Lite control. Likewise, the top function also accesses the streaming ports for data I/O. In Algorithm 7, they are `stream_input` and `stream_output`.

Algorithm 7 Accelerator file structure. This file will describe the accelerator capabilities, check the parameters at runtime and implement the static and dynamic logic. This structure is called *canonical*.

```
DEF_SET_MATRIX_CAPABILITIES(caps) {
    // Defines the capabilities of the accelerator
    // It matches a control register.
    WRITE_RANGE_CAP(rows, 2, 8);
    WRITE_RANGE_CAP(columns, 2, 2);
}

DEF_SET_MATRIX_PARAMS_CHECK(check) {
    // Checks the runtime parameters against the capabilities.
    CHECK_RANGE_AND_FALLBACK(rows, 2, 8, 2);
    CHECK_RANGE_AND_FALLBACK(columns, 2, 2, 2);

}

BEGIN_DEF_TOP_MATRIX_FUNCTION(accel, caps, check)
accel_top: {
    // Implementation details

    /* Temporal buffers */
    static DataType A[8][2] = {0.f};
    static DataType B[8][2] = {0.f};
    static DataType C[8][2] = {0.f};

    /* Get parameters */
    PropertyPort accumulate = READ_EXE_PARAM(en_accumulation);

    /* Instantiation of the static and dynamic modules */
    load_data(stream_input, A, B, C);
    execute(A, B, C);
    retrieve_data(stream_output, C, accumulate);
}
END_DEF_TOP_MATRIX_FUNCTION()
```

The **acceleration granularity** goal is addressed by the implementation details of the dynamic module. In Algorithm 7, it is referred to `execute`. Its implementation is similar to the ones presented in Algorithms 5, 6.

4.3 Matrix Accelerator

The matrix accelerator is the design template for accelerators that operates over matrices. The template presents an architecture that supports the GEMMA and activation functions in a SIMD form. This work will focus on the GEMMA implementation only.

In the last chapter, 3.1 was introduced as the generalised matrix multiply-add operation, taking $\mathcal{S}$ and $\mathcal{M}$ as the summation and product operators based on approximate arithmetic. The PE implementation based on 3.1 assumed a limited matrix size because of the granularity principle. In this work, some matrix sizes can be 2×2, 4×4, and 8×8. However, FCL layers are much larger than the sizes mentioned before. For example, in case the of LeNet-5, it is possible to find matrices of $10, 84, 120$, and 256 elements in their columns or rows.

The matrix accelerator comes to solve large matrix operations.

$$\mathbf{D_{ij}} = \mathcal{S}_{k=0}^{J}\{\mathbf{A_{ik}}\mathbf{B_{kj}}\} \tag{4.1}$$

where $\mathbf{D_{ij}}$ is the output submatrix formed by the top-left corner identified by the i-th row and j-th column of the output matrix $\mathbf{D}$. Similar to the output, $\mathbf{A_{ik}}$ and $\mathbf{B_{kj}}$ are the submatrices of the input matrices $\mathbf{A}$ and $\mathbf{B}$. Figure 4.1 illustrates a bigger matrix multiplication graphically. In this case, the figure depicts the operation performed by a single PE of $R \times C$. Since the matrix operation can be divided into three multiplications: $\mathbf{A_{0,0}} \times \mathbf{B_{0,0}}$, $\mathbf{A_{0,C}} \times \mathbf{B_{R,0}}$, and $\mathbf{A_{0,2C}} \times \mathbf{B_{2R,0}}$. it requires three runs until accumulating all the results before writing the output submatrix $\mathbf{D_{0,0}}$.

This exploits the capability of solving multiply-additions, $\mathbf{A} \times \mathbf{B} + \mathbf{C}$, where $\mathbf{A}$ and $\mathbf{B}$ are the current operands and $\mathbf{C}$ is the accumulation of the previous operations.

4.3.1 Template Overview

This work proposes the implementation of the accelerators as reconfigurable templates, where the execution stage can hold different PE implementations in a vector fashion to achieve SIMD. Moreover, the accelerator shall have a standard interface to keep compatibility with the driver, regardless of the parameters or the PE implementation. Hence, this work proposes an accelerator design with two types of modules: static and dynamic.

The static modules do not vary their implementation, but they can adapt according to the accelerator parameters to keep the interface standard. For instance, the data ex-

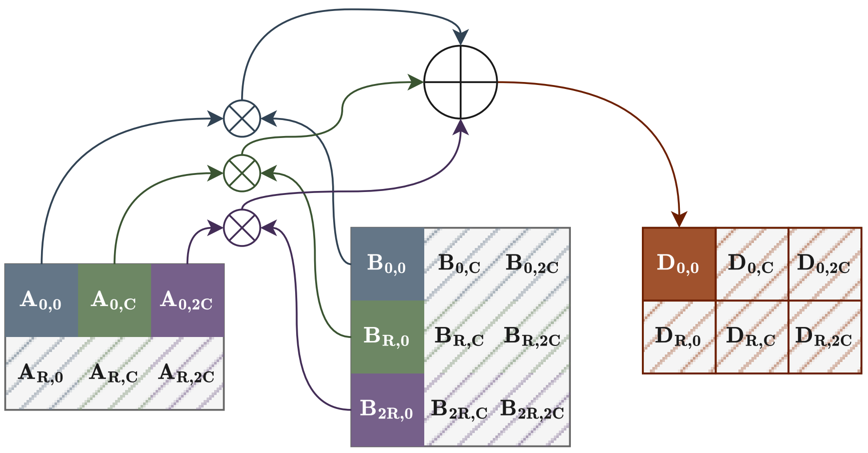


Figure 4.1: Illustration of the calculation of an output submatrix from the inputs.

change modules do not vary their implementation in terms of the streaming interface protocol, register implementation, and data retrieval/write patterns. However, the number of registers, register size and the number of packets received and transmitted through the streaming interface adapt to face the data required by the execution module of the accelerator.

On the other hand, dynamic modules can vary their implementation. This is the case of the execution stages, whose internal cores or PEs are replaceable and granular. Besides, the number of PEs in the execution vector is adjustable, incrementing the internal parallelism of the execution stage.

Figure 4.2 shows an overview of the matrix accelerator implementation. The static blocks are mostly the data exchange modules: *data load* and *data write*. The dynamic block is *execute*.

The data load module receives the packages from the stream. Then, it fills the cache registers in a row-major fashion, receiving the matrices MA, MB, and MC and distributing the data as in Figure 3.3. Through the accelerator configuration, an application can specify which matrices will be provided via the stream. A similar case happens with the data write module, which transmits the data from the cache register MC through the stream once the accumulation has finished. It is crucial to notice that MC is an input/output register, which works as an accumulator and is reset on accelerator read (once the output is finished, the accumulator is cleared).

Moreover, the execution module is the composition of several PEs in a vector. The number of PEs, their implementation, and configurations, such as the PE size and data type, are determined at synthesis time through some of the accelerator parameters.

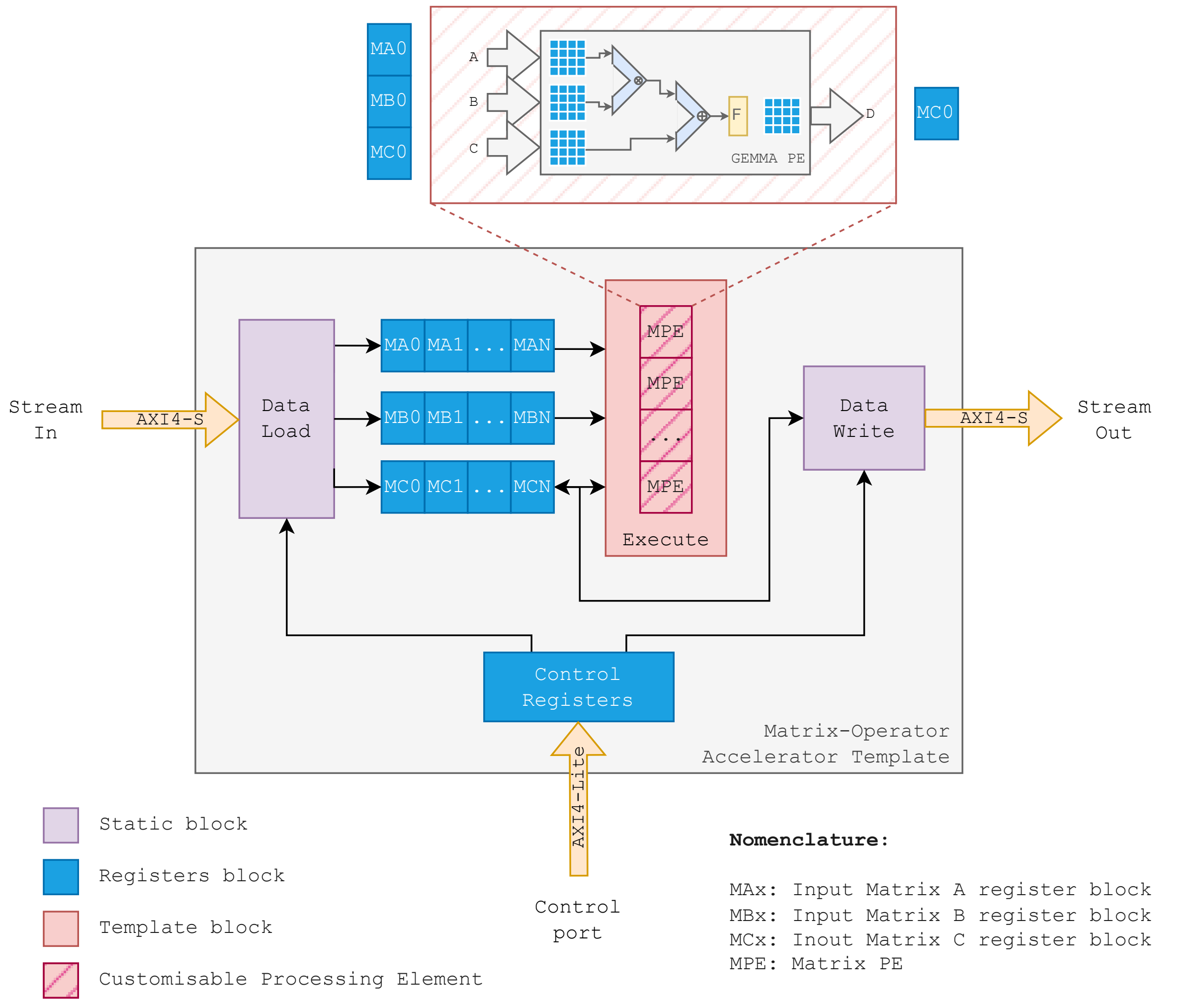


Figure 4.2: Matrix accelerator template. It has three main modules: (1) data load, (2) execute, and (3) data write

.

4.3.2 Capabilities and Parameters

Table 4.2 presents the synthesis-time parameters of the accelerator and its affection on every module. The *data type-related* and *matrix size* parameters affect all modules. The increase of any of these parameters affects the data streaming, where the input/output streams might require to transmit more packages as the number of data width bits increases. It also affects the number of flip flops, making the design consume more resources to deal with the increased bits. The *number of PEs* also affects static modules since the vector will require more data to be efficient. Nonetheless, the *core* and *arithmetic operators* do not affect the static modules but the dynamic ones, given that they specify how the PEs are implemented.

The accelerator has registers that describe the capabilities (read-only) and runtime parameters. These registers are described and checked by the implementation of the template accelerator macros. Table 4.3 describes some of these parameters.

Table 4.2: Matrix accelerator template parameters

Parameter	**Data load**	**Execute**	**Data write**
Data type-related	Input register size	None (only at the PE level)	Number of packets, Output register size
Matrix size	Input register size	None (only at the PE level)	Output register size, number of packets
Core	None	PE implementation	None
Number of PEs	Input register rows, number of packets	Number of PEs	Output register rows, number of packets
Arithmetic operators	None	PE implementation	None

Table 4.3: Matrix accelerator template capabilities and parameters (at runtime)

Parameter	**Description**	**Capability**	**Description**
Rows	Matrices rows	Rows	Minimum and maximum rows of the matrices
Columns	Matrices columns	Columns	Minimum and maximum columns of the matrices
Load matrix mask	Selection of matrices passed through stream	Masking	Accelerator supports masking
Enable accumulation	Enable the accumulation of the results	Accumulation	Accelerator supports accumulation
Enable activation	Enable the activation function	Activation	Accelerator supports activation
Scaling factor	Modifies the scale factor	Scaling	Accelerator supports scaling
Operation	Select the matrix operation	Operations	List of supported operations
		Default scaling	Default scaling applied to the results
		Number of cores	Number of PE units of the accelerator
		Datatype	Datatype supported by the accelerator
		Number of integer bits	Number of integer bits in Fixed-Point data types
		Number of fractional bits	Number of fractional bits in Fixed-Point data types

Each parameter or capability listed above is assigned to an address. The final RTL implementation has a description of these registers with their offsets to be included within an application.

4.3.3 Optimisation Points

The optimisation points in this implementation are the following:

- **Register bank partitioning**: the registers can be implemented by using either BRAM or registers (based on flip flops). Provided that this PE holds four matrices, partitioning can lead to a high FF utilisation.
- **Data load loops**: the data load modules are implemented using loops. They can be unrolled or pipelined.
- **Internal module registers**: they can be partitioned entirely or partially. It is relevant to mention that it might require reshaping to distribute the operands properly.
- **Internal PE implementation details**: the PE unit can be optimised for all the units present in the accelerator.
- **Vector unit**: the units can be serialised or parallelised.

These optimisation points are considered for the DSE in the following sections.

4.3.4 Possible Implementations

Table 4.4 shows the set of possible optimisations for the template matrix accelerator. It supports from the serial execution up to the pipeline architecture. In the case of the serial architecture, it does not have any directives set by default. The pipeline architecture can be in two possible configurations: a pipeline per module, or pipeline modules (all the modules become a pipeline stage). The dataflow is not supported given that there is a feedback between the output and one of the inputs in the accumulation matrix.

Figure 4.3 summarises the possible implementations. The serial implementation is computed as the sum of all stage times:

- Data load: $3 \times N \times R \times C \times \text{It}$, hence: $3 \times 4 \times 2 \times 2 \times 2 = 96$. The 3 comes from the number of matrices.
- Execute: $N \times R \times C^2$, hence: $4 \times 2 \times 2^2 = 32$
- Data write: $N \times R \times C \times \text{It}$, hence: $4 \times 2 \times 2 \times 2 = 32$

Table 4.4: Matrix accelerator template optimisations

Implementation	**Optimisations**
Serial architecture	None
Hybrid: pipelined modules but serial daisy chain	(1) Pipeline in execution (PE), (2) Loop unrolling in execution (PE) (3) Array partition in cache buffers. (4) Vectorisation
Pipeline	all above + pipelined modules (top module)

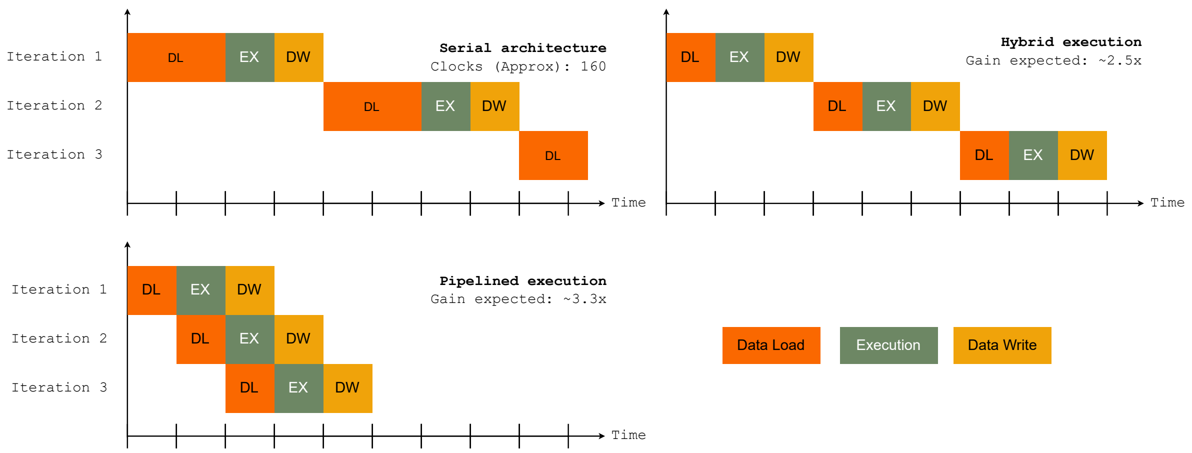


Figure 4.3: Possible implementations of the matrix GEMMA in terms of execution per stage. The serial architecture implies computing a single element at a time per stage, leading to high latencies, whereas hybrid pipelining and pipelining simplify each stage to get a lower latency. This assumes that data is not compressed into transfer packages and that the accelerator is composed of 2×2 operands. Each operation is assumed to take one clock cycle.

where N is the number of PEs, R, C are the matrix dimensions (assumed to be squared), It is the number of runs (matrix reshapening). For a serial implementation, the sum leads to 160 clock cycles.

In pipeline-based architectures, the clock cycles are reduced as follows:

- Data load: $3 \times N \times R \times C$, hence: $3 \times 4 \times 2 \times 2 = 48$. The reshapening is removed since it is fully unrolled. The latency can be less if the data is compressed in packets.
- Execute: since it is fully unrolled: 1
- Data write: $N \times R \times C$, hence: $4 \times 2 \times 2 = 16$

Hence, the hybrid will be the sum of the optimised stages, it leads to 65 clock cycles, whereas the full pipelined implementation to the slowest stage (data load), leading to 48 clock cycles. There are other possible optimisations, such as packeting, which need to be taken into account. However, it minimises the number of clock cycles in data movements,

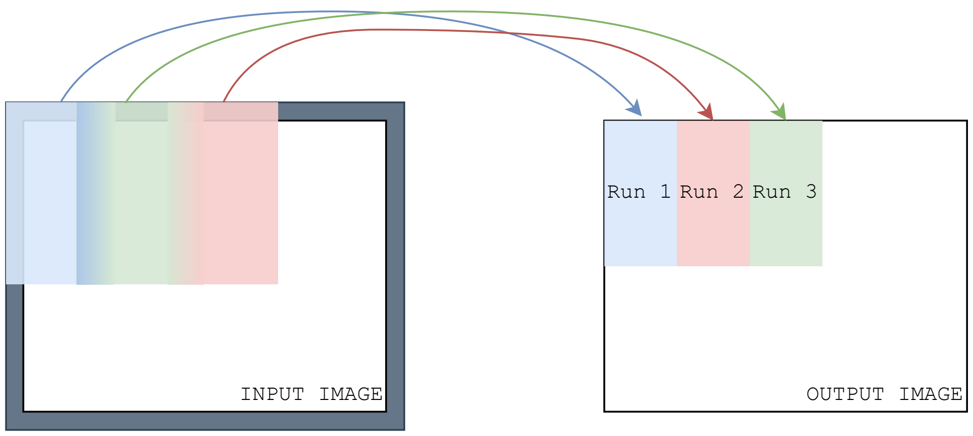


Figure 4.4: Image mapping when running a convolution accelerator. In this case, it includes an implementation with two PEs, where the accelerator is capable of running more rows than columns (twice columns). At the output image, it does not overlap whereas the input image has some overlappings, meaning that some pixels are transmitted twice.

reducing the performance until gains of 10× when using 8-bit operands. This will be discussed in the following sections.

4.4 Convolution accelerator

Unlike the matrix accelerator, the convolution accelerator only supports the convolution accelerator. 3.3 presented two PE implementations: Spatial and Winograd convolution. However, similar to matrix-based PEs, the convolution PEs only operate over small image regions. The convolution defined for a single pixel is

$$Y_{ij} = \mathbf{X}^{(\mathbf{ij})} \odot \mathbf{K} \tag{4.2}$$

where $\mathbf{X}^{(\mathbf{ij})}$ is a 3×3 windows centred at i, j. For a window, it is defined as

$$\mathbf{Y}' = \bigodot_{(i=0,j=0)}^{(N_Y,N_Y)} \left(\mathbf{X}^{(\mathbf{ij})}, \mathbf{K}\right) \tag{4.3}$$

where $\bigodot_{(i=0,j=0)}^{(N_Y,N_Y)}$ is the convolution operator over the pixel i,j when iterating $i = 0, 1, 2, \dots, N_Y - 1$, $j = 0, 1, 2, \dots, N_Y - 1$ in all their permutations $(N_Y \times N_Y)$. To generalise the operation over the entire image using a PE of output size $N_Y \times N_Y$, it is required to extract all the possible regions in the output image, as illustrated in Figure 4.4.

4.4.1 Template Overview

The convolution accelerator is based on the same ideas as the matrix accelerator template. It has a template module that integrates the PEs and other peripheral modules in charge of data transmission, caching and control. Figure 4.5 shows a block diagram of the convolution accelerator template, composed by

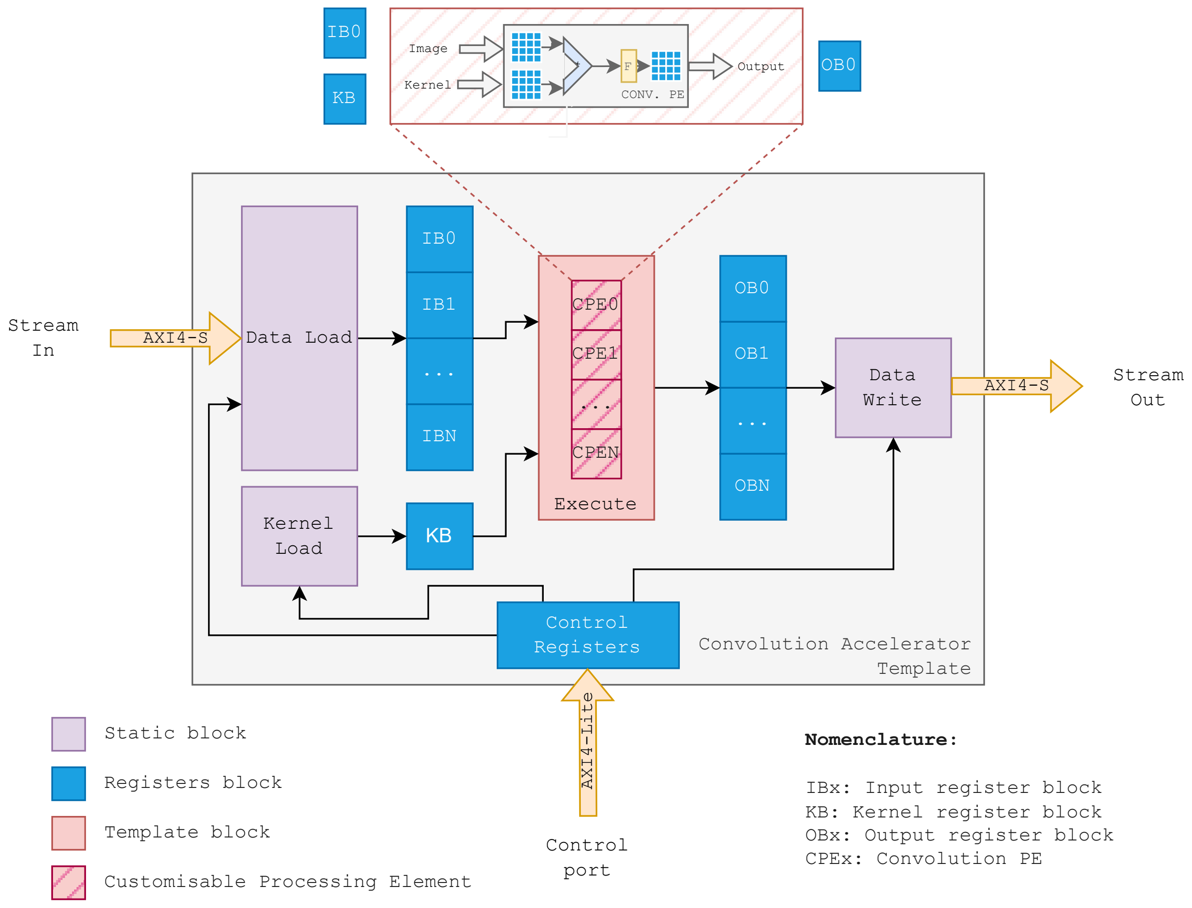


Figure 4.5: Convolution accelerator template. It has four main modules: (1) data load, (2), kernel load, (3) execute, and (4) data write

.

- data load: receives the input stream and stores the image windows into a register back,
- kernel load: receives the kernel from the control port and stores it in a register bank,
- execute: whose implementation is customised and based on PEs, and
- data write: which reads the output register bank with the output windows and transmits them to the output stream in row-major format.

In the execution block, the number of PEs and the implementation of them is selected by the user. In this work, the developer is able to choose either Winograd or the Spatial convolution. Moreover, the communication protocols are based on AXI-4. The data

Table 4.5: Convolution accelerator template parameters

Parameter	**Data load**	**Kernel load**	**Execute**	**Data write**
Data type-related	Number of packets, Kernel register size	Input register size	None (only at the PE level)	Number of packets, Output register size
Kernel size	Input register size, number of packets	Kernel register size	None (only at the PE level)	None
Output window size	Input register size, number of packets	Input register size	None (only at the PE level)	Output register size, number of packets
Core	None	None	PE implementation	None
Number of PEs	Input register rows and number of packets	None	Number of PEs	Output register rows, number of packets
Arithmetic operators	None	None	PE implementation	None

transmission is transferred using AXI Stream and the control using AXI Lite. The implementation of the registers block and in-module optimisations are still customisable by the users, allowing them to find directives that maximises throughput or balance performance and resource consumption.

4.4.2 Parameters

The synthesis-time parameters of the accelerator and its affection on every module is the presented in Table 4.5. The affectation results are in the same manner as in the Matrix Accelerator.

The accelerator has registers that describe the capabilities (read-only) and runtime parameters. These registers are described and checked by the implementation of the template accelerator macros. Table 4.6 details these registers.

Each parameter or capability listed above is assigned to an address. The final RTL implementation has a description of these registers with their offsets to be included within an application.

4.4.3 Optimisation Points

The optimisation points in this implementation are the following:

- **Register bank partitioning**: the registers can be implemented by using either BRAM or registers (based on flip flops). Provided that this PE holds four matrices,

Table 4.6: Convolution accelerator template capabilities and parameters (at runtime)

Parameter	**Description**	**Capability**	**Description**
Input width	Input matrix width	Input width	Minimum and maximum width of the input matrix
Input height	Input matrix height	Input height	Minimum and maximum height of the input matrix
Output width	Output matrix width	Output width	Minimum and maximum width of the output matrix
Output height	Output matrix height	Output height	Minimum and maximum height of the output matrix
Kernel size	Current size of the kernel	Maximum kernel size	Maximum kernel size supported
Number of kernels	Number of kernels to load	Maximum number of kernels	Maximum number of kernels supported
Scaling factor	Modifies the scale factor	Scaling	Accelerator supports scaling
Padding type	Padding type to apply	Padding	Accelerator supports padding
Dilatation	Dilatation in X and Y (individually)	Dilatation	Accelerator supports dilatation
Stride	Stride in X and Y (individually)	Stride	Accelerator supports striding
		Number of cores	Number of PE units of the accelerator
		Datatype	Datatype supported by the accelerator
		Number of integer bits	Number of integer bits in Fixed-Point data types
		Number of fractional bits	Number of fractional bits in Fixed-Point data types

partitioning can lead to a high FF utilisation.

- **Data load loops**: the data load modules are implemented using loops. They can be unrolled or pipelined.
- **Internal module registers**: they can be partitioned entirely or partially.
- **Internal PE implementation details**: the PE unit can be optimised for all the units present in the accelerator.
- **Vector unit**: the units can be serialised or parallelised.

Table 4.7: Convolution accelerator template optimisations

Implementation	**Optimisations**
Serial architecture	None
Hybrid: pipelined modules but serial daisy chain	(1) Pipeline in execution (PE), (2) Loop unrolling in execution (PE) (3) Array partition in cache buffers. (4) Vectorisation
Pipeline	all above + pipelined modules (top module)
Dataflow	implement a dataflow at the core level.

These optimisation points are considered for the DSE in the following sections.

4.4.4 Possible Implementations

Table 4.7 shows the set of possible optimisations for the template convolution accelerator. It supports from the serial execution up to the dataflow architecture. In the case of the serial architecture, it does not have any directives set by default. The pipeline architecture can be in two possible configurations: a pipeline per module, or pipeline modules (all the modules become a pipeline stage). The data flow is at the top level, where the synthesis tool decides how the data flow implementation is performed.

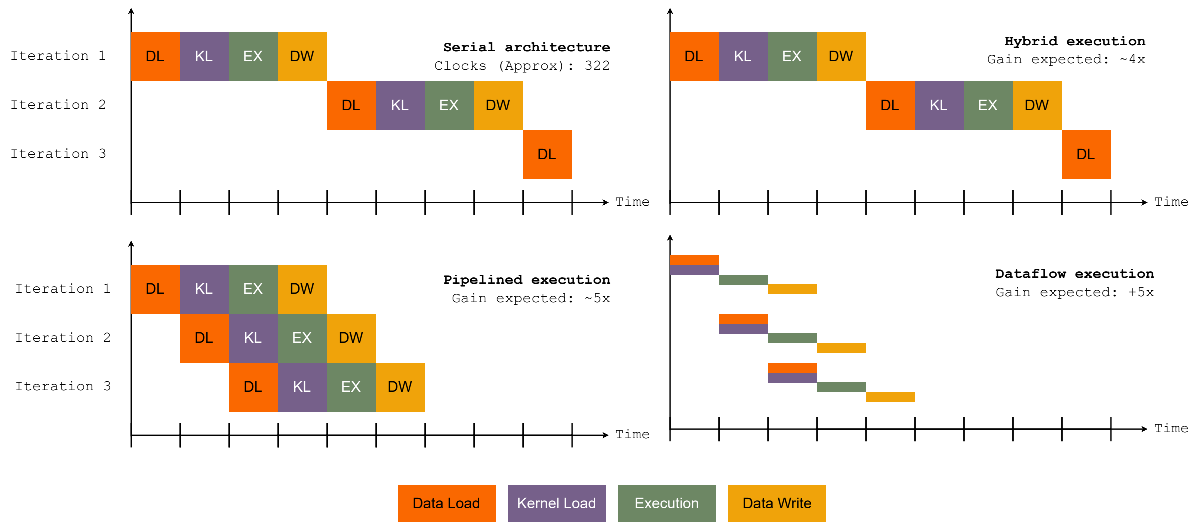


Figure 4.6: Possible implementations of the convolution in terms of execution per stage. The serial architecture implies computing a single element at a time per stage, leading to high latencies, whereas hybrid pipelining and pipelining simplify each stage to get a lower latency. Due to the synthesis tool decisions, the dataflow architecture is more complex to estimate. This assumes that data is not compressed into transfer packages and that the accelerator is composed of 4 PEs of 2×2 operands. Each operation is assumed to take one clock cycle.

Figure 4.6 summarises the possible implementations. The serial implementation is computed as the sum of all stage times:

- Data load: $N \times N_X \times N_X \times \text{It}$, hence: $4 \times 4 \times 4 \times 2 = 128$
- Kernel load: $N_K \times N_K \times \text{It}$, hence: $3 \times 3 \times 2 = 18$
- Execute: $N \times N_K^2 \times N_Y^2$, hence: $4 \times 3^2 \times 2^2 = 144$
- Data write: $N \times N_Y \times N_Y \times \text{It}$, hence: $4 \times 2 \times 2 \times 2 = 32$

where N is the number of PEs, N_X, N_K, N_Y are the convolution operand sizes (input, kernel, and output matrices, respectively), It is the number of runs (matrix reshaping). For a serial implementation, the sum leads to 322 clock cycles.

In pipeline-based architectures, the clock cycles are reduced as follows:

- Data load: $N \times N_X \times N_X$, hence: $4 \times 4 \times 4 = 64$. The reshaping is removed since it is fully unrolled. The latency can be less if the data is compressed in packets.
- Kernel load: since it is fully unrolled: 2 (one additional for resharpening)
- Execute: since it is fully unrolled: 1
- Data write: $N \times N_Y \times N_Y$, hence: $4 \times 2 \times 2 = 16$

The hybrid execution time will be the sum of the optimised stages, which leads to 83 clock cycles. In contrast, the full pipelined implementation has an execution time to the slowest stage (data load), leading to 64 clock cycles. The data flow is assumed to have slightly better performance than the pipeline and is hard to estimate because of the synthesis tool decisions. Moreover, the data is not in packets, and the gains can be even higher, reaching gains of $10\times$ when using 8-bit operands.

The following sections will expand on these results when performing the DSE.

4.5 Summary

Similar to 3.5, this chapter have addressed the accelerator design goals and implementation of accelerators taking into consideration design goals to differentiate similar research. The details about the approach followed to meet the design goals are:

- **Template structure**: the accelerators have static and dynamic structures, as presented in Algorithm 7. The dynamic structure can adjust the PE implementation and the number of units through template parameters. Moreover, the static parts also adapt according to other accelerator parameters such as the data type, PE operands' size, and data width.

- **Standard interface**: the accelerator parameters are incapable of modifying the interface, but the packets that flow through the stream ports. Moreover, the accelerator provides a series of registers for runtime configuration, assisting in driver compatibility and application control. The macros used in Algorithm 7 prove this goal accomplishment.

- **Configurable**: the accelerator is configurable at synthesis time by configuring parameters such as the data type, data width, core, operands' size, and others. Moreover, it can be also configured at runtime. Moreover, it can also be configured at runtime. Algorithms 5 and 6 prove the capability of configuring the accelerator at synthesis time, and the parameters check in Algorithm 7 proves the configurability at runtime.

- **Simulable**: since the accelerator is implemented using C++ and HLS, the framework can simulate the design. At the implementation detail, a testbench is implemented within the framework.

- **Acceleration granularity**: given the configurability, it is possible to choose between a big accelerator singleton or various small accelerators that can behave like processing units. Algorithms 5, 6 report this goal accomplishment.

By accomplishing these goals, this work offers a template to construct granular and configurable accelerators that can fit several use cases. DSE is crucial for determining the values of the parameters to configure the accelerator and adjust the granularity to a configuration that can satisfy the design requirements.

Chapter 5

Flexible Accelerators Library (FAL) Framework

Apart from the processing elements and the accelerators, this work proposes a framework to ease the implementation of new hardware units. Moreover, evaluating each implementation's consumption, performance, and error is critical to select the most appropriate designs.

The inspiration for creating FAL is fundamentally developing a set of PEs and accelerators for general computing applications highly inspired in DLI but not limited to this field exclusively. Hence, the framework seeks to be easy for developers and users.

5.1 Design Goals

The design goals set for the framework are:

- **Easy accelerator configuration**: an accelerator shall be configurable in several degrees of freedom (DoF). The idea is to get these configurations easily from the user.
- **Abstraction for development**: the user shall be capable of implementing their accelerators without worrying about communication interfaces or address spaces for the most common accelerators. FAL shall be enough for the user to focus on the actual algorithm.
- **Simulation**: FAL shall offer a way to run simulations at the C and RTL levels through C/RTL co-simulation.
- **Design Space Exploration**: FAL shall run DSE by specifying possible values for each DoF and greedily run all the permutations. Moreover, it shall make the most of all the development host capability (multi-process DSE). most of all the development host capability (multi-process DSE).

- **Extensibility**: FAL shall be extensible to other applications different from DLI.

As described in the design goals, FAL searches for usability. The **easy accelerator configuration** helps users to get several accelerator configurations and generate the RTL for further testing. Nevertheless, the **simulation** goal allows the user to shortcut the C++ to the bitstream path without incurring long waiting times, verifying the accelerator's performance, consumption and errors before proceeding with the synthesis on the FPGA. Also, it enables the possibility of determining by trial and error parameters, which is unsuitable when waiting until the bitstream is produced. The **design space exploration** capability allows users to explore numerous combinations of DoFs to determine the most suitable candidates in terms of performance, consumption and error. In the following chapter, this work presents an analysis of the PEs and accelerators that is possible with FAL.

This work aims to be different from other approaches being **extensible** to other applications. DLI is an application, but there are other fields in computing at the edge. An example is image processing applications that can be computed in low-end FPGAs. With extensibility, **abstraction for development** also comes to the table, easing the development of multi-purpose accelerators and offering tools to avoid users focusing on tasks unrelated to the actual algorithm or accelerator logic implementation.

5.2 General Structure

FAL is composed of a series of scripts based on Makefiles and Python that invokes a Vivado/Vitis HLS TCL script. FAL utilises Makefiles for single jobs, i.e. **simulation** and testing **accelerator configurations**. In contrast, the Python scripts run the **Design Space Exploration** and collect the data for data visualisation. After the TCL script, FAL adopts a standard implementation for the FPGA HLS workflow, receiving source code, testbench files and directives for optimisation. Figure 5.1 describes the structure and the goals before entering into the implementation.

About the source code, there is a hierarchy coming from the fact of using a granular design. FAL integrates source code from several Git repositories required for acquiring the PEs for acceleration implementation. The building system, based on Makefiles, sets the development environment to import these PE libraries to the user seamlessly, similar to a typical C++ project. This characteristic makes FAL **extensible**. Moreover, FAL also includes wrappers for creating accelerators, **abstracting the accelerator implementation** process and easing the development without losing generality. The following sections will detail these facts.

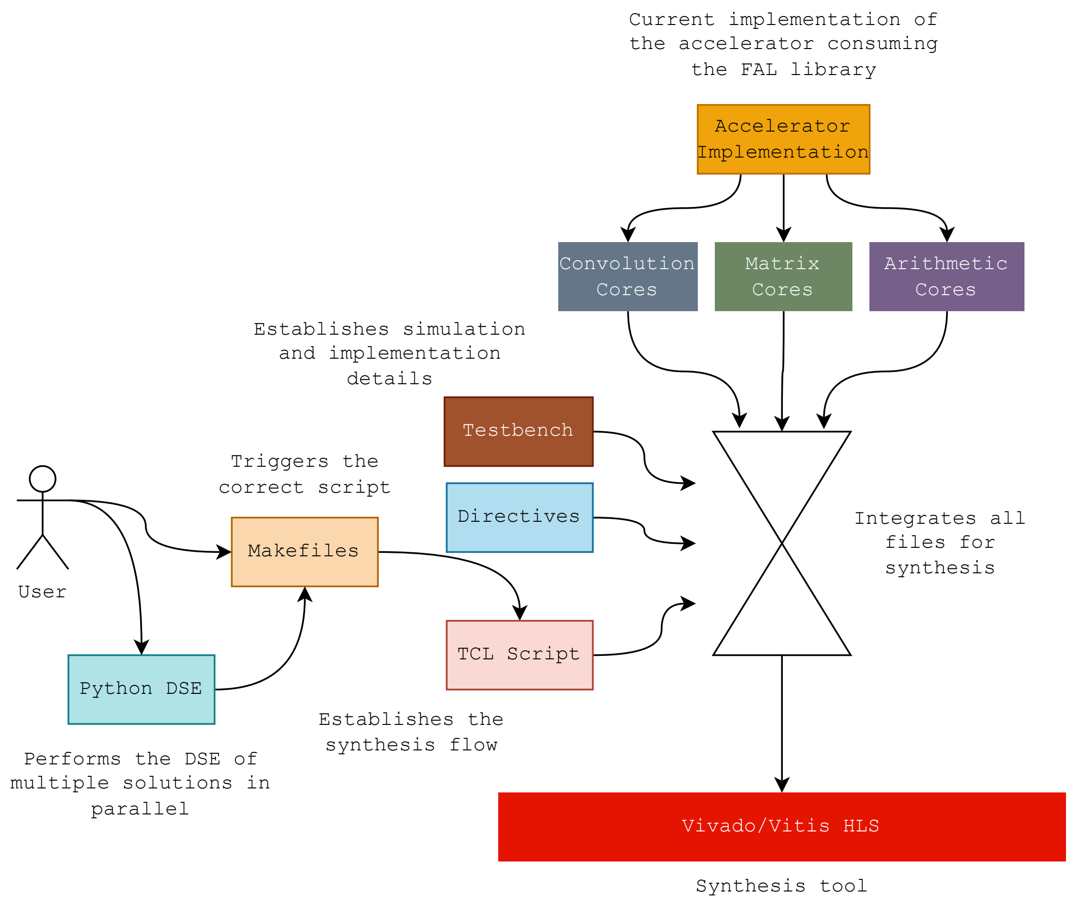

Current implementation of the accelerator consuming the FAL library
Accelerator Implementation
Convolution Cores
Matrix Cores
Arithmetic Cores
Establishes simulation and implementation details
Testbench
Triggers the correct script
Directives
Makefiles
Integrates all files for synthesis
User
TCL Script
Python DSE
Establishes the synthesis flow
Performs the DSE of multiple solutions in parallel
Vivado/Vitis HLS
Synthesis tool

Figure 5.1: FAL components flow. The user triggers a makefile for a single-design synthesis or the Python DSE for a massive DSE exploration. It also accepts directives for optimising the design and the test bench for testing. The FAL components and implementations come as the source code consumed by the synthesis tool.

5.3 Implementation

5.3.1 Framework Composition

The proposed framework is based on GNU Makefiles, Python Vivado/Vitis HLS. It allows the PE and accelerator parameterisation through compilation-time parameters, specialising a generic design according to different requirements and constraints. One of the novelties proposed by this work is the addition of arithmetic operators as parameters, giving more flexibility to define custom and approximate operators, such as approximate multipliers [88] (not explored in this work), which contribute to reducing the required resources for the implementation. Apart from the operators, the designs are parametrised regarding the PE (or core) implementation, matrix size, kernel size, activation functions, data width and type to explore different designs, customising the precision according to the accuracy constraints. This document later explores the impact of several configurations on the numerical error and the performance of PEs and accelerators.

Table 5.1: Accelerator-specific macros: used for declaring the structure of the accelerator.

Macro	**Scope**	**Description**
DEF_SET_*_CAPABILITIES()	Accelerator's core	Defines the capabilities of the accelerator.
DEF_SET_*_PARAMS_CHECK()	Accelerator's core	Defines the parameters check of the accelerator.
BEGIN_DEF_TOP_*_FUNCTION(), END_DEF_TOP_*_FUNCTION()	Accelerator's core	Defines the top function of the accelerator. It connects the static and dynamic modules.

Beyond the generation system, the framework proposes templates for constructing matrix and convolution accelerators, as addressed in 4. These templates act as wrappers to encapsulate implementation and characterisation. Some of these templates are composed of pre-processor macros. Table 5.1 sshows the structural macros, which define the canonical structure of an accelerator, and its capabilities, check runtime parameters and define the top function[1]. The `*` can be replaced by `MATRIX` or `CONV` for the matrix or convolution accelerator, respectively.

Table 5.2 shows the generic macros that can be used in any accelerator. They declare, write, and access parameters and capabilities. Some macros apply only to test benches. For instance, the `DECL_EXE_PARAM` ia used for declaring the function arguments when defining a structural macro from Table 5.1. Instead, `DECL_EXE_PARAM_TB` is used for declaring a variable in the body of a function in a testbench, that will be later accessed to call the accelerator.

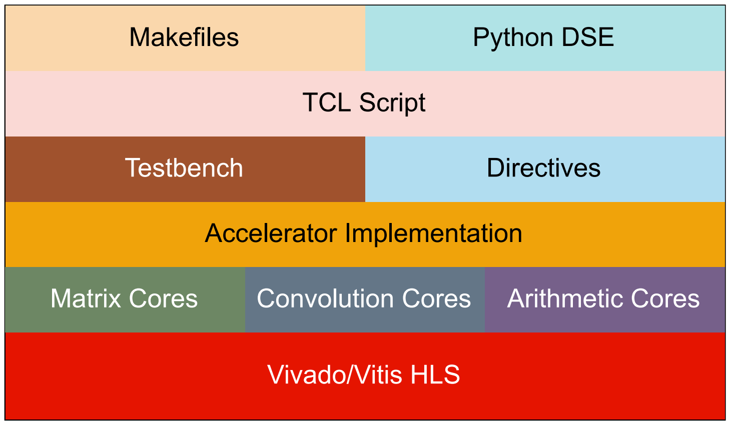


Figure 5.2: FAL stack: it runs on top of Vivado/Vitis HLS. The accelerator library is based on the PE library, which is imported as Git submodules. Then, there is a series of files that completes the implementation (directives) and the simulation (test benches). On top of the hierarchy, the framework runner files are presented.

[1]The top function is assumed to be the container of all the modules or the main module to implement

Table 5.2: Generic macros in FAL: they can be used in any accelerator

Macro	**Scope**	**Description**
`DECL_EXE_PARAM`, `DECL_EXE_PARAM_TB`	Function arguments and testbench	Declares a execution parameter. It declares two variables: one for reading and the other for writing.
`DECL_CAP`, `DECL_CAP_TB`	Function arguments and testbench	Declares a read-only variable to hold a capability.
`DECL_RANGE_CAP`, `DECL_RANGE_CAP_TB`	Function arguments and testbench	Declares two read-only variables to hold a minimum and maximum value.
`WRITE_CAP`, `ACCESS_CAP`	`DEF_SET_*_CAPABILITIES()`, top function	Writes a capability and access to its value.
`DISABLE_CAP`, `ENABLE_CAP`	`DEF_SET_*_CAPABILITIES()`	Disables or enables a capability
`ENABLE_OPERATION`	`DEF_SET_*_CAPABILITIES()`	Enables an operation
`WRITE_RANGE_CAP`	`DEF_SET_*_CAPABILITIES()`	Writes a range capability
`ACCESS_CAP_MIN`, `ACCESS_CAP_MAX`	Top function	Access to the minimum and maximum value of the capability
`READ_EXE_PARAM`, `WRITE_EXE_PARAM`	`DEF_SET_*_PARAMS_CHECK()` and top function	Reads and writes a parameter.
`READ_EXE_PARAM_TB`, `WRITE_EXE_PARAM_TB`	Testbench body	Reads and writes a parameter.
`CHECK_EQUALITY_AND_FALLBACK`	`DEF_SET_*_PARAMS_CHECK()`	Checks if the parameter is equal to a value. If not, it fallbacks into a default value.
`CHECK_RANGE_AND_FALLBACK`	`DEF_SET_*_PARAMS_CHECK()`	Checks if the parameter falls into a range. If not, it fallbacks into a default value.

Other accelerators might require defining macros for this purpose. However, it still follows the canonical structure of parameters checking, capabilities and top function. An example of a resolved pre-processor structure is presented in Algorithm 8, which is based on Algorithm 7. In this case, the structural macros solve the verbosity of declaring too many variables, particularly when the accelerator is highly parameterised, where managing too many arguments becomes problematic.

Apart from offering the macros, the source code for the PEs is also available in the FAL framework. It is integrated as Git submodules for handling the source code versioning. The whole project is open source under Apache 2. The implementations for the matrix multiply-add, the convolutions, and their respective generation frameworks are available in [92], [93]. The full stack of the components presented above is illustrated by Figure 5.2, presenting the runner files, the accelerator and PE libraries and the synthesiser.

Algorithm 8 Generic accelerator file structure. This explicitly defines the functions required for the canonical accelerator

```
void caps(DECL_RANGE_CAP(DimensionPort, rows),
    DECL_RANGE_CAP(DimensionPort, columns)) {
   // Defines the capabilities of the accelerator
   WRITE_RANGE_CAP(rows, 2, 8);
   WRITE_RANGE_CAP(columns, 2, 2);
}

void check(DECL_EXE_PARAM(DimensionPort, rows),
    DECL_EXE_PARAM(DimensionPort, columns),
    DECL_EXE_PARAM(PropertyPort, en_accumulation)) {
   // Checks the runtime parameters against the capabilities.
   CHECK_RANGE_AND_FALLBACK(rows, 2, 8, 2);
   CHECK_RANGE_AND_FALLBACK(columns, 2, 2, 2);
}

void accel(StreamPort& stream_input,
  StreamPort& stream_output,
  DECL_RANGE_CAP(DimensionPort, rows),
  DECL_RANGE_CAP(DimensionPort, columns),
  DECL_EXE_PARAM(DimensionPort, rows),
  DECL_EXE_PARAM(DimensionPort, columns),
  DECL_EXE_PARAM(PropertyPort, en_accumulation)) {
   // Invoke checking
   caps(ACCESS_CAP_RANGE(rows), ACCESS_CAP_RANGE(columns));
   check(ACCESS_EXE_PARAM(rows), ACCESS_EXE_PARAM(columns),
      ACCESS_EXE_PARAM(en_accumulation));

   // Implementation details
   static DataType A[8][2] = {0.f};
   static DataType B[8][2] = {0.f};
   static DataType C[8][2] = {0.f};

   // Properties
   PropertyPort accumulate = READ_EXE_PARAM(en_accumulation);

   // Modules
   load_data(stream_input, A, B, C);
   execute(A, B, C);
   retrieve_data(stream_output, C, accumulate);
}
```

5.3.2 FAL Runner Suite

FAL provides a building system for the synthesis and simulation entirely based on Python, Makefiles, Open Message Passing Interface (MPI), and Vivado/Vitis HLS. It aims to offer an automated interface for design exploration, allowing users to get information from several design solutions and add their top functions, directives, and test benches straightforward without invoking the Vivado/Vitis HLS graphical user interface. For synthesis, it is possible to vary the parameters for the design.

For generating the synthesis and the simulation, the Makefile framework offers the following targets:

- `synthesis` (Accelerator): synthesises the selected design. It just runs Vivado HLS configuring the project and generating the solution.
- `test` (PE): synthesises and simulates the selected design. It just runs Vivado HLS configuring the project and generating the solution.
- `measure` (PE): same as test and extracts the synthesis report and simulation results for further analysis. It runs the selected design given its parameters only.
- `measure-all` (PE): same as `measure` but for all possible solutions described in the configuration scripts The solutions are synthesised and simulated in parallel using MPI. It extracts all data required for this research.
- `extract-data` (PE): post-processes the information collected by `measure-all`, curates data, and generates the plots used for reports.

The *accelerator* and *PE* define the applicability of the makefile target.

For selecting the design and its parameters, it is possible to modify the environment variables:

- `ACCELERATOR`: chooses the accelerators. By default, the examples are synthesisable. Examples: `gemma, convolution, minimum`
- `Q_KS` (convolution): kernel size in $Q_{KS} \times Q_{KS}$
- `Q_BW/Q_WL`: bit-length in fixed-point. It defines the total width per number.
- `Q_INT`: bit-length of the integer part in fixed-point. It defines the total width of the integer part per number.
- `Q_O`: output size in $Q_O \times Q_O$
- `Q_INPUTS_TB` (gemma): number of inputs elements.
- `Q_OUTPUTS_TB` (gemma): number of output elements.
- `Q_CORE`: PE core. It is possible to select the spatial convolution or Winograd in case of convolution.
- `TB_ARGS`: arguments for the testbench. It is used to change the image sample in convolution.

- `Q_PES`: chooses the number of PEs of the accelerador.
- `Q_SEED`: chooses the Seed for random numbers.
- `CLOCK_PERIOD`: chooses the clock period in [ns].

Moreover, it is possible to define ranges of the values for the measurement targets to produce multiple solutions per run through JSON files. Hence, it is possible to take several design samples from the design space for design exploration tasks.

5.3.3 Design Exploration Support

The proposed framework allows changing the implementation directives for trying several optimisations at the HLS level. The implementation directives help map the C++ code into an RTL architecture, allowing pipeline constructions, defining protocols, and data flows. This capability allows users to have multiple options during the design exploration, where the options are likely to have different design performances in terms of latency and resource consumption. The next chapter introduces a novel figure of merit to compare the solutions and evaluate the efficiency of each design produced by the framework.

Apart from changing the directives, FAL also allows changing the DoF values of the design. A Python script reads the values to explore from a JSON file and computes the permutations, allowing a greedy DSE process. An example is presented in Algorithm 9, which performs a DSE of a Winograd-based convolution accelerator of 54 different configurations regarding data width, the number of PEs and input images.

Algorithm 9 Example configuration file to perform a DSE on a convolution, selecting a `Winograd` core.

```
{
  "ACCELERATOR": ["convolution"],
  "Q_KS": [3],
  "Q_BW": [4, 6, 8, 10, 12, 16],
  "Q_INT": [1],
  "Q_O": [2],
  "Q_CORE": ["Winograd"],
  "Q_PES": [1, 2, 4],
  "CLOCK_PERIOD": [7.5],
  "TB_ARGV":
      ["examples/convolution/misc/lenna.png",
       "examples/convolution/misc/baboon.png",
       "examples/convolution/misc/boat.png"],
  "Q_FIXED_RATIO": 0.5,
  "Q_INPUTS_TB": [0],
  "Q_OUTPUTS_TB": [0],
  "Q_SEED": [0]
}
```

Invoking the DSE script not only synthesises the designs for every permutation. The DSE only captures the synthesis data and the simulation results, post-processing the results for data visualisation using `matplotlib`. It makes accessible the report generation.

Finally, the DSE can be executed with MPI. It requires a multi-core processor to execute in parallel, accelerating the DSE process using task-farming techniques.

Chapter 6

Results

After defining all the design goals and the implementation, it is time to evaluate the solutions. In this case, there are three significant fronts to cover from the measurement point of view: (1) resource consumption, (2) performance, and (3) quality of the results. Moreover, the evaluation shall consider both PEs and accelerators since the accelerators integrate other peripherals apart from the PEs.

This chapter introduces the metrics and figures of merits to evaluate both kinds of implementations. The following goals will guide the chapter:

- **Quantify the resources utilised by several configurations**: quantifying the scaling of the resources when modifying the values of the DoF will provide an idea of the affectation on the system.
- **Quantify the error introduced by several configurations**: similarly, quantifying the impact is crucial to discard numerical-ill solutions, although they could be good in terms of resources.
- **Quantify the performance of the solutions**: performance metrics work as a point of comparison in terms of time to solution, helping to contrast against other types of devices, such as GPUs and CPUs.
- **Determine the efficiency of the design**: this is one of the novelties of this work. Determining the efficiency will allow considering resource utilisation and performance in a single value that can be compared to determine how suboptimal a design is.

Following these goals, the chapter will cover the analysis of the solutions proposed during this work, in particular, the PEs and accelerators for performing GEMMA and convolutions. Activation functions are left aside, given that they are less computationally relevant in DLI [42].

6.1 Measurement Instruments

The synthesis tool already provides measurements of resource consumption and latency. However, its conversion to performance and design efficiency requires manual computation provided that they consider the *number of operations*, latency, and clock speed. On the other hand, the error depends on the simulation process and requires generalisation, requiring instrumentation at the test bench level. This section will establish the performance, design efficiency and error metrics used for the solution's evaluation. *Most of the plots and tables concerning the resource consumption analysis, performance evaluation, and error quantification are provided automatically by the FAL framework.*

6.1.1 Performance and Design Efficiency

Given that this research produces several design solutions, it requires a comparison instrument to determine the quality of these solutions when performing design space exploration. The quality considers the trade-offs associated with approximate computing and general design implementation. Hence, performance, resource utilisation, and error are interesting in selecting the most balanced design solutions, which also depend on the user requirements. To the best of our knowledge, no metric or figure of merit allows comparing designs by evaluating computing performance **and** resource utilisation. The closest attempt is presented in [46], proposing the introduction of effective performance by considering a non-complete utilisation of an accelerator and adding the frequency:

$$\text{IPS} = \frac{fP \times \eta}{W} \tag{6.1}$$

where f is the working frequency, P is the number of computation units, η is the inference utilisation ratio, and W is the workload in GOP/s.

Given the absence of the comparison tool, this work proposes a novel figure of merit to quantify the design performance, considering the resource consumption, the number of operations per clock, and the maximum throughput achieved by the DSPs of an FPGA. First, Amdahl's law [94] was adapted to integrate frequency and clock cycles. In the resulting adaptation, the speedup for the p-th solution is

$$S_p = \frac{T_0 f_p}{T_p f_0} \tag{6.2}$$

where T_0 and f_0 are the baseline number of clock cycles and maximum frequency. T_p and f_p correspond to the p-th design. The key difference from the traditional Amdahl's law is that the scaling depends on the frequency, provided that different solutions may have different maximum clock frequencies.

Then, for evaluating the efficiency, the maximum theoretical efficiency of the generated PE, $P_{\max}$, is compared against the maximum performance achievable by the FPGA's DSP units, P_{peak} (this is the theoretical maximum performance of the platform, reported in each FPGA's datasheet), given that frameworks like HLS4ML focus on these units for computing $P_{p,\max}$ is computed as

$$P_{p,\max} = \frac{O f_p}{T_p} \times \left\lfloor \frac{1}{r_p} \right\rfloor \tag{6.3}$$

where p is the solution index in the design space, T_p is the number of clocks, f_p is the frequency of the design, r_p is the maximum occupation of a single PE in the current platform, O is the number of MAC operations performed by the PE, and $\lfloor \cdot \rfloor$ is the floor integer operator.

Everything is ready to introduce the *design efficiency*, which measures how well the design computationally performs as it consumes the FPGA resources. It is measured by

$$\eta_p = \frac{P_{p,\max}}{P_{\text{peak}}} \tag{6.4}$$

In this case, the research targets an Avnet Zedboard based on a ZYNQ 7z020 FPGA-based System-on-Chip (SoC), with $P_{\text{peak}} = 276$ GMAC/s [95] of theoretical throughput. Hence, the design efficiency can be interpreted as *the ratio of the performance obtained after filling the FPGA with the design p replicated* $\lfloor \frac{1}{r_p} \rfloor$ *times compared to the maximum performance obtained using all the FPGA DSPs.* Nevertheless, this work also considers other FPGAs in the following sections.

6.1.2 Error Evaluation

This work proposes measuring the error through the generation of random matrices and computing a pair of data: (1) golden data, computing the results in single-precision floating-point (`float32`) numbers, and (2) the solution results. The choice of using `float32` is because most DLI applications are initially trained using this numerical representation and can be taken as the reference. Regarding the solution results, the generated matrices are quantised to the custom datatype and transferred to the PE. The resulting elements are compared against the gold data as

$$\epsilon_{ij}^{(p)} = \frac{||\hat{x}_{ij}^{(p)} - x'_{ij}||}{\max(\mathcal{S}) - \min(\mathcal{S})} \tag{6.5}$$

where the error is computed as the L1-distance (or error distance) between the result $\hat{\mathbf{X}}$ from the p-th solution and the golden data scaled $\mathbf{X}'$, normalised against the sum of the boundaries of the vector space $\mathcal{S}$. (6.5) will be used to compute the mean error $\mathbb{E}[\epsilon^{(p)}]$, the standard deviation $\sigma^{(p)}$, and the histograms for the error distribution illustrations.

Apart from the L1 distance, other metrics are also taken into account. This is the case of Mean Squared Error (MSE), which corresponds to the deviation of the model (6.6),

Table 6.1: Error metrics usage in the research field

Metric	**Equation**	**Application**
Error distance (L1-distance)	(6.5)	Multiplier design [99], [100]
Mean Squared Error	(6.6)	Activation functions [101]
Peak Signal-to-noise Ratio (PSNR)	(6.7)	Convolution, Multipliers [102]
Frobenius Norm	(6.9)	Convolution, Multipliers [103]
Structural Similarity Index (SSIM)	(6.8)	Convolution [104]
Error histograms based on error distance	(6.5)	Operator design [105]

Peak Signal-to-Noise Ratio (PSNR), providing a measurement of the quality of reconstruction [96] (6.7), and the Frobenius Norm, which is a similarity matrix [97].

$$\mathrm{MSE} = \frac{\sum_{i=1}^{N}(\hat{x}_i - x'_i)}{N} \tag{6.6}$$

$$\mathrm{PSNR} = 10\log\left(\frac{\max(\hat{\mathbf{X}}^2)}{\mathrm{MSE}}\right) \tag{6.7}$$

$$\mathrm{SSIM} = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c2)}{(\mu_x^2 + \mu_y^2 + c1)(\sigma_x^2\sigma_y^2 + c2)} \tag{6.8}$$

$$||\mathbf{A}||_F = \sqrt{\mathbf{Tr}(\mathbf{E}\mathbf{E}^H)} \tag{6.9}$$

For the case of the convolvers, such as the Structural Similarity (SSIM) Index [98] (6.8), and Root Mean Square Error (RMSE). They are going to be used for appropriate image analysis. Moreover, the error quantification will consider the dimensions of the outputs, kernels, and width of the numerical representation.

Table 6.1 presents a summary of the utilisation of the metrics mentioned above.

6.2 Processing Elements

6.2.1 Generic Matrix Multiply-Addition

The matrix multiply-add is implemented with a complete unrolling of its inputs, using one register per element and pipelining on the loop of the rows, unrolling the inner for-loops (columns and dot-product). This research evaluates the PE through a design space exploration of the PE modifying the matrix size and datum's width based on a fixed-point representation with one integer bit and the rest being fractional bits, with 500 runs of the PEs to generate the data exposed in this section.

Resource Consumption and Performance

The first step is determining how the matrix size and datum's width impact the resource consumption and the average latency (clocks to solution). Figure 6.1 shows how the matrix size and the width of the data affect resource consumption. These tests were co-simulated for an Avnet Zedboard equipped with a ZYNQ XC7Z020.

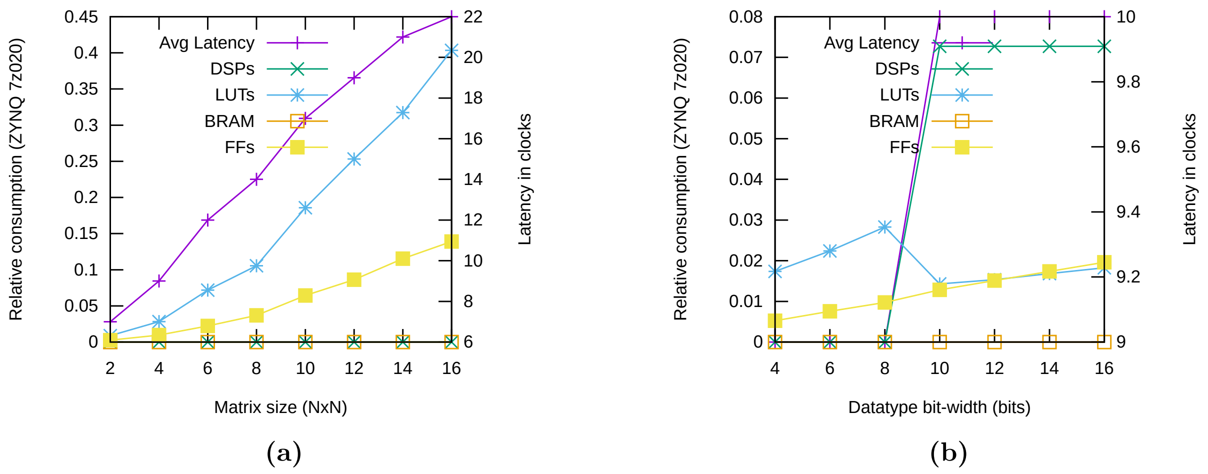


Figure 6.1: Resource utilisation of the generic matrix multiply-add PE on an Avnet Zedboard with a Xilinx XC7Z020 FPGA-based System-on-Chip: **(a)** Impact of the matrix size on consumption. The datum width is fixed to 8-bits, **(b)** Impact of the datum's bit-width on consumption. The matrix size is fixed to 4×4. Average Latency is measured with the right Y-axis.

Figure 6.1 (a) shows how the average latency and the resource consumption scale linearly as the matrix size increases while keeping a fixed datum's width in 8 bits. In this case, none of the solutions consumes Block Random Access Memory (BRAM) or Digital Signal Processor (DSP) cells. The consumption of 2×2 s below 2.5% of the FPGA. In the case of a 16×16 PE, it consumes up to 45% of the FPGA's available resources. Having small PEs is beneficial in terms of resource utilisation, offering more granular execution and having a small footprint on the resources. With the vectorisation capability explained in 3.4, accelerators can process matrices with one of the dimensions larger than the other, i.e. more rows than columns. It makes better use of the resources using a *divide-and-conquer* approach.

Moreover, Figure 6.1 (b) shows how the resource consumption and latency scale while modifying the width of the data in bits. Flip-Flops (FFs) scaling tends to scale linearly for all widths. However, when the PE uses more than 8-bits, the overall resource consumption jumps from 3% to 7%, keeping the total consumption fixed to 7% from 10 bits to 16 bits.

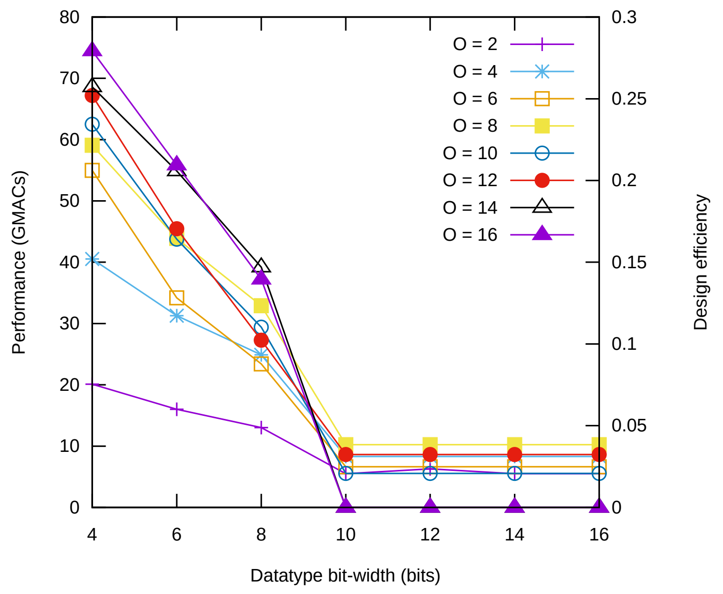


Figure 6.2: General matrix multiply-add performance and design efficiency with respect to the data width in bits and the output size $O \times O$. The data type is a fixed point with B bits (1-bit integer and $B-1$ fractional bits). The performance and efficiency are computed using Equations (6.3) and (6.4). The design efficiency is proportional to the performance and a point in the plot matches both Y-axis.

There is also a jump of one clock cycle in the average latency. It means that, for a 4×4 matrix, it makes sense to have less than 9 bits. Then, there is no benefit in varying data widths from 10 up to 16 bits since the overall consumption keeps constant in the Zedboard. However, this jump does not apply to all FPGAs. Later sections will look deeply into this aspect.

Figure 6.2 shows the performance and the design efficiency of the PE when using several configurations of data widths and output matrix size. It highlights that *short data widths and big output matrices are better in efficiency*, reaching up to 75 out of 276 GFLOPs (27% of efficiency).

After joining the results obtained in Figure 6.1 with the ones presented in Figure 6.2, this research suggests that using small matrices leads to low resource consumption. Nonetheless, having small PEs leads to suboptimal design efficiency, according to Figure 6.2, demonstrating a trade-off between *granularity vs efficiency*.

Regarding data widths, short data widths lead to better design efficiency and better performance. It is due to the capability of allocating more PEs in the FPGA, increasing the potential parallelism. Figure 6.1 (b) shows that after 8 bits, the overall resource consumption is constant for any datatype from 10 to 16 bits for a fixed matrix size. According to Figure 6.2, using big matrices is impacted by this jump in consumption when exceeding more than 8 bits, showing a drop from 15% to less than 1% (15x degradation). Interestingly, small matrices are penalised in efficiency for short data widths, but it is compensated in large data widths, showing degradation of roughly 3x. It demonstrates

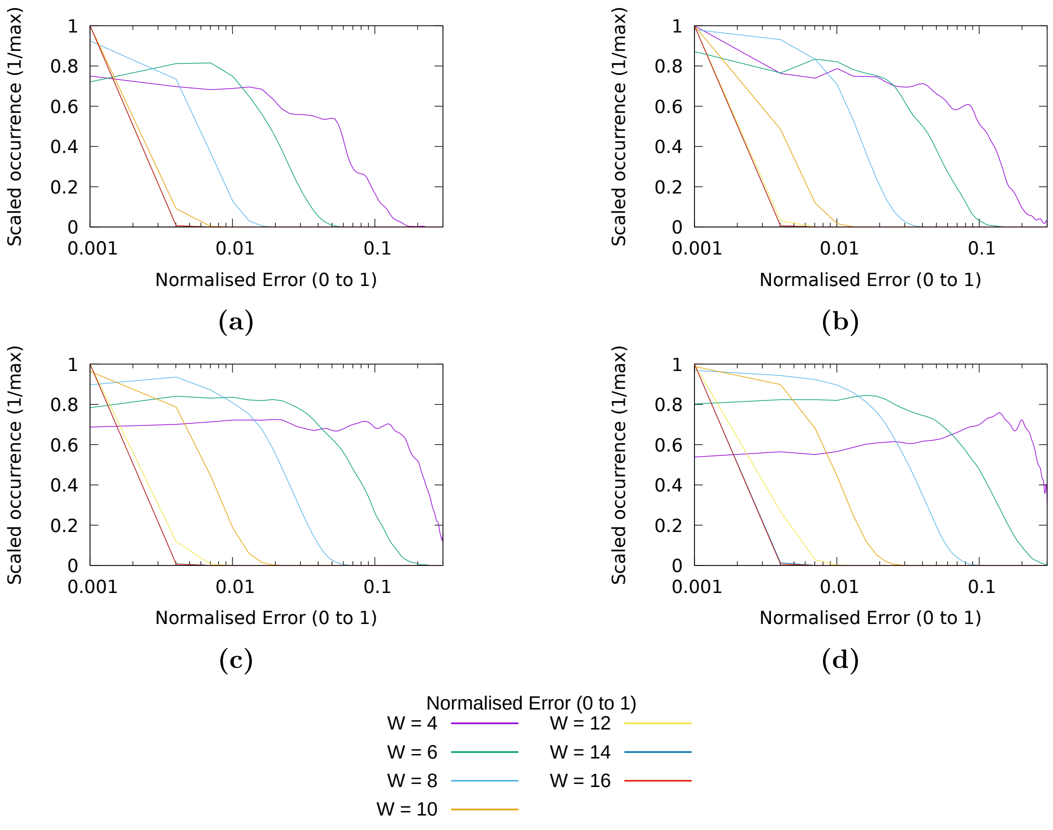


Figure 6.3: Error histogram of the matrix multiply-add PE when varying data width from 4 up to 16 bits. Each histogram group (subfigure) takes into account different matrix sizes. The histograms have been normalised, taking into account the maximum value of the distribution for better visualisation. The error (x-axis) is measured by using (6.5). The error has been truncated to 30% and data has been smoothed using bezier, chopping some of the maximum values that reach 1. Matrix sizes: **(a)** 2x2, **(b)** 4x4, **(c)** 6x6, **(d)** 8x8

that it is more convenient to use big matrices for short data widths (≤ 8 bits). For large data widths, having a matrix size of 8×8 is the best choice. Hence, for PEs with ≤ 8 bits of numerical precision, the *granularity vs efficiency* must be taken into account. For data widths ≥ 8, Pareto's optimal solution is 8×8, and the trade-off from before seems not to be longer applicable in the implementation.

Error Quantification

Since the PE uses custom-precision fixed-point representation, the goal is to have an error characterisation that allows knowing how the error distributes given a set of random matrices. For the experiment setup, this error analysis presents the result differences of 500 sets of operands randomly for each data width for a single-precision floating-point implementation.

Figure 6.3 depicts the error distributions when varying the data width from 4 to 16 bits and the output matrix size from 2 to 8. The normalised errors (X-axis) are computed using (6.5), and the number of occurrences is divided by its maximum to lead to getting all the distributions within the same scale for visualisation purposes. The error distribution tends to be Gaussian distributed with zero mean (recalling to the law of large numbers).

In all the matrix size cases, the behaviour of varying data width is similar; increasing the number of bits of the data representation reduces the variance of the error distribution, shrinking the distribution around 0. For instance, a 4-bit PE reaches a maximum error of 23%. Instead, 6-bit reduces the error up to 6% (73% improvement). In the case of 8×8, the error exceeds 30% for a 4-bit PE, and it is close to 30% for a 6-bit PE.

It also suggests that the matrix size impacts the error because of the scaling of the operands: as the number of rows increases, the operands become smaller, as presented in (3.2), leading to *operands vanishing*. For an 8-bit PE, the maximum error is about 2%, 3%, 6%, and 8% for 2×2, 4×4, 6×6 and 8×8, respectively. Hence, there is a compromise between the matrix size and data width vs the error in the matrix multiply-add implementation. It is possible to mitigate the effects of the matrix size on the error by increasing the data width to compensate for the variance of the normalised error. Thus, the trade-off at the error level is described as

$$\mathrm{Var}\left[\epsilon_{ij}^{(p)}\right] \propto \phi(N_O, \frac{1}{N_W}) \tag{6.10}$$

where N_W is the data width in bits, N_O is the number of rows/columns of the output matrix, and $\phi(\cdot)$ is a function that describes how the values scale positively up as its arguments increase. For instance, for a single argument $\phi(x)$, $\phi(x/2) \leq \phi(x) \leq \phi(2x)$. The analytical form of this function will be addressed in future work.

In summary, the resource consumption scales linearly as the matrix size N_O increases and directly depends on the data width N_W. There is an inflexion point where the consumption becomes fixed after a data width of 8 bits, leading to an equal overall resource consumption from 10 up to 16 bits. In terms of efficiency, there is a trade-off between granularity and efficiency, where the design efficiency favours small output matrices. It is shown that, after 8 bits, the design efficiency converges to the same value, given that the overall consumption gets constant and the number of operations does not increase.

Remarks on acceleration. The limitation of using Equation (3.2) to prune the bits of the results is manageable at large-scale levels. The PEs are intended to be integrated into vector units through a vector wrapper explained in Algorithm 5. For computing the GEMMA of two matrices, the accumulation capability is heavily exploded and requires an expansion in the numerical range of the operation. Further sections evaluate the affectation of bit pruning in large matrices. Still, the effects on actual neural networks should be studied in future work. As a first hint, the error mitigation might require fine-tuning training to fit the model into the hardware to absorb the error characterisation of the accelerator [46].

6.2.2 Window-based Convolution

This section analyses the effect of varying the DoF values on the window-based spatial convolution and the Winograd convolution design performance. The analysis integrates an optimised version of the spatial convolution for every configuration and the Winograd convolution for $N_k = 3$. The effect of the arithmetic operators will be out of the scope of this document and will lead to future work. Other configurations of Winograd presented in this work are suboptimal and will be addressed in future work.

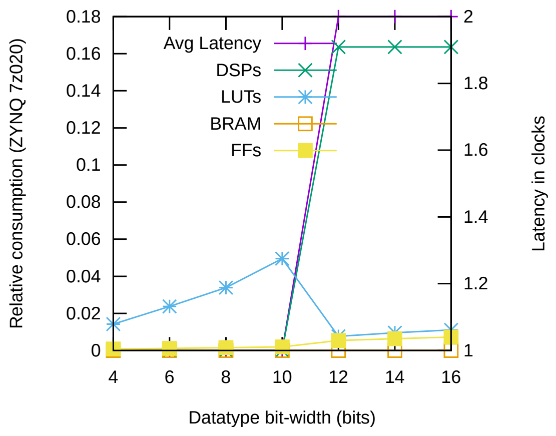
Resource Consumption and Performance

Similar to 6.2.1, this work evaluates the resource consumption, the average latency and the efficiency of the implementations while varying data widths from 4 to 16 bits in steps of two, the output sizes, and kernel sizes with $N_K \in \{3, 5, 7\}$.

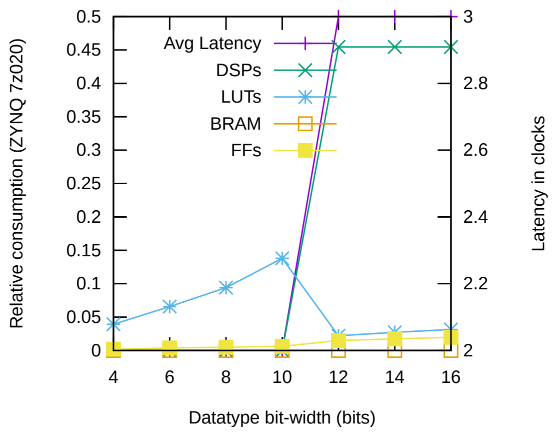
Figure 6.4 illustrates the behaviour of the window-based spatial convolution and the Winograd convolution. In spatial convolution, the average latency tends to be constant with a jump of a clock cycle when reaching more than 10 bits in the data width, combined with a jump in the DSP cell consumption. This behaviour is also observed in the generic matrix multiply-add PE when reaching more than 8-bits. The scaling is linear when the data width is less or equal to 10 bits. After that, the overall consumption is fixed to a value; 3×3 kernels is about 16%, 5×5 45%, and 7×7 90% of the FPGA resources. The overall consumption for all cases after the jump is approximately three times the maximum consumption when using 10 bits. From these observations, it is possible to confirm that: 1) *the data width does not have a significant impact on the average latency*, and 2) *after 10 bits, the overall consumption makes a transition from linear scaling to a fixed constant value that is* $3\times$ *larger than the 10-bit PE consumption.*

Figure 6.5 shows the consumption of the window-based spatial convolution PE while varying the matrix size from 2 to 8 and keeping the data width to 8 bits. It has been chosen 8-bits since it is one of the lowest data representations in host systems (1 byte). Likewise, it is under the 10-bit threshold before the resource jump presented in Figure 6.4. The average latency keeps constant at 1 for a 3×3 kernel, 2 for 5×5, and 3 for 7×7. Hence, the average latency presents a dependency on the kernel size. On the other hand, the output window size does not influence the average latency for a fixed data width and kernel size. In terms of resources, increasing the output window size leads to a quadratic scaling in look-up tables, exceeding the FPGA resources in $N_K \in \{5, 7\}$ when using 8×8 output windows. Hence, 3) *keeping the output window small will help to keep the resource consumption low, and it will not impact the average latency of the PE.* The kernel size, instead, also has a non-linear scaling according to the observations in Figure 6.5.

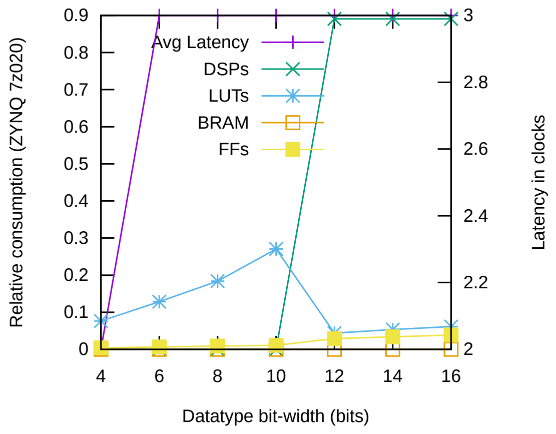
Regarding the design's performance and efficiency, Figure 6.6 illustrates the results for the window-based spatial and Winograd convolution for several data widths, kernel sizes and output sizes. For a 3×3 kernel, Winograd shows the best performance, achieving a



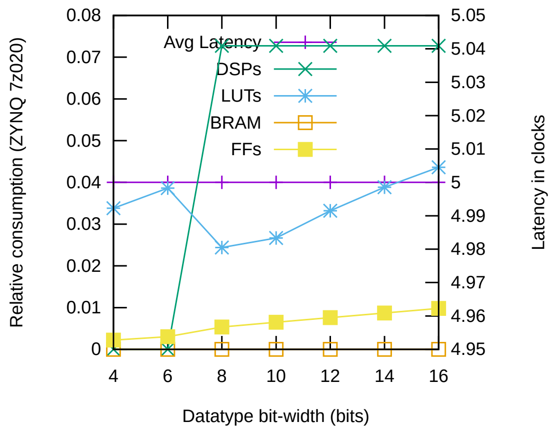
(a)



(b)



(c)



(d)

Figure 6.4: Resource utilisation of the convolution PEs referred to an Avnet Zedboard with a Xilinx XC7Z020 FPGA-based System-on-Chip. This includes an analysis of the data width impact on the resources and average latency for the Spatial convolution with kernels 3×3, 5×5, 7×7, and Winograd with a 3×3 kernel. Other kernel sizes are not included in Winograd because of suboptimality. Average Latency is measured with the right Y-axis. The output window is 2×2. **(a)**, **(b)**, **(c)** are the results for the Spatial Convolution with 3×3, 5×5, 7×7, and **(d)** contains the results for Winograd and a kernel size of 3×3

design efficiency of up to 47%, followed by its equivalent for the spatial convolution for a 2×2 output window with 40%. TThe efficiency and performance decay as the data width increases since more resources are required, and the number of operations is still the same. Besides, the performance and efficiency get constant from 8 bits and 12 bits for the case of Winograd and the spatial convolution, respectively. The constant behaviour is due to the overall consumption observed in the resource utilisation after the jump in DSP cells consumption, observed in Figure 6.4. For the output window size, the efficiency of Winograd at 4-bits drops from 47% to approximately 10%. It shows that resource consumption has a more significant effect than the scaling of the number of operations

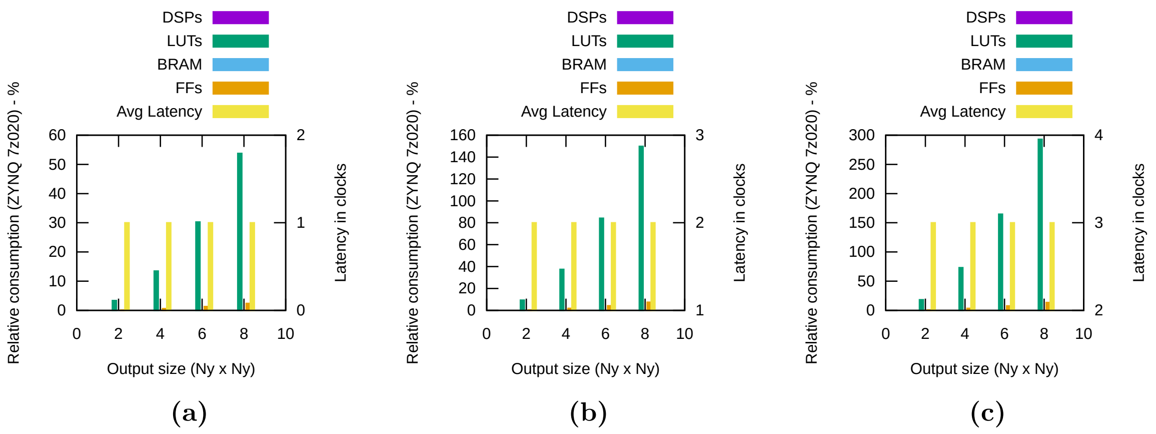


Figure 6.5: Resource utilisation of the window-based spatial convolution PE. Each plot represents the relative consumption vs the output window size ($N_K \times N_K$) for different kernels and data width of 8 bits. Winograd is skipped because it does not support variable window sizes for now. The consumptions are with respect to an Avnet Zedboard with a Xilinx XC7Z020 FPGA-based System-on-Chip. Average Latency is measured with the right Y-axis. Kernel sizes: **(a)** 3×3, **(b)** 5×5, and **(c)** 7×7.

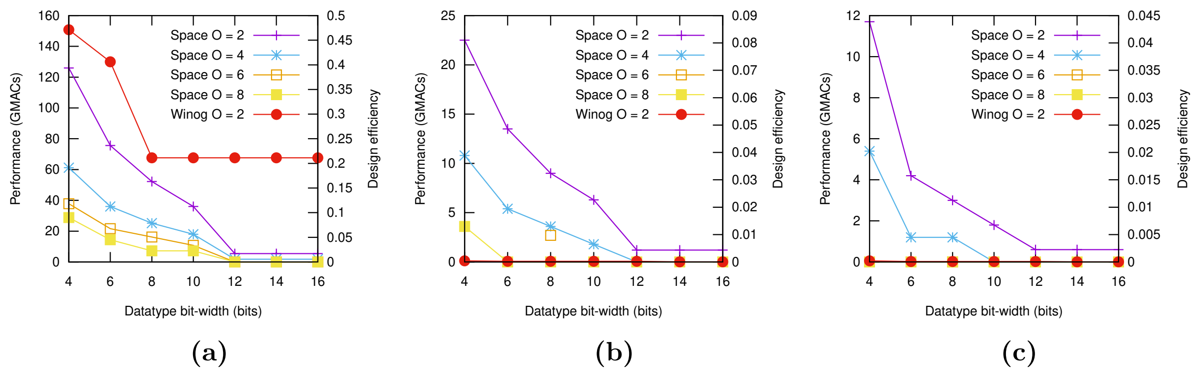


Figure 6.6: Windows-based spatial (Space) and Winograd (Winog) convolution performance and design efficiency with respect to the data width in bits (X-axis), the output size (O, series), and the kernel size (subfigures). The datatype is a fixed-point with B bits (1-bit integer and $B-1$ fractional bits). The performance and efficiency are computed using Equations (6.3) and (6.4). The design efficiency is proportional to the performance and a point in the plot matches both Y-axis. Kernel sizes: **(a)** 3×3, **(b)** 5×5, and **(c)** 7×7.

performed by the PE.

Moreover, when varying the kernel size, the Winograd convolution underperforms due to the lack of optimisation at the logic level. It will be the subject of future work. Nevertheless, the kernel's behaviour also demonstrates a degradation in the design efficiency in spatial convolution. For $N_K = 3, N_Y = 2$, the efficiency reached approximately 40%, for $N_K = 5$ 8%, and 4.4% for $N_K = 7$. It also suggests that kernel size scaling involves a greater growth in resource consumption compared to the number of operations performed by the PE.

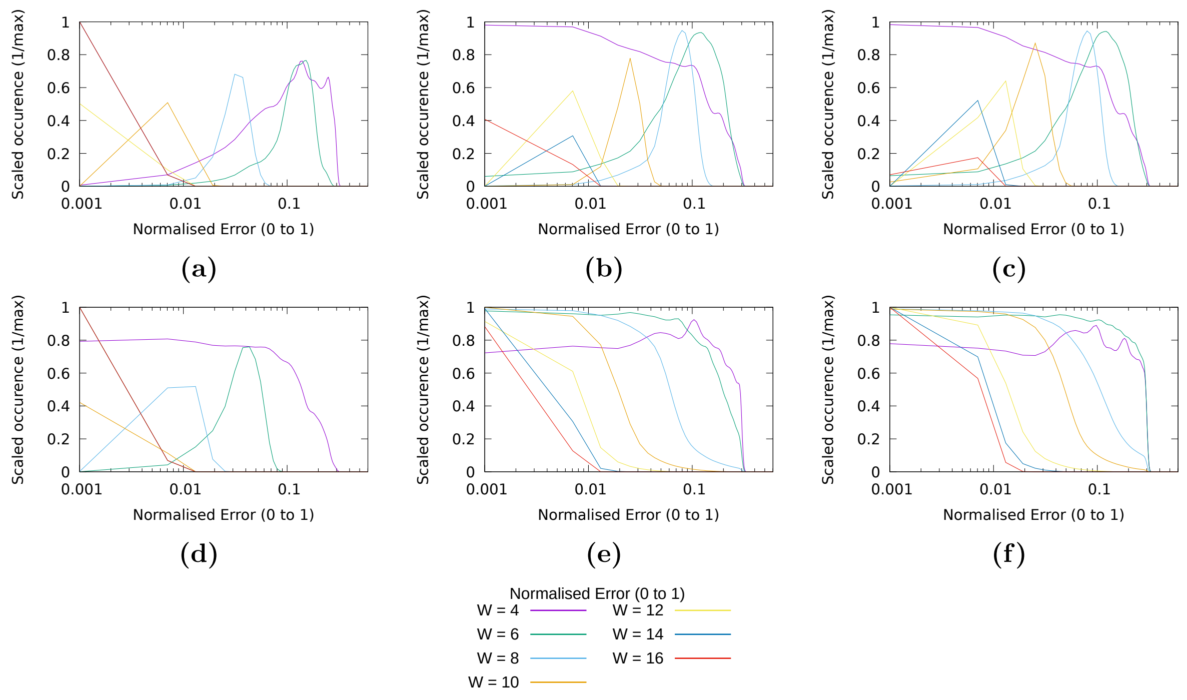


Figure 6.7: Error histogram of the convolutional PEs when varying data width from 4 to 16 bits. Each histogram group (subfigures) takes into account different matrix sizes. The histograms have been normalised using the maximum value of the distribution for better visualisation. The error (x-axis) is measured by using (11). The error has been truncated to 30% and data has been smoothed using bezier, chopping some of the maximum values that reach 1. **(a, b, c)** correspond to the spatial convolution with kernels 3×3, 5×5, and 7×7, and **(d, e, f)** correspond to the Winograd convolution with kernels 3×3, 5×5, and 7×7.

The effects mentioned above affirm that *keeping the output window as small as possible* contributes to better performances and low resource consumption. Large data widths do not cause degradation in efficiency or performance. This is proven by choosing a data width between 12 to 16 bits for the case of the spatial convolution and from 8 up to 16 bits for the case of Winograd in $N_K = 3$. Another remark is that 4) *it is better to fit the required PE size in terms of kernel size, preferring the smallest ones.*

Error Quantification

The convolution PEs extensively use custom-precision fixed-point representation. Unlike the matrix multiply-add, the Winograd implementation has internal buffers twice the data length. It resulted in being more sensitive to quantisation when implementing the generic design. The aim is also to see how the error of the results behaves after the quantisation and processing.

The experimental setup evaluates four well-known test grayscale images of 512×512: Baboon, Lenna, Barbara, and Peppers, evaluating the numerical error per pixel (262144 in

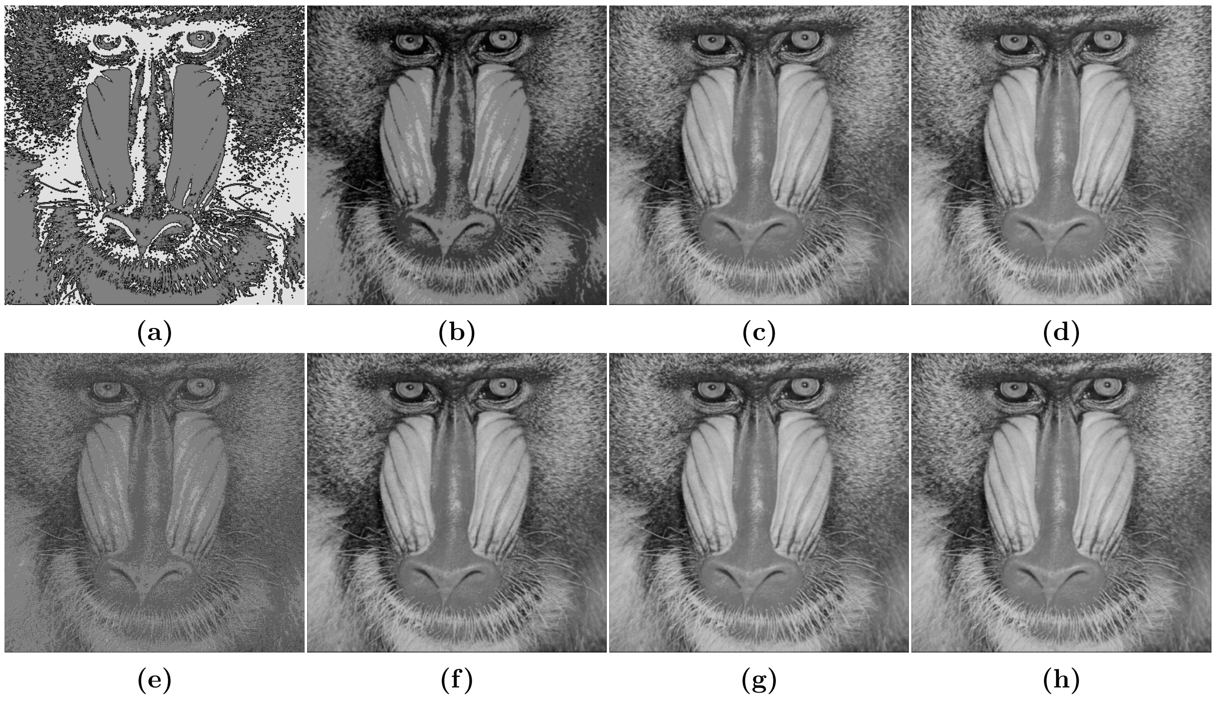


Figure 6.8: Qualitative evaluation of the baboon test image after a convolution with a 3×3 Gaussian kernel for data widths $N_W \in \{4, 6, 8, 16\}$. The Peak Signal-to-Noise Ratio (PSNR), Structural Similarity Index (SSIM), and Root Mean Squared Error (RMSE) are specified per image in Table 6.2.

Table 6.2: Quality descriptors for the samples presented in Figure 6.8

Figure	**Algorithm**	**Data width**	**PSNR**	**SSIM**	**RMSE**
(a)	Space	$N_W = 4$	PSNR: 5.89dB	SSIM: -0.617	RMSE: 0.444
(b)	Space	$N_W = 6$	PSNR: 15.85dB	SSIM: 0.922	RMSE: 0.141
(c)	Space	$N_W = 8$	PSNR: 28.01dB	SSIM: 0.996	RMSE: 0.035
(d)	Space	$N_W = 16$	PSNR: 52.22dB	SSIM: 0.999	RMSE: 0.002
(e)	Winograd	$N_W = 4$	PSNR: 16.28dB	SSIM: 0.731	RMSE: 0.134
(f)	Winograd	$N_W = 6$	PSNR: 23.53dB	SSIM: 0.974	RMSE: 0.058
(g)	Winograd	$N_W = 8$	PSNR: 38.91dB	SSIM: 0.999	RMSE: 0.010
(h)	Winograd	$N_W = 16$	PSNR: 52.22dB	SSIM: 0.999	RMSE: 0.002

total per image) using (6.5) and filtering the errors above of 30% for practical reasons. This study aims to determine the error distribution according to PE DoF values to examine how the parameters affect the numerical results of PE.

Figure 6.7 illustrates the error distributions. The errors in convolution are invariant to the output size. The figure shows the error distribution for the window-based spatial convolution. In this case, the mean error and the variance increase are inversely dependent on the data width. For $N_W = 4$, the maximum mean error is around 38.2%, with a significant standard deviation of 22%. Increasing the data width by 2 bits leads to an improvement in the error, resulting in a mean error of 13.3% and a standard deviation of

4.6%. There is an influence of the kernel size on the error, in particular, from $N_K = 3$ to $N_K = 5$. The influence from $N_K = 5$ to $N_K = 7$ does not significantly affect the mean and the variance of the distributions, even for the shortest data width. For $N_W = 4$, the mean evolved from 12.768% to 12.782%, leading to 0.014%, and the standard deviation differs by 0.012%.

Winograd, instead, presents a better error distribution than spatial convolution. The error is more compressed to the left, making it closer to zero and suggesting no bias to a particular error value as in spatial convolution. For $N_K = 3, N_W = 4$, the mean is 11.01%, leading to a reduction of the error of 27.2% for the former implementation. The kernel has a more apparent influence on the distribution variance, increasing as the kernel size grows. For $N_W = 8$, the standard deviations for $N_K = 3$, $N_K = 5$, and $N_K = 7$ are 0.301%, 8.940%, and 11.204%, affirming the dependence of the variance on the kernel size.

Figure 6.8 and Table 6.2 show the Baboon test image convolved against a 3×3 Gaussian kernel using the spatial and Winograd convolution PEs configured to work with data widths of $N_W \in \{4, 6, 8, 16\}$. The artefacts found when using short data widths are different between PEs. In the spatial convolution, there is a saturation of the values, where the darker zones become lighter zones, suggesting possible under-flowing. In the Winograd case, the image looks more legible but with different tones of grey with respect to the actual convolution result.

The PSNR and SSIM metrics help to illustrate the signal degradation in both PEs. In spatial convolution, the SSIM becomes negative. Given that the data are always positive, the means and standard deviations $\mu_x, \mu_y, \sigma_x, \sigma_y$ are also positive. However, the covariance σ_{xy} can be negative if the data become inverted. It means that the low values become high and vice-versa, as in Figure 6.8(a). It can be caused by under-flowing effect and highlights the dissimilarity between the experimental result and the reference. The PSNR also highlights the poor signal quality obtained after applying the PE with 4-bit data width. Winograd, instead, presents an SSIM of 0.731 and a PSNR of 16.28 dB. It shows that the quality of the signal increases. This can be linked to Figure 6.7, where the distribution for 4 bits in Winograd is better than the spatial PE. In $N_W = \{6, 8\}$, Winograd outperforms the spatial convolution, showing higher signal qualities and better SSIMs. For $N_W = 6$, the SSIM reaches 92.2% of similarity in spatial and 97.4% in Winograd. For $N_W = 8$, the differences are less than 1%, where both algorithms start to converge in quality. In both cases, it demonstrates the robustness of Winograd in numerical errors and how the error distributions presented in Figure 6.7 behave qualitatively.

In summary, *Winograd outperforms in terms of efficiency and error to the spatial convolution* for $N_K = 3$ with mean error differences of up to 27.2% compared to the spatial convolution in 4-bit data width. In terms of scaling in resource consumption and its trade-offs, the data width does not significantly impact the average latency for both implementations. Moreover, after 8-bits in Winograd and 10-bits in spatial, the overall consumption makes a transition of linear scaling with respect to the data width to a constant value given by DSP cells, fixing the design efficiency to a constant value, allowing greater data widths without a direct influence on the scaling up to 16 bits. Regarding the output window size, the conclusion is that the design efficiency degrades as the size increases, preferring smaller window sizes to save resources. In terms of kernel size, it is recommended to fit it according to the model requirements, avoiding clearance due to resource consumption scaling. It is possible to conclude that the resource consumption is given by

$$r_p \propto N_W N_Y^2 \rho(N_K) \tag{6.11}$$

where $\rho(\cdot)$ is the function describing the kernel size scaling, which will be analysed in future work and the error

$$\mathbb{E}[\epsilon_{ij}^{(p)}] \propto \psi(N_K, \frac{1}{N_W}), \mathrm{Var}[\epsilon_{ij}^{(p)}] \propto \phi(N_K, \frac{1}{N_W})$$

The mean value has an incremental dependence on the inverse of the data width (N_W) and the kernel size (N_K). Winograd has less influence than spatial, and the variance has incremental dependence on the kernel size, and inverse dependence on the data width. Winograd presents the most significant dependence on the N_W and N_K than spatial, where this last one has less influence on N_K. $\psi(\cdot)$ and $\phi(\cdot)$ are functions that scale positively up the values as their arguments increase, and they are not necessarily the same for all the architectures (i.e. matrix multiply-add, Winograd, and Spatial convolution). These functions will be addressed in future work.

6.3 Accelerator

6.3.1 Generic Matrix Multiply-Addition

The analysis of the Generic Matrix Multiply-Addition implies the exploration of the data width, the width of the integer part in the case of fixed-point numbers, the PE matrix size, and the input matrix size. This analysis will address the input matrix size under two possible FCL configurations: 120×10 and 400×120.

This work only considers accelerators based on the GEMMA PE and exact fixed-point arithmetic. The approximate operators are out of the scope of this analysis and will lead to future work.

The metrics utilised for the analysis include:

- **Resource consumption**
  - Relative consumption of BRAM, FFs, LUT, and DSPs in a ZYNQ XC7Z020 with respect to the data width, integer width, and PE matrix size.

  - Maximum consumption of these components.
  - Resource consumption of the template without taking into account the execution module.

Solution Selection

Subsection 4.3.4 presented three possible implementations of the GEMMA accelerator: (1) serial, (2) hybrid, and (3) pipeline. This part presents the optimisations performed during the accelerator implementation, considering a balance between resource consumption and latency. Dataflow architecture was unsuitable because of data feedback (or dependency).

For this case, the following DoF values are fixed:

- `Q_INT` (integer part): 2-bits
- `Q_O` (output matrix size): 2×2
- `Q_PES` (number of PEs): 4
- `CLOCK_PERIOD`: 7.5 ns target.

In terms of the data width, two values are used for comparison: 8 and 16 bits. The solution optimisations are the following:

- **Solution 0: Serial**
  - None (defaults).
- **Solution 1: Hybrid - Only execute module optimisation**
  - Pipeline the execute module only.
  - Unroll inner loops at the PE level (recalling the baseline GEMMA algorithm).
- **Solution 2: Hybrid - Partition arrays**
  - Include solution 1.
  - Partition interfaces: all interfaces are partitioned completely into registers.
  - Partition cache buffers: all the matrix buffers (or caches) are implemented as registers.
- **Solution 3: Hybrid - Partition arrays**
  - Include solution 2.
  - Pipeline data retrieval functions.
  - Inlining to matrix receptors (in case of the inputs).

Table 6.3: GEMMA solution optimisation results. The consumptions are with respect to the ZYNQ XC7Z020. There are multiple configurations for the hybrid architecture, and each solution is incremental. "L" stands for Look-up Tables, "F" for Flip-Flops, "B" for BRAM, "D" for DSPs, and "DW" for Data Width

Solution	DW	Architecture	Consumptions	Latency
0	8	Serial	B: 0(0%) F: 1561(1%) L:4091 (7%) D: 0(0%)	598 cycles
0	16	Serial	B: 0(0%) F: 1727(1%) L:4071 (7%) D: 1(0%)	646 cycles
1	8	Hybrid	B: 3(1%) F: 1811(1%) L:5028 (9%) D: 0(0%)	407 cycles
1	16	Hybrid	B: 1(0%) F: 2397(1%) L:4797 (9%) D: 8(3%)	427 cycles
2	8	Hybrid	B: 0(0%) F: 5758(5%) L:10903 (20%) D: 0(0%)	395 cycles
2	16	Hybrid	B: 0(0%) F: 9114(8%) L:11017 (20%) D: 8(3%)	411 cycles
3	8	Hybrid	B: 0(0%) F: 5456(5%) L:8218 (15%) D: 0(0%)	29 cycles
3	16	Hybrid	B: 0(0%) F: 9341(8%) L:8960 (16%) D: 8(3%)	37 cycles
4	8	Hybrid	B: 0(0%) F: 3799(3%) L:5067 (9%) D: 0(0%)	12 cycles
4	16	Hybrid	B: 0(0%) F: 7604(7%) L:4839 (9%) D: 3 (14%)	20 cycles
5	8	Pipeline	B: 0(0%) F: 3405(3%) L:5105 (9%) D: 0(0%)	10 cycles
5	16	Pipeline	B: 0(0%) F: 6826(6%) L:4877 (9%) D: 3 (14%)	18 cycles

- **Solution 4: Hybrid - Vectorisation**
  - Include solution 3.
  - Vectorise PE units.
- **Solution 5: Pipeline**
  - Include solution 4.
  - Pipeline at the core level (top module).

Table 6.3 shows the results for the optimisations intended for GEMM. The serial implementation has the worst latency but the best area usage, expected for a serial architecture with high hardware reuse. Optimising only the execute stage is not enough to bring the latency down (solution 1 and 2); however, adding optimisations to the other static modules brought the latency down by 11×, suggesting that the most critical modules are the static ones. Solution 4 vectorises the PEs, optimising the execution module to execute the PEs concurrently, speeding up by 1.85 times compared to solution 3. Finally, moving to a full pipeline architecture only improved the solutions by two clock cycles with a slight increment in LUTs and decrement in the FFs. Solutions 4 and 5, hence, achieve the best trade-off between resources and latency. It also suggests that the static modules are compulsory targets during the optimisation and can lead to future work in optimisation. Moreover, they are sensitive to data transfers, leading to increased latency when modifying the operands and the data type DoF.

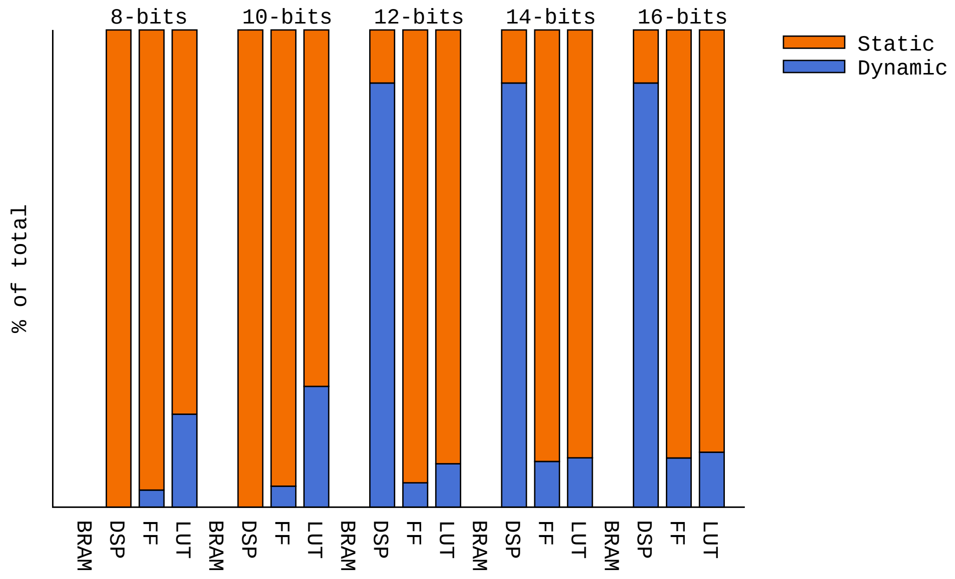


Figure 6.9: Dynamic part (PE) vs the total accelerator usage. After 12-bits, the utilisation becomes constant in terms of overall consumption. The configuration belongs to a single-core accelerator with a 2×2 PE.

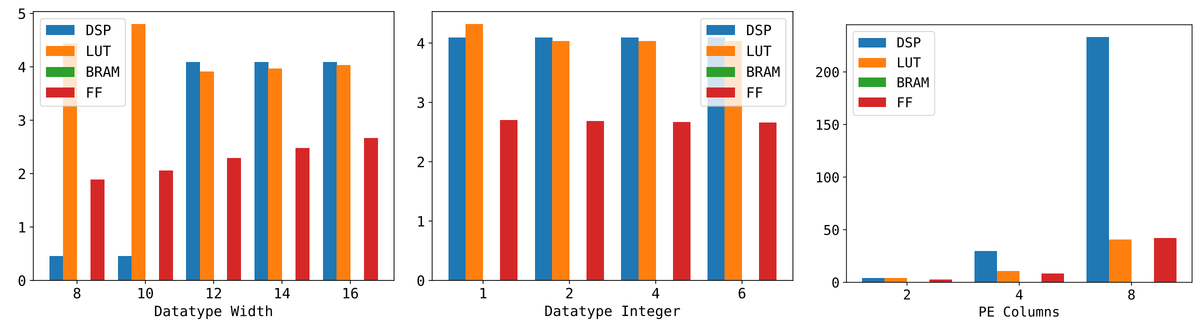
Resource Consumption

According to 4.3, the GEMMA accelerator is composed of static and dynamic parts. The dynamic part changes according to the PE implementation. However, the static part should only adapt according to the data representation details, such as data width, number of PEs and PE matrix size. Figure 6.9 summarises the proportion of accelerator usage in terms of the static and dynamic parts. For accelerators with a data width of fewer than 12 bits, the consumption keeps less than 25%.If the width exceeds 12 bits, the DSP becomes the highest utilisation, reaching more than 90% of the accelerator. Nevertheless, in terms of LUTs and FFs, the utilisation is less than 15%. It suggests that the template takes most of the LUTs and FFs of the accelerator due to the FF-based registers used for implemented arrays partitioned completely and the implementation of the communication interface.

After knowing the composition of the accelerator, it is time to evaluate the affectation of the DoF on the overall resource consumption. Figure 6.10 shows the affectation of the data width (Figure 6.10(a)), integer width (Figure 6.10(b)), and the PE matrix size (Figure 6.10(c)) on the consumption of resources. Using a fixed-point unit with a 4-bit integer, varying the total data width leads to 5% of LUT utilisation, whereas the FF is about 2%. A data width with less than 12 bits leads to a linear scaling at the PE level, as demonstrated in Figure 6.1(b). The accelerator template keeps similar in size due to the number of packages. For storing 8-bit matrices in a single core accelerator, it requires $3 \times N_R \times N_C \times N_W = 3 \times 2 \times 2 \times 8 = 96$ registers, where 3 is the number of matrices and $N_R \times N_C$. For 10 bits, it requires 120 registers, which is negligible compared to the total number of FFs in the ZYNQ XC7Z020 (106400). However, after 12-bits, the LUT utilisation decreased because the tool started to implement the arithmetic using DSP. The FF consumption still scales at the same rate, similar to the PE resource consumption. Thus, it is possible to notice that the behaviour is the same as presented by the PE in Figure 6.1(b).

Figure 6.10(b) shows the affectation of varying the integer part while keeping the data width at 16 bits. The behaviour remains the same for any data width from 2 to 6 bits without significantly affecting resource usage due to the nature of fixed-point, where the values are subject to interpretation and arithmetically operated seamlessly.

The affectation while varying the PE matrix size is more severe than the former DoF variations. Figure 6.10(c) summarises this behaviour, showing that the consumption scales at the same rate as the number of elements scales. This is still similar to the observations from Figure 6.1 at the PE level.



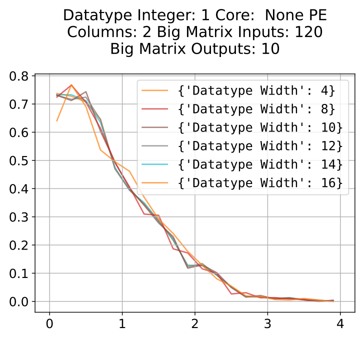
(a) (b) (c)

Figure 6.10: Consumption report of the GEMMA accelerator while varying data type width, integer width and the matrix size of the PE in a single-PE accelerator. The X-axis refers to the values of the DoF and the Y-axis to the FPGA resource consumption (%). The number of PEs is one core. (a) varies the data width while keeping the PE matrix size to 2×2 and the integer part to 4. (b) varies the integer part while keeping the data width to 16 bits and the PE matrix size to 2×2, and (c) varies the PE matrix size (square matrix) while keeping the data width at 16 bits and the integer part at 4 bits.

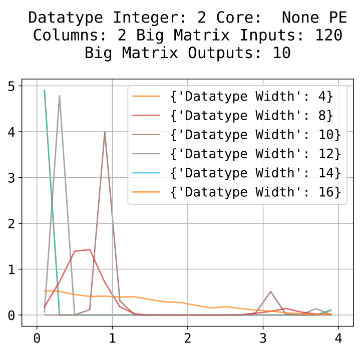
Hence, the resource consumption analysis determines that the behaviour is close to the observed in the PE resource consumption analysis, presenting the same behaviour even though the template adds resource overhead due to caching and protocol implementation. The performance analysis will be done after the error evaluation in order to discard candidates in the DSE.

Error Quantification

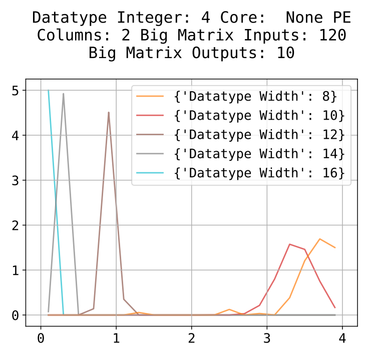
The error evaluation will take into consideration matrices with a similar size as the LeNet-5: 400×120 and 120×10 with elements between -0.5 and 0.5. This will provide a similar scenario as in small Neural Networks. Moreover, it will consider the DoF that can cause



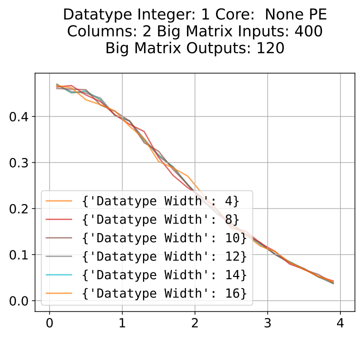
(a)



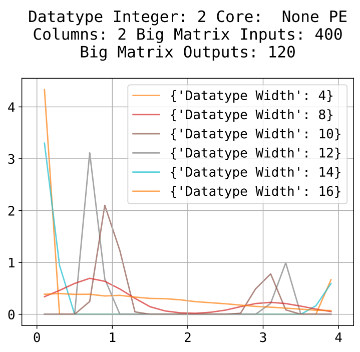
(b)



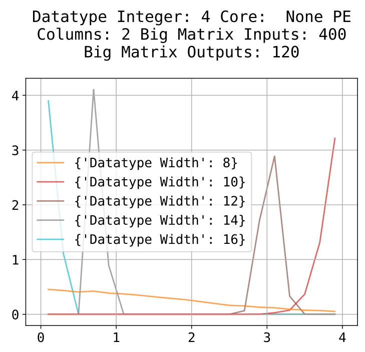
(c)



(d)



(e)



(f)

Figure 6.11: Error histogram of the matrix multiply-add accelerator when varying data width from 4 up to 16 bits. Each histogram group (subfigure) takes into account different matrix sizes (first row corresponds to a matrix of 120 $\times$ 10 elements and the second to a matrix of 400 $\times$ 120 elements). Each column, instead, varies the integer part (1, 2, and 4) The Y-axis is not normalised and allows showing the shape of the error distribution. The normalised error (X-axis) is measured by using (6.5); exceeding 1, the errors reach more than 100%.

error degradation, such as the data type width and integer bits. The PE matrix size will be fixed to 2 $\times$ 2 given that it offers the best trade-off from DSE performed on the PE analysis presented in 6.2.1.

Figure 6.11 shows the error histograms in the accelerator in configurations with 1, 2, and 4 integer bits and for the two matrices considered during this analysis. In this case, since the matrices are bigger than the ones presented the PE analysis, low data widths are not suitable in this context because of the bit pruning. Moreover, the integer part has increased to reduce the error because of overflows in the accumulation. For the 1-bit integer part (Figure 6.11(a) and Figure 6.11(d)), none of the data widths were suitable for getting errors below 100%, suggesting substantial changes with respect to the golden data. The best cases started to appear when increasing the bits of the integer part to more than 2 bits. In this case, assuming that an accelerator is usable for mean errors less than 30%, the best options have more than 14 bits in data width with more than 2 bits in the integer part.

Continuing with other metrics, Figure 6.12 shows the PSNR in the same fashion as in

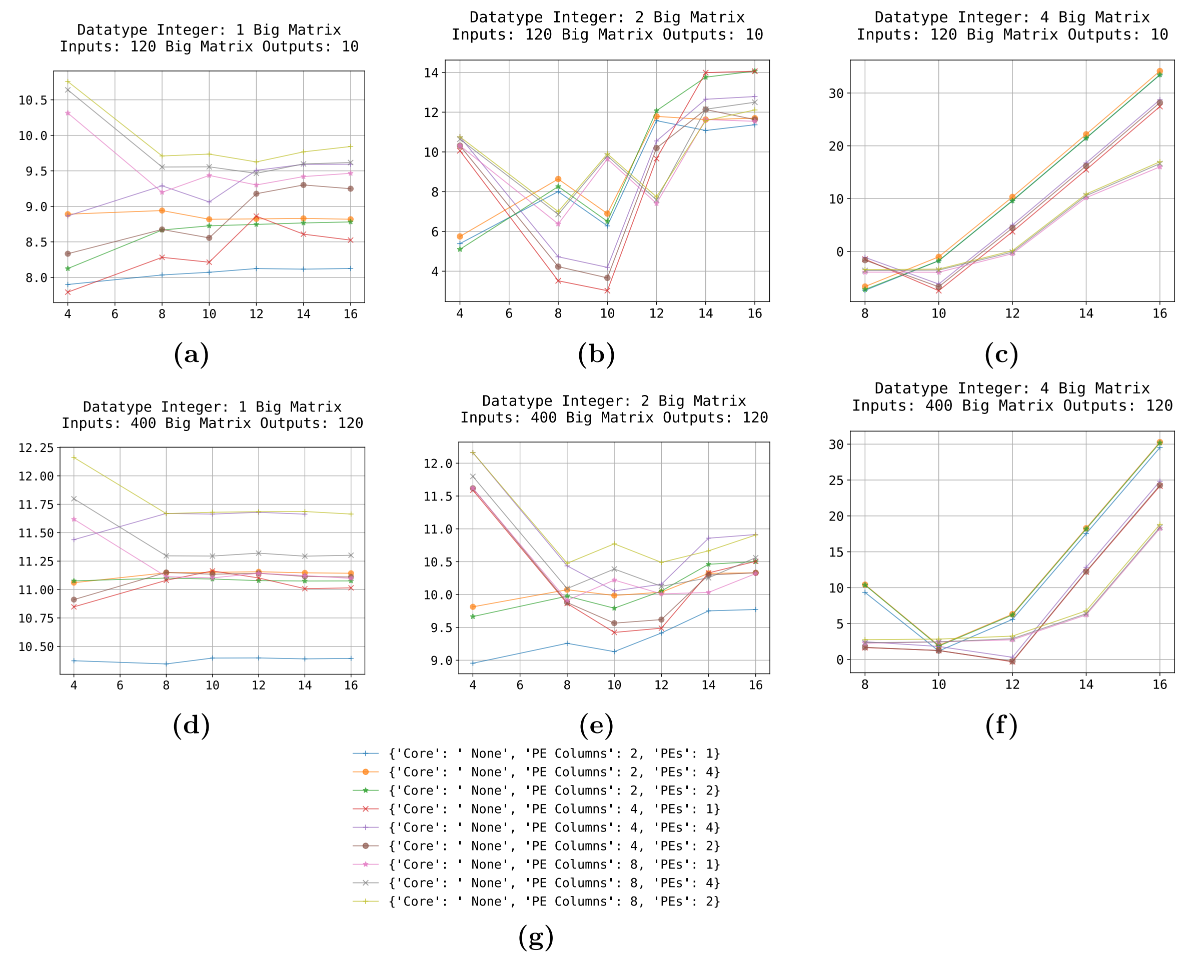


Figure 6.12: PSNR-based quality plots of the matrix multiply-add accelerator when varying data width from 4 to 16 bits. Each subfigure considers different matrix sizes (first row corresponds to a matrix of 120 × 10 elements, and the second to a matrix of 400 × 120 elements). Each column, instead, varies the integer part (1, 2, and 4) The X-axis shows the data width, Y-axis the PSNR value in [dB], and the series are PE configurations (matrix size and number of PEs).

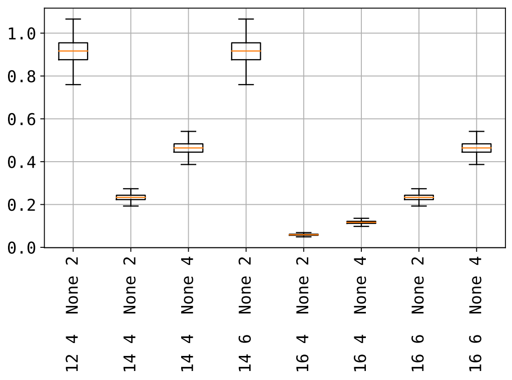
the histograms presented in Figure 6.11. Figure 6.12 presents the variations in the data width (presented as series), integer bits (column plots) and matrix sizes (row plots) and their effect on the PSNR values. The best PSNR reaches about 30dB, leading to 1000× the signal-to-noise ratio, suggesting a good quality in the results (more PSNR, better quality). The configurations with few integer bits (Figure 6.12(a, b, d, e)) present PSNR less than 15 dB, which are not as good as the ones presented with 4 bits in the integer part (Figure 6.12(c, f)), with PSNR greater than 30 dB for configurations with 2 × 2 PE matrix size and 16 bits of data width.

Until this point, the best configuration has more than 14 bits, more than 2 bits in the integer part and PEs with a matrix size of 2 × 2. Figure 6.13 proposes a configuration vs error diagram to summarise the findings from the histograms and the quality plots. Figure 6.13(a) shows the case of 120 × 10 matrix operations, and Figure 6.13(b) for 400 × 120. *The more robust accelerator configuration is 16-bit data width (4-bit integer*

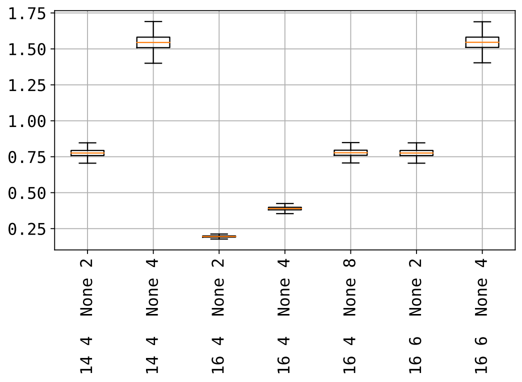
Table 6.4: Best solutions from Figure 6.14(a). In this context, area is assumed to be the same as resource utilisation. Matrix size: 120×10

Solution	PEs	PE Matrix	Data Type	Latency	Area	Error
1.1	1	2×2	$W = 12, I = 2$	7 clocks	4.1%	35%
1.2	1	2×2	$W = 14, I = 2$	7 clocks	4.1%	20.6%
1.3	1	2×2	$W = 14, I = 4$	7 clocks	4.1%	23.3%
1.4	1	2×2	$W = 16, I = 2$	7 clocks	4.1%	15.4%
1.5	1	2×2	$W = 16, I = 4$	7 clocks	4.1%	5.9%
1.6	1	2×2	$W = 16, I = 6$	7 clocks	4.1%	23.3%

part) with a 2×2 PE matrix size. From the PSNR analysis, this configuration achieves the highest PSNR value (> 30 dB).



(a)



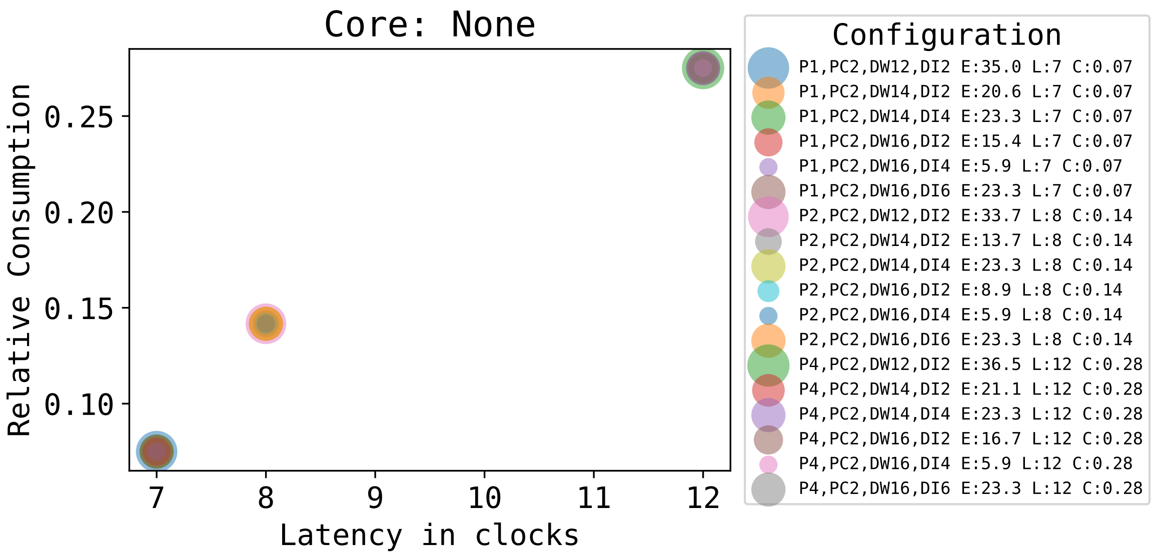
(b)

Figure 6.13: Error affectation of several configurations. The configurations in the X-axis follow the syntax: Datatype width, integer width, core, PE matrix size. In this case, the core is defined as `None` given that there are no other GEMMA configurations. The Y-axis refers to the normalised error. Few configurations are shown due to the error magnitude (less than 100% of error). (a) shows the errors for 120×10 matrices, and (b) shows the errors for 400×120 matrices.

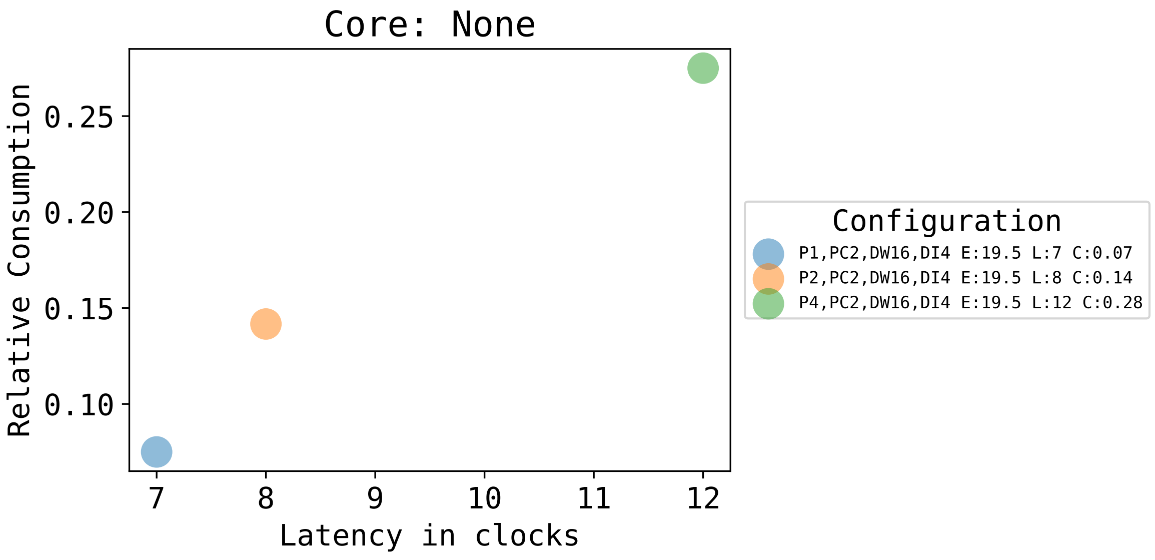
Performance analysis and best candidates

For the matrix sizes analysed by this work, the best candidate is the accelerator with 16-bit data width (4-bit integer part) and 2×2 PE matrix size, presenting the best quality metric, lowest mean error and most narrow distribution. However, recalling that some error tolerance can be accepted to get better candidates regarding resource consumption and performance, this work presents a distilled DSE plot for filtering the most promising solutions.

Figure 6.14 shows two bubble plots illustrating the DSE results for the solutions presented in this section. Taking into account the possibility of having two accelerators



(a)



(b)

Figure 6.14: Scatter plot of solutions. Closest to the origin, the best solution. The solutions are represented as bubbles, whose position is given by the relative consumption and latency. The size of the bubbles represents the error.

(one per matrix size), Figure 6.14(a) shows the case of 120 × 10 matrix evaluation, and Figure 6.14(b) shows the 400 × 10 case. The solutions closest to the origin present the lowest resource consumption and the lowest latency (more computational performance and design efficiency). Moreover, most miniature bubble diameter solutions are the best in terms of error. In the case of 120 × 10, the legend of the bubble plot suggests six configurations with the best trade-off between performance, area, and error. Table 6.4 shows the details of these configurations. One of the interesting facts is that having 6 bits in the integer part in a 16-bit data width configuration is worse than having 4 bits in the integer part for the same width. From the table, the user can safely choose the 16-bit data

Table 6.5: Best solutions from Figure 6.14(b). In this context, area is the same as resource utilisation. Matrix size: 400 × 120

Solution	PEs	PE Matrix	Data Type	Latency	Area	Error
2.1	1	2×2	$W = 16, I = 4$	7 clocks	4.1%	19.5%
2.2	1	4×4	$W = 16, I = 4$	12 clocks	29.5%	39%
2.3	2	2×2	$W = 16, I = 4$	8 clocks	7.7%	19.5%
2.4	4	2×2	$W = 16, I = 4$	12 clocks	15.0%	19.5%

Table 6.6: Performance of the distilled DSE solutions for the GEMMA accelerator

Solution	PEs	PE Matrix	Data Type	Performance	Efficiency
1.1	1	2×2	$W = 12, I = 2$	9.0 GOP/s	3.2%
1.2	1	2×2	$W = 14, I = 2$	9.0 GOP/s	3.2%
1.3	1	2×2	$W = 14, I = 4$	9.0 GOP/s	3.2%
1.4	1	2×2	$W = 16, I = 2$	9.0 GOP/s	3.2%
1.5	**1**	2×2	$W = 16, I = 4$	**9.0 GOP/s**	**3.2%**
1.6	1	2×2	$W = 16, I = 6$	9.0 GOP/s	3.2%
2.1	1	2×2	$W = 16, I = 4$	9.0 GOP/s	3.2%
2.2	1	4×4	$W = 16, I = 4$	6.7 GOP/s	2.4%
2.3	2	2×2	$W = 16, I = 4$	8.1 GOP/s	3.0%
2.4	4	2×2	$W = 16, I = 4$	5.9 GOP/s	2.1%

width 4-bit integer configuration without penalties at the latency and the resource level, ensuring a good mean error value. Moreover, the number of PEs can be scaled freely in a proportional fashion, increasing efficiency and parallelism.

Table 6.5 summarises the distilled DSE results for a matrix size of 400x120, presenting configurations with the most optimal data type configuration (width: 16 bits, integer part: 4 bits) with similar errors. In this case, it is safe to discard the second solution and keep the error under control, changing the focus from the error to the latency and resource perspective. A critical remark is that the number of PEs affects the latency of the accelerator. Thus, it is possible to choose a configuration with 4 PEs (solution 2.4) to take advantage of the parallelism or the configuration of 2 PEs (solution 2.3), keeping the latency low. Solution 2.3 implies having more clock cycles in latency by consuming 3.6% more resources. Solution 2.4 consumes 3.7× more resources and takes 1.7× more clock cycles. Therefore, the most reasonable is Solution 2.3 if having the resources available for investment.

Looking at the performance and design efficiency, Table 6.6 summarises the peak performance and the design efficiency of the solutions mentioned in Tables 6.4 and 6.5. From the table, there are configurations with the same design efficiency. Hence, the configuration with 16-bit width (4-bit integer part) and a single PE is the most efficient. It might suggest that the overhead added by the accelerator places room for optimisation and future work.

6.3.2 Window-based Convolution

Similarly to the GEMM, the analysis of the Window-based Convolution implies the exploration of the data width, the width of the integer part in the case of fixed-point numbers, the PE matrix size, and the kernel size. The difference is that the convolution no longer depends on the input matrix.

This work only considers accelerators based on exact fixed-point arithmetic and 3×3 kernel for simplicity. The metrics utilised for the analysis include:

- **Resource consumption**
  - Relative consumption of BRAM, FFs, LUT, and DSPs in a ZYNQ XC7Z020 with respect to the data width, integer width, and PE matrix size.
  - Maximum consumption of these components.
  - Resource consumption of the template without taking into account the execution module.

Solution Selection

Similar to the GEMM, the convolution accelerator presented several implementations (see subsection 4.4.4), including (1) serial, (2) hybrid, (3) pipeline, and (4) dataflow architectures. For this case, the following DoF values are fixed:

- `Q_INT` (integer part): 1-bit
- `Q_K` (kernel size): 3×3
- `Q_PES` (number of PEs): 4
- `CLOCK_PERIOD`: 7.5 ns target.

In terms of the data width, two values are used for comparison: 8 and 16 bits. The solution optimisations are the following:

- **Solution 0: Serial**
  - None (defaults).
- **Solution 1: Hybrid - Only execute module optimisation**
  - Pipeline the modules.

Table 6.7: Spatial solution optimisation results. The results are similar for the Winograd except for the additional latency as illustrated in the PE analysis (see 6.2.2). The consumptions are referred to the ZYNQ XC7Z020. There are multiple configurations for the hybrid architecture and each solution is incremental. "L" stands for Look-up Tables, "F" for Flip-Flops, "B" for BRAM, "D" for DSPs, and "DW" for Data Width

Solution	DW	Architecture	Consumptions	Latency
0	8	Serial	B: 0(0%) F: 4610(4%) L: 6143(11%) D: 0(0%)	764 cycles
0	16	Serial	B: 0(0%) F: 7587(7%) L: 6613(12%) D: 1(0%)	922 cycles
1	8	Hybrid	B: 0(0%) F: 10619(9%) L: 17267(32%) D: 1(0%)	34 cycles
1	16	Hybrid	B: 0(0%) F: 14344(13%) L: 16273(30%) D: 36(16%)	41 cycles
2	8	Pipeline	B: 0(0%) F: 6073(5%) L: 10343(19%) D: 0(0%)	16 cycles
2	16	Pipeline	B: 0(0%) F: 10510(9%) L: 4589(8%) D: 144(65%)	23 cycles
3	8	Dataflow	B: 0(0%) F: 6196(6%) L: 11554(22%) D: 0(0%)	18 cycles
3	16	Dataflow	B: 0(0%) F: 9689(9%) L: 5591(10%) D: 144(65%)	24 cycles
4	8	Dataflow	B: 0(0%) F: 7363(6%) L: 15297(28%) D: 0(0%)	38 cycles
4	16	Dataflow	B: 0(0%) F: 10891(10%) L: 14373(27%) D: 36(16%)	45 cycles

  - Pipeline data reads.
  - Inline modules.
  - Partition interfaces: all interfaces are partitioned completely into registers.
  - Partition cache buffers: all the matrix buffers (or caches) are implemented as registers.

- **Solution 2: Pipeline**
  - Include solution 1.
  - Remove inlining.
  - Vectorisation to the execution module.
  - Optimise execution unit by unrolling.
- **Solution 3: Dataflow**
  - Include solution 2.
  - Apply dataflow to the top module over the processing flow.
  - Make outer arrays static to avoid unwanted initialisation.
- **Solution 4: Dataflow unvectorised**
  - Include solution 3.
  - Removes vectorisation (as an experiment).

Table 6.7 shows the results for the optimisations intended for spatial convolution. The serial implementation has the worst latency but the best area usage, similar to the GEMM.

Solution 1 includes a series of optimisations to all the modules to implement a hybrid architecture. It considers that static modules have a greater impact than the execution alone, leading to 22× gains in latency, using twice the resources (overall). The pipeline architecture proposed in Solution 2 implements a pipeline architecture, leading to 2 times less latency than Solution 1 but spending twice the resources overall but freeing logic resources. Solution 3 proposes a data flow architecture with a similar consumption and more latency, which is inconvenient. Removing the vectorisation (Solution 4) demonstrates the effect of the vectorisation on resource consumption, freeing 75% DSPs and leading to almost twice the latency. Depending on the available resources, vectorisation can be removed in exchange for resources.

Therefore, the best implementation is the pipeline (Solution 2) regarding latency; however, the resource consumption is critically high for high data widths. Moreover, Solution 4 can be suitable for constrained resources in exchange for latency.

From now on, the best solutions highlighted in GEMMA (Solution 5) and convolution (Solution 2) are considered for the rest of the analysis.

Resource consumption

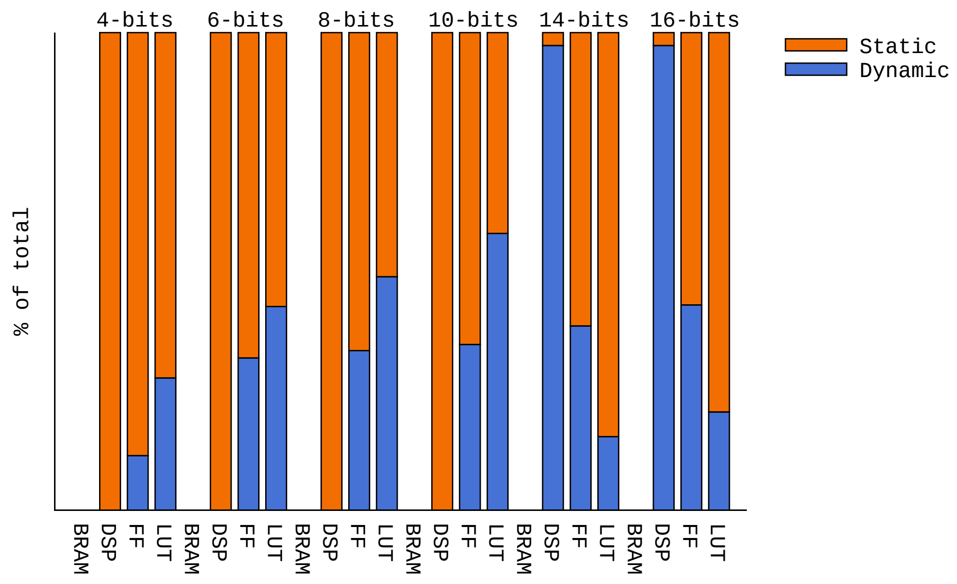


Figure 6.15: Dynamic part (PE) vs the total accelerator usage. After 14-bits, the utilisation becomes constant in terms of maximum utilisation. The configuration belongs to a single-core accelerator with a 2 × 2 PE.

According to 4.4, the convolution accelerator integrates a static and dynamic part, which behaves similarly to the GEMMA accelerator. Figure 6.15 summarises the proportion of accelerator usage in terms of the static and dynamic part.

For accelerators with a data width of fewer than 14 bits, the consumption keeps less than 57.9%. If the width exceeds 14 bits, the DSP becomes the highest utilisation, reaching more than 97.3% of the accelerator. Nevertheless, in terms of LUTs and FFs, the PE utilisation is less than 43% of FFs and 21% of LUTs. It suggests that the template takes most of the LUTs and FFs of the accelerator due to the FF-based registers used for implemented arrays partitioned completely and the implementation of the communication interface, similar to the GEMMA accelerator.

With the composition of the accelerator already analysed, it is time to perform the general DoF analysis on the accelerator. Figures 6.16 illustrate the behaviour of the accelerator when varying the data width and the matrix size of the PE for Spatial and Winograd convolution PEs. The behaviour when scaling the data width is the same as observed in the PE analysis in Figure 6.4. In the Spatial accelerator (Figure 6.16(a)), the LUT consumption is predominant when having 4-10 bits in data width with a linear scaling as the data width scales. The behaviour is followed by the FFs as well. It suggests that the scaling is because of the array partitioning to implement registers. After 12-bits, the DSP consumption becomes predominant, and the LUTs and FFs preserve the scaling behaviour. The pattern repeats in the Winograd accelerator (Figure 6.16(b)), but the predominance change happens when having 8-bit data width. Comparing the Spatial and Winograd accelerators, the consumption at the DSP level is lower in Winograd. *After 8-bits, Winograd consumes less area (or resources) than the Spatial.*

Varying the PE matrix is only analysed on the Spatial convolution provided that this work does not cover the optimised variants of Winograd for windows greater than 2×2. The affectation of this DoF is more severe than the other DoF variation. Figures 6.16(c, d) summarises the behaviour of using 4-bit and 8-bit data widths, respectively. The consumption scales proportionally with respect to the data width in this case. Regarding the PE matrix, a 2×2 PE-based accelerator consumes $5\times$ less than the 8×8 configuration. The PE analysis presented in Figure 6.6 shows that the best configuration has the smallest window size with 4-bit data width. The results, in this case, preserve this fact.

Hence, the resource consumption analysis determines that the behaviour is close to the observed in the PE resource consumption analysis, presenting the same behaviour despite the fact that the template adds resource overhead due to caching and protocol implementation. The conclusion is the same as in the matrix accelerator. Moreover, in terms of the best candidate, *Winograd offers less resource consumption than the Spatial convolution for every data width.*

Error quantification

The error evaluation considers the 3×3 convolution applied to images, given that image processing-based convolution is the same operation as the one used in DLI. Moreover, it proves the extensibility of the FAL project for other applications. Although LeNet-5 uses 5×5 kernels, this analysis will limit the kernel to 3×3, given the Winograd limitations. This evaluation varies the data width to see the error affectation, given that it is the only variable that affects the precision. The integer part is fixed to 1 bit since the convolution

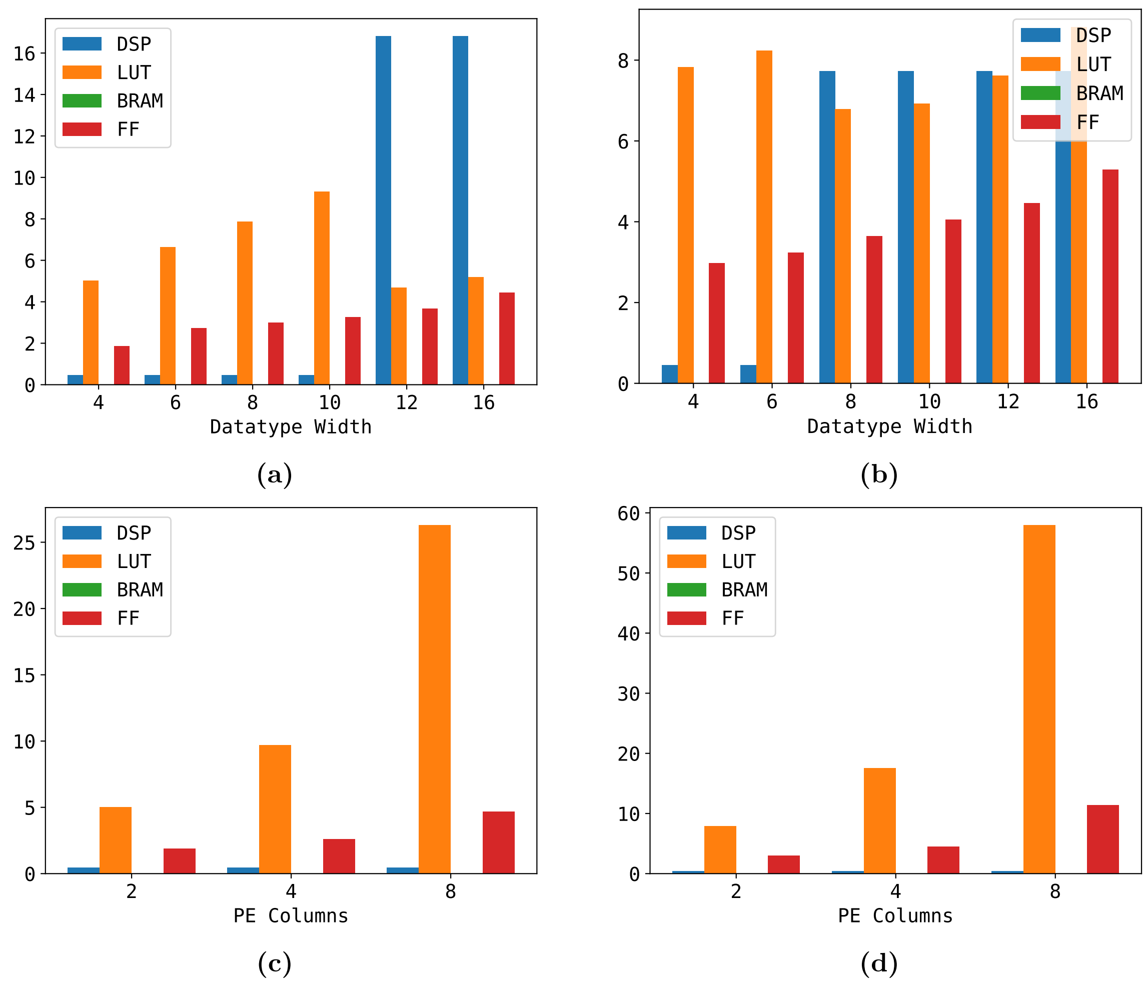


Figure 6.16: Consumption report of the convolution accelerator while varying data type width and the PE matrix size. The X-axis refers to the values of the DoF and the Y-axis to the FPGA resource consumption (%). The number of PEs is one core and the integer part has only one bit. (a) varies the data width while keeping the PE matrix size to 2×2 and the integer part to 1 by using the Spatial PE. (b) has the same conditions as (a) but uses the Winograd PE, (c) varies the PE matrix size (square matrix) while keeping the data width in 4 bits (1-bit integer) and using the Spatial PE, and (d) has the same conditions as (c) but uses 8 bits data width.

is numerically stable (fixed number of multiplications and additions) within the range of -0.5 to 0.5.

Figure 6.17 shows the error histograms for the Spatial and Winograd-based convolution accelerators. The results are similar to those presented in Figure 6.7, highlighting that Winograd's error resilience is stronger than the Spatial convolution's. The error introduced in the Spatial convolution is more severe than the one introduced by Winograd. For instance, for a 6-bit configuration, the error in Spatial is around 10-15%, whereas Winograd's error is more concentrated in $< 10\%$. *Winograd continues to be the best in terms of error.*

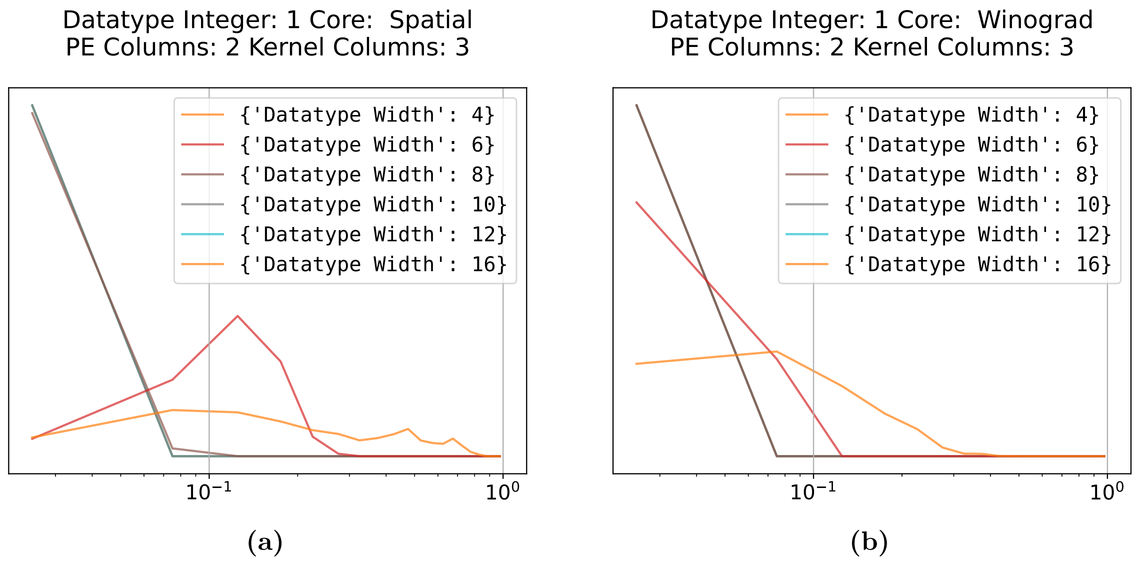


Figure 6.17: Error histogram of the convolution accelerator when varying data width from 4 up to 16 bits. Each histogram takes into account different implementations (Spatial on the left, Winograd on the right). The Y-axis is not normalised and allows showing the shape of the error distribution. The normalised error (X-axis) is measured by using (6.5).

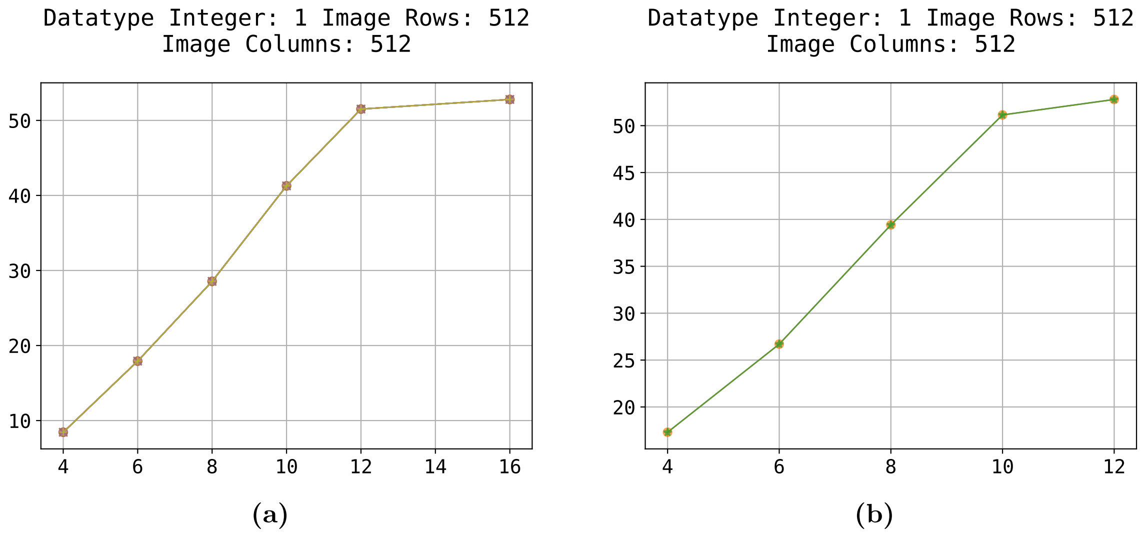
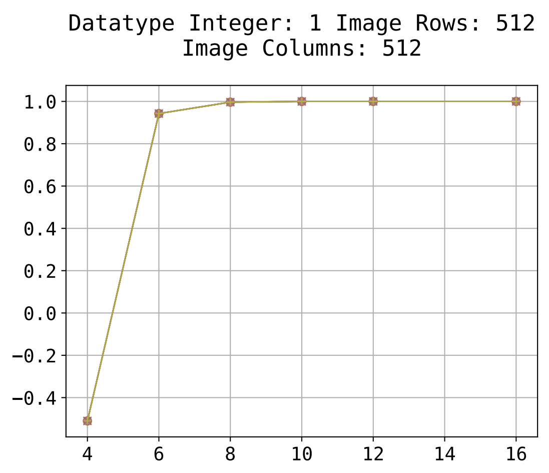
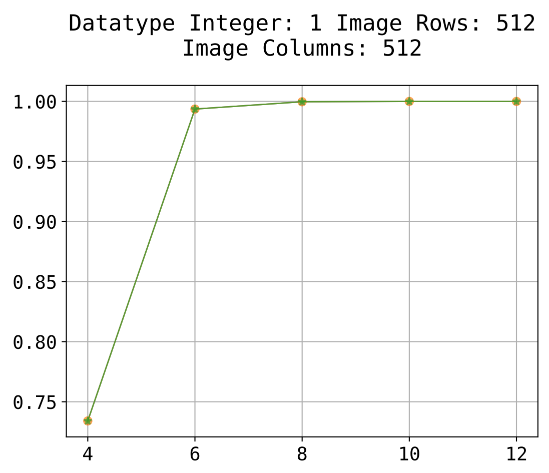


Figure 6.18: PSNR behaviour as the data width increases. On the left, the Spatial-based convolution results are presented (a). On the right, the Winograd-based ones (b). Series include configurations with 1, 2, and 4 PEs, and $2 \times 2, 4 \times 4$, and 8×8 (for Spatial) matrix sizes. These parameters do not make any difference in convolution.

Besides the error distribution, the convolution accelerator can be evaluated by quality metrics. Figures 6.18 and 6.19 show the quality evaluation by using PSNR and SSIM. In the case of SSIM, the 95% of similarity index is achieved with 6 bits or more, suggesting that the reconstruction of images is quite robust for low bit widths. Under the premise



(a)



(b)

Figure 6.19: SSIM behaviour as the data width increases. On the left, the Spatial-based convolution results are presented (a). On the right, the Winograd-based ones (b). Series include configurations with 1, 2, and 4 PEs, and $2 \times 2, 4 \times 4$, and 8×8 (for Spatial) matrix sizes. These parameters do not make any difference in convolution.

that 95% is acceptable, looking at the PSNR values, the Spatial-based convolution has a PSNR of approximately 18 dB, whereas the Winograd-based convolution exceeds the 25 dB (Figure 6.12). The Winograd-based convolution has a similar performance when using 4-bits compared to the Spatial-based convolution with 6-bit data width. Exceeding 30 dB requires, in both cases, using more than 8 bits. Compared to the GEMMA accelerator, the convolutions are more friendly in terms of approximation due to the number of operations to compute a single pixel (or value) and are more resilient to errors. Given this fact, convolutional-predominant networks might benefit from using approximate convolution accelerators.

This analysis also applies the Frobenius norm on the differences between matrices to measure the similarity between the resulting matrices. In the case of the Spatial convolution, the Frobenius norm is more than two times higher than the results in Winograd, suggesting that the dissimilarity increased by that factor for a given data width. Figure 6.20 provides two plots illustrating the behaviour of the Frobenius norm in the Spatial and Winograd convolutions.

To conclude the error analysis, Figure 6.21 shows the solution candidates and the error affectation of every solution. In this case, Winograd is the best candidate even for 4-bit data width configurations, achieving a mean error of less than 10%. Compared to the GEMMA accelerator, the convolution accelerators have more control over the errors since, in the plots, there are configurations with the minimum data width that can still be useful for the user, not reaching more than 100% of error.

Until this point, if the developer requires more error tolerance, *the best candidates are the Winograd convolution accelerators because of their approximation tolerance.*

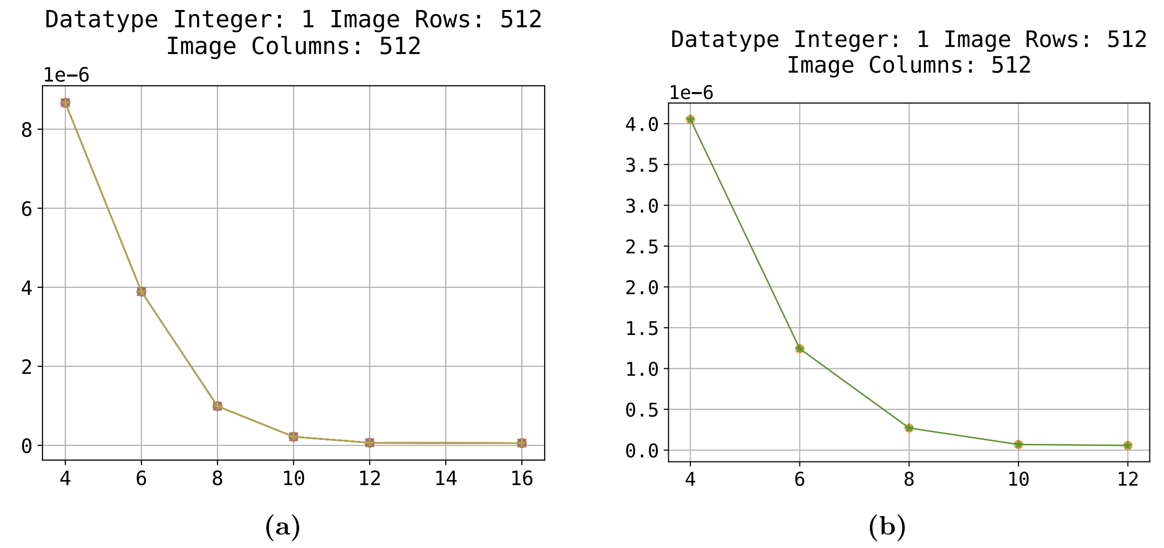


Figure 6.20: Frobenius norm behaviour as the data width increases. On the left, the Spatial-based convolution results are presented (a). On the right, the Winograd-based ones (b). Series include configurations with 1, 2, and 4 PEs, and $2 \times 2, 4 \times 4$, and 8×8 (for Spatial) matrix sizes. These parameters do not make any difference in convolution.

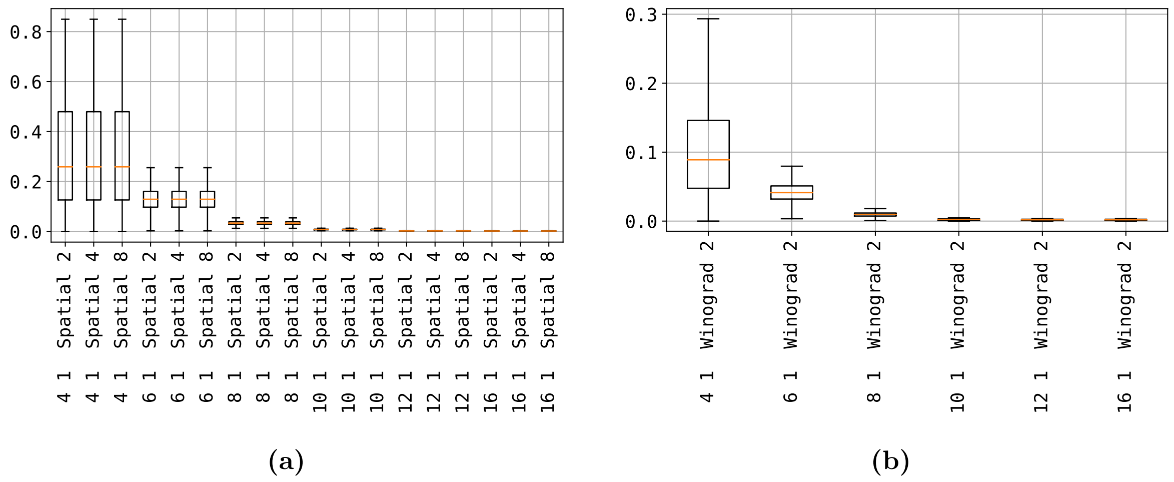
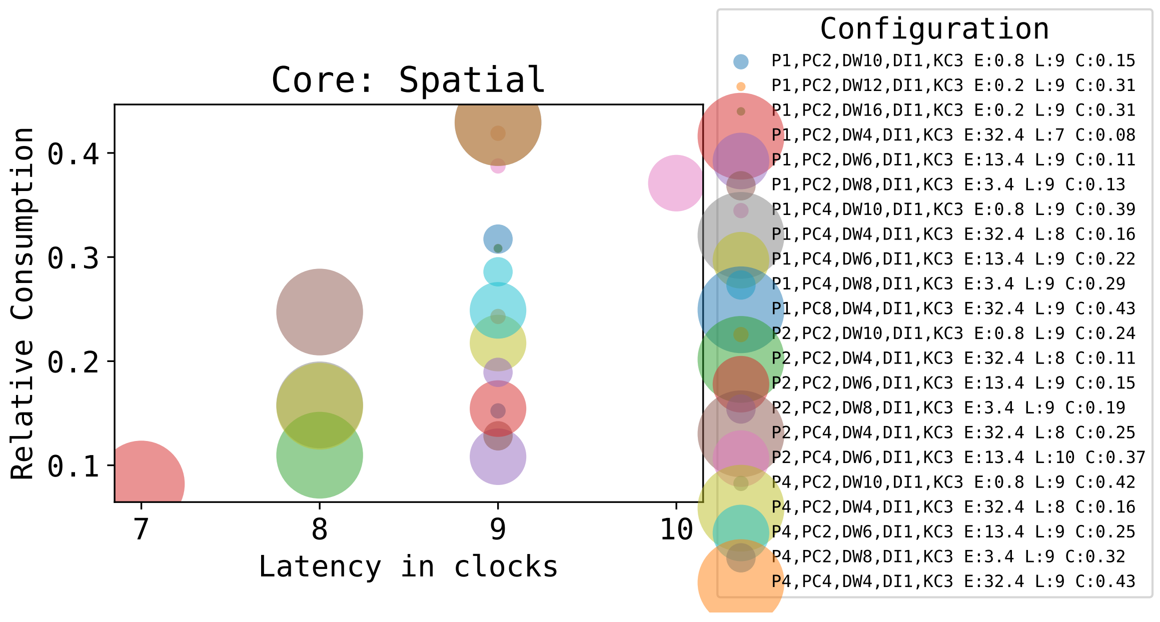


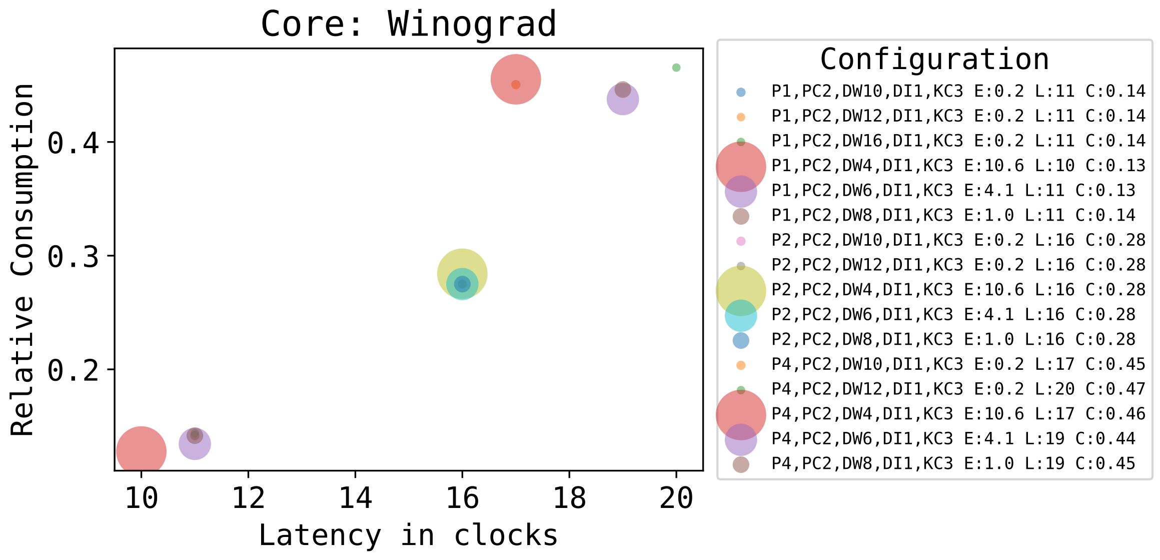
Figure 6.21: Error affectation of several configurations. The configurations in the X-axis follow the syntax: Datatype width, integer width, core, PE matrix size. The Y-axis refers to the normalised error. (a) shows the Spatial convolution candidates, and (b) the Winograd cantidates.

Performance analysis and best candidates

Until this moment, in terms of resource consumption and error, Winograd results are the best implementation, even for low data widths. Similar to the affirmation in the GEMMA accelerator, the designer can face different scenarios in which the resources are limited to give room to other IP cores and require high throughput. Usually, the error is the price-to-pay to achieve high performance in systems with restrictive resources.



(a)



(b)

Figure 6.22: Scatter plot of solutions. Closest to the origin, the best solution. The solutions are represented as bubbles, whose position is given by the relative consumption and latency. The size of the bubbles represents the error. The syntax of each series is: `P4,PC2,DW10,DI1,KC3`, which leads to an accelerator with 4 PEs (P4), 2×2 matrix size (PC2), 10-bit data width (DW10), 1-bit integer (DI1), and 3×3 kernel size (KC3). `E` specifies the error in %, `L` the latency in clocks, and `C` the resource consumption.

Figure 6.22 presents the bubble plots for the Winograd and Spatial convolution, illustrating the most suitable solutions discovered during the DSE. The plot illustrates more solutions than GEMMA, given that the error is much less than in that accelerator. The solutions closest to the origin are the Pareto optimal. Although Winograd is the best in

Table 6.8: Top 3 for different data width values. This ranking highlights the best architecture in terms of its balance between resource consumption and performance. The first group of solutions is based on the Spatial, whereas the second is on the Winograd PE.

Solution	PEs	PE Matrix	Data Type	Performance	Efficiency
1.1	2	8×8	$W = 4, I = 1$	90.3 GOP/s	32.7%
1.2	1	8×8	$W = 6, I = 1$	45.1 GOP/s	16.4%
1.3	2	4×4	$W = 8, I = 1$	36.1 GOP/s	13.1%
2.1	1	2×2	$W = 4, I = 1$	14.7 GOP/s	5.3%
2.2	1	2×2	$W = 6, I = 1$	13.3 GOP/s	4.8%
2.3	1	2×2	$W = 8, I = 1$	13.3 GOP/s	4.8%

resource consumption and error, Spatial is the best in latency.

Assuming that an application requires 30 dB, the accelerator requires at least 8 bits when using Winograd or Spatial. Fixing this case, a solution with 8-bit data width, 1 PE Core and 2×2 window implies having an error of 3.4%, 7.9% of resource consumption in the Spatial and a latency of 9 clocks in the Spatial convolution, according to Figure 6.22(a). For Winograd, a similar solution has a 1% error, but it takes 11 clocks to solve the operation, consuming 7.7% of the resources. For this example, the Spatial becomes the best option if the priority is performance. However, Winograd works well if the priority is either minimising the error or consuming fewer resources. Therefore, there is an inflexion point from the earlier analysis. Spatial became one of the best options depending on the design requirements.

A better overview of the performance analysis is presented in Table 6.8, showing a top 1 ranking of the solutions for several data widths from 4 to 8. To provide a fair comparison, the Winograd convolution was assigned the same number of operations as the Spatial convolution, determined by: $O = N_K^2 + N_O^2$. In principle, Winograd performs more operations, benefiting the number of operations and the peak performance, as Figure 6.6 illustrates. However, this assumption makes both solutions comparable by assuming that both perform the same work. Hence, in all the cases, the Spatial convolution was the most efficient according to the design efficiency figure of merit. Recalling the meaning of design efficiency, it is the balance between resource consumption invested in exchange for performance (or latency). Thus, *the best trade-off in terms of performance-resources is given by the Spatial convolution*, whereas *in terms of error, the best implementation is Winograd.* This highlights the relevance of performing DSE and the framework's capability for this type of exploration.

Table 6.9: Top 10 of the GEMMA Accelerator solutions in terms of performance and efficiency. *Zed* refers to Zedboard, *Pico* to the PicoEVB, and *Kria* to the Xilinx Kria. *Mat.* refers to the PE matrix size.

	Configuration			**Performance GOP/s**			**Efficiency**		
Sol.	**Datatype**	**Mat.**	**PEs**	**Zed**	**Pico**	**Kria**	**Zed**	**Pico**	**Kria**
1	W=12, I=2	2	1	8.96	5.02	20.42	**0.032**	**0.033**	**0.013**
2	W=12, I=4	2	1	8.96	5.02	20.42	**0.032**	**0.033**	**0.013**
3	W=12, I=6	2	1	8.96	5.02	20.42	**0.032**	**0.033**	**0.013**
4	W=14, I=2	2	1	8.96	5.02	20.06	**0.032**	**0.033**	0.013
5	W=14, I=4	2	1	8.96	5.02	20.06	**0.032**	**0.033**	0.013
6	W=14, I=6	2	1	8.96	5.02	20.06	**0.032**	**0.033**	0.013
7	W=16, I=2	2	1	8.96	5.02	19.70	**0.032**	**0.033**	0.013
8	W=16, I=4	2	1	8.96	5.02	19.70	**0.032**	**0.033**	0.013
9	W=16, I=6	2	1	8.96	5.02	19.70	**0.032**	**0.033**	0.013
10	W=12, I=1	2	1	8.96	5.02	19.34	**0.032**	**0.033**	0.012

6.4 Inter-FPGA comparison

So far, the analyses have used the Zedboard (ZYNQ XC7Z020) as the reference board for relative consumption, peak performance, and efficiency. This section compares the top 10 solutions of the matrix multiplication-addition (GEMMA), spatial convolution, and Winograd convolution accelerators using three platforms:

- Xilinx Kria KV26 (Kria): a mid-end FPGA-based SoC.
- Xilinx XC7Z020 (Zedboard): a low-end FPGA-based SoC.
- Xilinx XC7A50T (PicoEVB): a low-end / low-power FPGA.

These platforms represent three possible applications. The XC7A50T is the smallest FPGA on the list, and it can implement standalone applications or work as an accelerator. The SoC are self-contained systems that can act as edge computing devices. In the case of the Xilinx Kria KV26, it targets computer vision applications.

Tables 6.9, 6.10 and 6.11 show the top 10 accelerators analysed in previous sections. Table 6.9 summarises configurations with more than 12-bit widths. The top 10 is mainly composed of 2 × 2 PE matrices and single-PE accelerators. In terms of efficiency, changing the data width does not affect efficiency or performance. Depending on the FPGA, the solutions have different efficiency values due to different relative consumptions and peak performances. The PicoEVB is the smallest FPGA within the sample, and the Kria is the largest. According to the results, the GEMMA accelerator has more efficiency in the PicoEVB and less performance on the Kria, favouring small FPGAs. Moreover, the top 3 are the most efficient solutions for the three boards simultaneously.

Table 6.10: Top 10 of the Spatial Convolution Accelerator solutions in terms of performance and efficiency. *Zed* refers to Zedboard, *Pico* to the PicoEVB, and *Kria* to the Xilinx Kria

	Configuration			**Performance GOP/s**			**Efficiency**		
Sol.	**Datatype**	N_O	**PEs**	**Zed**	**Pico**	**Kria**	**Zed**	**PicoEVB**	**Kria**
1	6	4	4	45.14	30.09	90.28	**0.16**	**0.20**	**0.06**
2	6	8	1	45.14	30.09	90.28	**0.16**	**0.20**	**0.06**
3	6	4	2	45.14	27.08	90.28	**0.16**	0.18	**0.06**
4	6	8	2	45.14	22.57	67.71	**0.16**	0.15	0.04
5	6	4	1	40.12	25.07	85.26	0.15	0.17	0.05
6	8	4	2	36.11	18.05	72.22	0.13	0.12	0.05
7	6	2	4	35.10	25.07	75.23	0.13	0.17	0.05
8	6	8	4	30.09	30.09	60.18	0.11	0.20	0.04
9	8	4	1	30.09	20.06	65.20	0.11	0.13	0.04
10	8	2	4	30.09	20.06	60.18	0.11	0.13	0.04

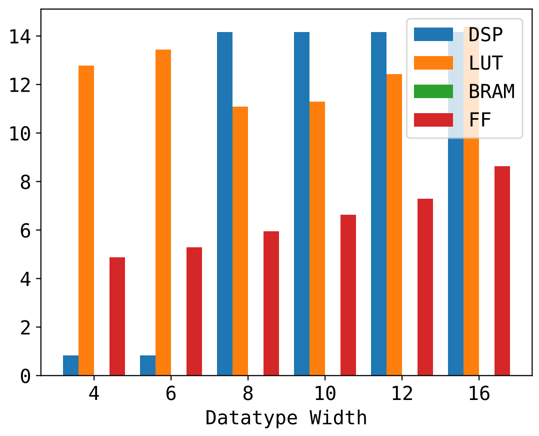
Table 6.11: Top 5 of the Winograd Convolution Accelerator solutions in terms of performance and efficiency. *Zed* refers to Zedboard, *Pico* to the PicoEVB, and *Kria* to the Xilinx Kria

	Configuration		**Performance GOP/s**			**Efficiency**		
Sol.	**Datatype**	**PEs**	**Zed**	**Pico**	**Kria**	**Zed**	**Pico**	**Kria**
1	4	1	14.67	9.02	32.72	**0.05**	**0.06**	0.02
2	8	1	13.33	8.20	33.85	0.05	0.05	**0.02**
3	10	1	13.33	8.20	32.82	0.05	0.05	0.02
4	12	1	13.33	8.20	29.75	0.05	0.05	0.02
5	6	1	13.33	8.20	27.70	0.05	0.05	0.02

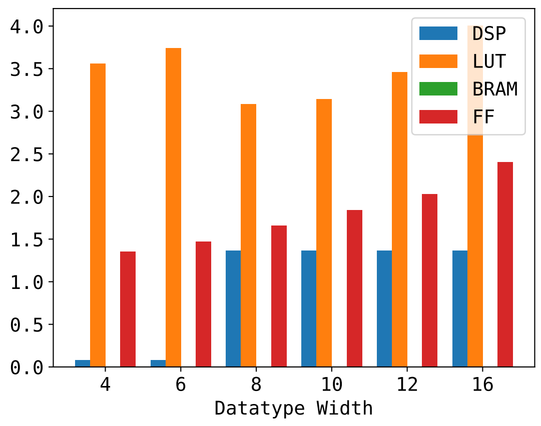
According to Tables 6.10 and 6.11, the convolution accelerators present the same phenomenon where the efficiency is the best for small FPGAs. The spatial convolution shows 20% efficiency in the PicoEVB, 16% in the Zedboard and 6% in the Kria. In Winograd, the efficiency degrades by three times. However, the divergence with respect to the GEMMA is found in the most efficient solutions. In the spatial convolution, the two first solutions are the most efficient for all the boards, whereas, in the Winograd convolution, there is no common solution.

Recalling the design performance (6.3) and the design efficiency (6.4), keeping the RTL implementation, the number of operations, and the latency, the main variations are the resource proportion r_p and the peak performance P_{peak}. Both quantities are FPGA-dependent. According to the definition of P_{peak}, it is the DSP performance, whereas r_p depends on the maximum resource consumption.

Focusing on the resource proportions (Table 6.11), Winograd's accelerator has an effi-



(a)



(b)

Figure 6.23: Winograd FPGA resource consumption in (a) the PicoEVB and (b) the Kria. Both represent extreme cases of low-end and mid-end FPGAs.

ciency of 6% in the PicoEVB and 2% in the Kria. The reason is because of the total available resources available in each FPGA. The PicoEVB integrates fewer DSPs than the Kria, altering the relative consumptions. Figure 6.23 shows the consumption as a function of the data width, and Winograd has been chosen for convenience since it cannot vary the matrix size and has the best error behaviour. It illustrates how the resource switch from LUTs to DSPs is less severe in the Kria compared to the PicoEVB, given that the proportion of DSPs is different concerning the other resources. In the Kria, the constant resource consumption does not happen because of the DSP consumption as in the Zedboard (Figure 6.16(a)), suggesting a different proportion of resources. Another interesting observation is that the PicoEVB FPGA keeps constant until 14 bits of data width. After 16 bits, the LUTs once more determine the resource consumption.

The number of DSPs also determines the peak performance. A platform with a low proportion of DSPs will show a better performance, suggesting an unbalance in efficiency. In frameworks such as FINN and HLS4ML, the absence of DSPs is an implementation killer, given that the designs tend to depend on the number of DSPs available. However, it might depend on the number of bits of the solution to implement. Implementing a whole neural network on a low-end FPGA can be challenging for these frameworks. Because of its granularity, FAL can implement accelerators to run the operations required for neural network computations on this FPGA.

In summary, the FPGA resource proportions can modify the conditions for the optimality of a solution. Thus, the DSE shall take into account the platform in order to determine the best-fit design solution.

6.5 Key Findings

This chapter has compiled all the results concerning evaluating the proposed PE and accelerator architectures for GEMMA and convolution. The analysis used the results provided by a greedy DSE that varied the data width, integer width, output/window size, inputs, number of units, and PE implementation (in the case of the accelerators). Moreover, this work has found that the FPGA and its available resources influence the DSE results, varying the best solutions in terms of performance.

The matrix multiply-add PE has been demonstrated to be sensitive to the matrix size, scaling linearly in terms of resources but even more sensitive in terms of errors because of the overflow control mechanisms. The effect of (3.2) is critical for scaling in matrix size, provoking operand vanishing as the matrix size increases, and losing information of small operands. After implementing an accelerator based on this PE, the resource consumption has a similar scaling behaviour. However, the error metrics worsened as the matrix size increased. To mitigate the adverse effect imposed on large matrices, the data width has been increased along with the integer width to keep the overflow under control. In this case, the overflow is caused by the inherent nature of matrix-multiplication additions, where adding causes numerical instability and goes out of bounds in the numerical representation. In the end, the best error results were 15% error and 30dB of PSNR by using a 16-bit data width with a 4-bit integer part, operating on 400×120 FCL layer.

On the other hand, the convolution PEs are more stable in terms of error, showing promising results when having low data widths. Winograd achieved 16.28 dB PNSR and 0.731 SSIM, whereas the Spatial required at least 6 bits to achieve similar results. Implementing an accelerator did not affect the error due to the dependency of the pixels, which required only the values of their neighbour pixels. In terms of scaling, the resource consumption is even more promising due to the tolerance of low data widths, requiring much less area than the GEMMA PE and accelerator. From the performance perspective, Winograd outstands in terms of solution error, whereas Spatial reaches the best performances.

Chapter 7

Conclusions

This thesis proposes a library with three different PE architectures for generic matrix multiplication-addition and convolution using generic programming and a High-Level Synthesis tool for the RTL generation. The addressed PE designs include a generic matrix multiply-add, a window-based spatial convolution, and a Winograd convolution. Each PE is customisable in the datatype, the data width, the arithmetic operators, and operand sizes (i.e. matrix sizes), allowing multiple solutions for an extensive design exploration analysis. The proposed framework explored the design space generated for each PE when changing the data width in bits for a fixed-point representation, the output sizes, and the kernel sizes. The design exploration analyses the numerical error for each PE as a density distribution and the resource consumption as the product of the permutations within the customisable parameters. The design efficiency has been analysed using the proposed figure-of-merit from a previous work, which measures how well the design consumes the resources in exchange for the computational operations.

After the solution analysis obtained by the DSE, it was possible to find Pareto's optimal solutions for accelerators based on these PE architectures, performing a brief analysis of the most suitable design architectures. For the matrix multiplication addition, the most suitable configuration required 16-bit data width with a 4-bit integer part to keep the error below 20%. When discarding solutions with more than 20% of error, the most promising solution was a single-PE accelerator with 2×2 operands, achieving up to 9 GOP/s with 3.2% efficiency in a ZYNQ XC7Z020. These results target an operation of a 400×120 FCL.

On the other hand, this research proposed two architectures to perform convolutions: a window-based spatial convolution and the Winograd convolution. The DSE analysis determined that these architectures were more error-resilient than the matrix multiplication-addition, admitting architectures with 4-bit data widths. Winograd outstands in terms of error, achieving 16.28 dB PNSR, 0.731 SSIM, and 10% of mean error when using 4-bit data width. The Spatial convolution required a 6-bit data width to achieve similar results. Nevertheless, the Spatial convolution achieved the best performance, presenting less latency at the cost of using more resources.

This work also highlights the relevance of performing a DSE analysis also considering the target platform due to the resource proportions. The Pareto's optimal can vary depending on the availability of the DSP units and logic cells.

Apart from the implementations, this work contributes by open-sourcing the results of this work for different FPGAs. The implementations can be adopted in other projects different to neural networks because of the versatility of the generic programming performed in C++ and design parameterisation. Moreover, the users can determine their best solutions regarding error, resource consumption and performance by running the DSE framwork suite included in the project, avoiding running a whole project and simulating under their use case conditions. The DSE suite also integrates a novel figure-of-merit called design efficiency that helps to rank the architectures in terms of performance vs resource consumption.

7.1 Future Work

The next step is analysing the resource consumption and the error distribution according to the customisable parameters. The idea is to have a model to estimate the effect of the parameters on the final results without running the synthesis, speeding up the design exploration given a deep learning model. Also, extend the research on an actual deep learning model for a complete design exploration and analysis of the implementations, including finalising the activation functions research and other activation functions.

References

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, “State-of-the-art in artificial neural network applications: A survey”, *Heliyon*, vol. 4, no. 11, e00938, 2018, ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2018.e00938. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405844018332067.

[2] C. J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang, “Machine learning at facebook: Understanding inference at the edge”, *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*, pp. 331–344, 2019. DOI: 10.1109/HPCA.2019.00048.

[3] T. Wu, W. Liu, and Y. Jin, “An End-to-End Solution to Autonomous Driving Based on Xilinx FPGA”, *Proceedings - 2019 International Conference on Field-Programmable Technology, ICFPT 2019*, vol. 2019-December, pp. 427–430, 2019. DOI: 10.1109/ICFPT47387.2019.00084.

[4] N. Lin, H. Lu, X. Hu, J. Gao, M. Zhang, and X. Li, “When deep learning meets the edge: Auto-masking deep neural networks for efficient machine learning on edge devices”, *Proceedings - 2019 IEEE International Conference on Computer Design, ICCD 2019*, no. Iccd, pp. 506–514, 2019. DOI: 10.1109/ICCD46524.2019.00076.

[5] NVIDIA, *Data Sheet NVIDIA Jetson Nano System-on-Module*, 2014.

[6] V. Gpu and C. Cpu, “DATA SHEET NVIDIA Jetson AGX Xavier Series System-on-Module”, pp. 1–72, 2019. [Online]. Available: https://developer.nvidia.com/embedded/downloads#?search=data&tx=$product,jetson_agx_xavier.

[7] N. Jouppi, C. Young, N. Patil, and D. Patterson, “Motivation for and evaluation of the first tensor processing unit”, *IEEE Micro*, vol. 38, no. 3, pp. 10–19, 2018. DOI: 10.1109/MM.2018.032271057.

[8] Google, *dvanced neural network processing for low-power devices*, https://coral.ai/technology, 2020.

[9] Intel, *Intel® Xeon® Platinum 8260 Processor*. [Online]. Available: %7Bhttps://ark.intel.com/content/www/us/en/ark/products/192474/intel-xeon-platinum-8260-processor-35-75m-cache-2-40-ghz.html%7D.

[10] NVIDIA, "NVIDIA A100 TENSOR CORE GPU", [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf.

[11] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "TABLA: A unified template-based framework for accelerating statistical machine learning", *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2016-April, pp. 14–26, 2016, ISSN: 15300897. DOI: 10.1109/HPCA.2016.7446050.

[12] Xilinx, *Zynq-7000 SoC*. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[13] Xilinx Inc., *Kria K26 SOM: The Ideal Platform for Vision AI at the Edge*, 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/white_papers/wp529-som-benchmarks.pdf.

[14] E. Mohsen, "Performance and Bandwidth in a Cost-Optimized", vol. 423, pp. 1–14, 2018.

[15] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, *Pruning and quantization for deep neural network acceleration: A survey*, 2021. arXiv: 2101.09671 [cs.CV].

[16] Xilinx Inc., *DPUCAHX8L for Convolutional Neural Networks*, 2021. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/dpucahx8l/v1_0/pg366-dpucahx8l.pdf.

[17] D. Rongshi and T. Yongming, "Accelerator Implementation of Lenet-5 Convolution Neural Network Based on FPGA with HLS", *2019 3rd International Conference on Circuits, System and Simulation, ICCSS 2019*, pp. 64–67, 2019. DOI: 10.1109/CIRSYSSIM.2019.8935599.

[18] M. A. Arshad, S. Shahriar, and A. Sagahyroon, "On the Use of FPGAs to Implement CNNs: A Brief Review", *Proceedings - 2020 International Conference on Computing, Electronics and Communications Engineering, iCCECE 2020*, pp. 230–236, 2020. DOI: 10.1109/iCCECE49321.2020.9231243.

[19] A. Lavin and S. Gray, *Fast algorithms for convolutional neural networks*, 2015. arXiv: 1509.09308 [cs.NE].

[20] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional Neural Networks using Logarithmic Data Representation", 2016. arXiv: 1603.01025. [Online]. Available: http://arxiv.org/abs/1603.01025.

[21] J. Li and A. Louri, "AdaPrune : An Accelerator-aware Pruning Technique for Sustainable CNN Accelerators", vol. XX, no. XX, 2021. DOI: 10.1109/TSUSC.2021.3060690.

[22] X. Chang, H. Pan, W. Lin, and H. Gao, "A Mixed-Pruning Based Framework for Embedded Convolutional Neural Network Acceleration", *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–10, 2021, ISSN: 15580806. DOI: 10.1109/TCSI.2020.3048260.

[23] K. Bhardwaj, N. Suda, and R. Marculescu, "EdgeAI: A Vision for Deep Learning in IoT Era", *IEEE Design and Test*, vol. 2356, no. c, pp. 1–6, 2019, ISSN: 21682364. DOI: 10.1109/MDAT.2019.2952350.

[24] F. Fahim, B. Hawks, C. Herwig, J. Hirschauer, S. Jindariani, N. Tran, L. P. Carloni, G. D. Guglielmo, P. Harris, J. Krupa, D. Rankin, M. Blanco, V. J. Hester, Y. Luo, J. Mamish, S. Orgrenci-Memik, T. Aarestaad, H. Javed, V. Loncar, M. Pierini, A. A. Pol, S. Summers, J. Duarte, S. Hauck, S.-C. Hsu, J. Ngadiuba, M. Liu, D. Hoang, E. Kreinar, H. Herndon, Z. Wu, M. Blanco Valentin, and J. Hester, "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices", in *TinyML Research Symposium*, 2021. arXiv: 2103.05579v1.

[25] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, pp. 1–23, 2018.

[26] B. Janßen, P. Zimprich, and M. Hübner, "A dynamic partial reconfigurable overlay concept for pynq", in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056786.

[27] W. S. Mcculloch and W. Pitts, "A Logical Calculus Of The Ideas Immanent In Nervous Activity", *Bulletin of Mathematical Biology*, vol. 52, no. l, pp. 99–115, 1990.

[28] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain.* US, 1958. DOI: 10.1037/h0042519.

[29] M. Minsky and S. Papert, "Perceptrons - an introduction to computational geometry", 1969.

[30] E. Alpaydin, "Neural Networks and Deep Learning", *Machine Learning*, 2021. DOI: 10.7551/mitpress/13811.003.0007.

[31] A. Khaled, A. F. Atiya, and A. H. Abdel-Gawad, "Applying Fast Matrix Multiplication to Neural Networks", *Proceedings of the ACM Symposium on Applied Computing*, pp. 1034–1037, 2020. DOI: 10.1145/3341105.3373852.

[32] M. de Rooij, *Ultra low latency deep neural network inference for gravitational waves interferometer*, 2021.

[33] Y. Kochura, Y. Gordienko, V. Taran, N. Gordienko, A. Rokovyi, O. Alienin, and S. Stirenko, "Batch size influence on performance of graphic and tensor processing units during training and inference phases", in *Advances in Computer Science for Engineering and Education II*, Z. Hu, S. Petoukhov, I. Dychka, and M. He, Eds., Cham: Springer International Publishing, 2020, pp. 658–668, ISBN: 978-3-030-16621-2.

[34] D. HUBEL and T. WIESEL, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex", pp. 106–154, 1962. DOI: doi:10.1113/jphysiol.1962.sp006837.

[35] K. Fukushima and S. Miyake, "Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position", *Pattern Recognition*, vol. 15, no. 6, pp. 455–469, 1982, ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(82)90024-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0031320382900243.

[36] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[37] H. J. Nussbaumer, "The fast fourier transform", in *Fast Fourier Transform and Convolution Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 80–111, ISBN: 978-3-662-00551-4. DOI: 10.1007/978-3-662-00551-4_4. [Online]. Available: https://doi.org/10.1007/978-3-662-00551-4_4.

[38] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[39] M. Hardieck, M. Kumm, K. Moller, and P. Zipf, "Reconfigurable convolutional kernels for neural networks on fpgas", in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19, Seaside, CA, USA: Association for Computing Machinery, 2019, pp. 43–52. DOI: 10.1145/3289602.3293905. [Online]. Available: https://doi.org/10.1145/3289602.3293905.

[40] J. Wang, J. Lin, and Z. Wang, "Efficient convolution architectures for convolutional neural network", in *2016 8th International Conference on Wireless Communications Signal Processing (WCSP)*, 2016, pp. 1–5. DOI: 10.1109/WCSP.2016.7752726.

[41] G. Bradski, "The OpenCV Library", *Dr. Dobb's Journal of Software Tools*, 2000.

[42] S. Skansi, *Introduction to deep learning: From Logical Calculus to Artificial Intelligence*, 6. 2018, vol. 114, p. 196, ISBN: 978-3-319-73003-5. DOI: 10.1007/978-3-319-73004-2.

[43] T. T. Sivri, N. P. Akman, and A. Berkol, "Multiclass classification using arctangent activation function and its variations", in *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2022, pp. 1–6. DOI: 10.1109/ECAI54874.2022.9847486.

[44] NVIDIA, "Layers and Precision", [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet.

[45] P. K. Shibo Wang, "BFloat16: The secret to high performance on Cloud TPUs", 2019. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[46] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] A survey of FPGA-based neural network inference accelerators", *ACM Transactions on Reconfigurable Technology and Systems*, vol. 12, no. 1, pp. 1–26, 2019, ISSN: 19367414. DOI: 10.1145/3289185.

[47] I. Corporation, "Intel ® Advanced Vector Extensions Programming Reference", *Intel Corporation*, no. July, p. 750, 2011.

[48] O. Guide, "Deep Learning with Intel ® AVX-512 and Intel ® Deep Learning Boost Tuning Guide on 3rd Generation Intel ® Xeon ® Scalable Processors", pp. 1–26,

[49] Intel, "Intel Architecture Instruction Set Extensions Programming Reference", *Technology*, no. February, 2021.

[50] ARM, *NEON Programmer's Guide*. 2013, ISBN: 9781439806104.

[51] ARM Developer, *Introduction to SVE*, 2021. [Online]. Available: https://bit.ly/3uYqlUX.

[52] L. G. Leon-Vega, *NanoSciTracker : an object tracking library for microbiology and an industrial collimation algorithm optimisation*, 2020.

[53] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The arm scalable vector extension", *IEEE Micro*, vol. 37, no. 2, pp. 26–39, 2017. DOI: 10.1109/MM.2017.35.

[54] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus", in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 79–92. DOI: 10.1109/ISPASS.2019.00016.

[55] NVIDIA Corporation, "NVIDIA TESLA V100 GPU ARCHITECTURE ", 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[56] NVIDIA, "NVIDIA AMPERE GA102 GPU ARCHITECTURE", [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf.

[57] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus", in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 79–92. DOI: 10.1109/ISPASS.2019.00016.

[58] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit", *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017, ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. [Online]. Available: https://doi.org/10.1145/3140659.3080246.

[59] J. Sengupta, R. Kubendran, E. Neftci, and A. Andreou, "High-speed, real-time, spike-based object tracking and path prediction on google edge tpu", in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020, pp. 134–135. DOI: 10.1109/AICAS48895.2020.9073867.

[60] NVIDIA Corporation, *NVDLA*, http://nvdla.org/.

[61] ARM Developer, *Ethos-N70*, https://developer.arm.com/ip-products/processors/machine-learning/arm-ethos-n/ethos-n78.

[62] M. Shafique, R. Hafiz, M. U. Javed, S. Abbas, L. Sekanina, Z. Vasicek, and V. Mrazek, "Adaptive and Energy-Efficient Architectures for Machine Learning: Challenges, Opportunities, and Research Roadmap", *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 627–632, 2017, ISSN: 21593477. DOI: 10.1109/ISVLSI.2017.124. [Online]. Available: http://ieeexplore.ieee.org/document/7987592/.

[63] Q. Xu, N. S. Kim, and T. Mytkowicz, "Approximate Computing: A Survey", *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, 2016, ISSN: 2168-2356. DOI: 10.1109/MDAT.2015.2505723. [Online]. Available: http://ieeexplore.ieee.org/ielx7/6221038/7386744/07348659.pdf?tp=%7B%5C&%7Darnumber=7348659%7B%5C&%7Disnumber=7386744%7B%5C%%7D5Cnhttp://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7348659%7B%5C&%7Dfilter%7B%5C%%7D3DAND%7B%5C%%7D28p%7B%5C_%7DIS%7B%5C_%7DNumber%7B%5C%%7D3A7386744%7B%5C%%7D29.

[64] S. Hashemi, H. Tann, F. Buttafuoco, and S. Reda, "Approximate Computing for Biometric Security Systems : A Case Study on Iris Scanning", pp. 319–324, 2018.

[65] M. Wyse, "Modeling Approximate Computing Techniques", [Online]. Available: https://homes.cs.washington.edu/%7B~%7Dwysem/publications/wysem-msreport.pdf.

[66] S. Sidiroglou-douskos, S. Misailovic, H. Hoffmann, and S. Sidiroglou, “perforation Citation Accessed Citable Link Detailed Terms Managing Performance vs . Accuracy Trade-offs With Loop Perforation”, 2013.

[67] L. Lai, N. Suda, and V. Chandra, “Deep Convolutional Neural Network Inference with Floating-point Weights and Fixed-point Activations”, no. Icml, 2017. arXiv: 1703.03073. [Online]. Available: http://arxiv.org/abs/1703.03073.

[68] C. Chen, “High-order Taylor series approximation for efficient computation of elementary functions”, *IET Computers & Digital Techniques*, vol. 9, no. 6, pp. 328–335, 2015, ISSN: 1751-8601. DOI: 10.1049/iet-cdt.2014.0158. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/iet-cdt.2014.0158.

[69] N. S. K. Qiang Xu Todd Mytkowicz, “Approximate computing: A survey”, *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[70] Intel, “Intel 64 and IA-32 Architectures Optimization Reference Manual”, *Intel Technology Journal*, vol. 09, no. 03, pp. 1–660, 2005, ISSN: 15222594. DOI: 10.1535/itj.0903.05.

[71] M. Abramowitz, *Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables,* USA: Dover Publications, Inc., 1974, ISBN: 0486612724.

[72] R. H. Bartels, J. C. Beatty, and B. A. Barsky, *An Introduction to Splines for Use in Computer Graphics Geometric Modeling.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, ISBN: 0934613273.

[73] M. Barbareschi, F. Iannucci, and A. Mazzeo, “Automatic design space exploration of approximate algorithms for big data applications”, in *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, 2016, pp. 40–45. DOI: 10.1109/WAINA.2016.172.

[74] ——, “An extendible design exploration tool for supporting approximate computing techniques”, in *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2016, pp. 1–6. DOI: 10.1109/DTIS.2016.7483888.

[75] S. Kim and Y. Kim, “Energy-efficient hybrid adder design by using inexact lower bits adder”, in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2016, pp. 355–357. DOI: 10.1109/APCCAS.2016.7803974.

[76] J. Lee, H. Seo, Y. Kim, and Y. Kim, “Design of a low-cost approximate adder with a zero truncation”, in *2020 International SoC Design Conference (ISOCC)*, 2020, pp. 69–70. DOI: 10.1109/ISOCC50952.2020.9332971.

[77] S. L. Harris and D. Harris, “4 - hardware description languages”, in *Digital Design and Computer Architecture*, S. L. Harris and D. Harris, Eds., Morgan Kaufmann, 2022, pp. 170–235, ISBN: 978-0-12-820064-3. DOI: https://doi.org/10.1016/B978-0-12-820064-3.00004-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128200643000040.

[78] Xilinx, “Vitis High-Level Synthesis User Guide”, *Ug1399*, vol. 2, pp. 1–657, 2020.

[79] T. H. Melissa Sussmann, “Intel® HLS Compiler: Fast Design, Coding, and Hardware”, 2016.

[80] Y. Sun, G. Wang, B. Yin, J. R. Cavallaro, and T. Ly, “Chapter 8 - high-level design tools for complex dsp applications”, in *DSP for Embedded and Real-Time Systems*, R. Oshana, Ed., Oxford: Newnes, 2012, pp. 133–155, ISBN: 978-0-12-386535-9. DOI: https://doi.org/10.1016/B978-0-12-386535-9.00008-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123865359000081.

[81] M. Schoeberl, “Digital Design with Chisel”, Tech. Rep., 2019, pp. 1–80.

[82] Xilinx Inc., “Vivado Design Suite User Guide: High-Level Synthesis”, *Ug902*, 2018. [Online]. Available: https://docs.xilinx.com/v/u/2018.2-English/ug902-vivado-high-level-synthesis.

[83] John L. Hennessy; David A. Patterson, *Computer Architecture - A Quantitative Approach 5th edition*, 9. 2012, vol. 53, ISBN: 9780123838728.

[84] S. Froehlich, L. Klemmer, D. Grose, and R. Drechsler, “ASNet: Introducing Approximate Hardware to High-Level Synthesis of Neural Networks”, *Proceedings of The International Symposium on Multiple-Valued Logic*, vol. 2020-Novem, pp. 64–69, 2020, ISSN: 0195623X. DOI: 10.1109/ISMVL49045.2020.00-28.

[85] D. H. Noronha, K. Gibson, B. Salehpour, and S. J. Wilton, “LeFlow: Automatic Compilation of TensorFlow Machine Learning Applications to FPGAS”, *Proceedings - 2018 International Conference on Field-Programmable Technology, FPT 2018*, pp. 396–399, 2018. DOI: 10.1109/FPT.2018.00082.

[86] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, “From software to accelerators with legup high-level synthesis”, in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2013, pp. 1–9. DOI: 10.1109/CASES.2013.6662524.

[87] S. Colleman, M. Verhelst, and S. Member, “Coprocessor for Image Pixel Processing on FPGA”, vol. 29, no. 3, pp. 1–11, 2021.

[88] G. Zervakis, H. Saadat, H. Mrouch, A. Gerstlauer, S. Parameswaran, and J. Henkel, “Approximate Computing for ML: State-of-the-art, Challenges and Visions”, *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pp. 189–196, 2021. DOI: 10.1145/3394885.3431632.

[89] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, “Fast inference of deep neural networks in FPGAs for particle physics”, *Journal of Instrumentation*, vol. 13, no. 07, P07027–P07027, Jul. 2018. DOI: 10.1088/1748-0221/13/07/p07027. [Online]. Available: https://doi.org/10.1088/1748-0221/13/07/p07027.

[90] vloncar, S. Summers, J. Duarte, N. Tran, B. Kreis, jngadiub, N. Ghielmetti, D. Hoang, E. Kreinar, K. Lin, M. Graczyk, A. A. Pol, ngpaladi, D. Golubovic, Y. Iiyama, Z. Wu, Delon, P. Cretaro, veyron8800, A. Wind, David, GDG, J. Mitrevski, K. Vinogradov, K. Vinogradov, P. Zejdl, S. Nuntaviriyakul, T. Aarrestad, and drankincms, *Fastmachinelearning/hls4ml: Coris*, version v0.6.0, Nov. 2021. DOI: 10.5281/zenodo.5680908. [Online]. Available: https://doi.org/10.5281/zenodo.5680908.

[91] J. P. Wang, S. R. Kuang, and S. C. Liang, "High-Accuracy Fixed-Width Modified Booth Multipliers for Lossy Applications", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 1, pp. 52–60, 2011, ISSN: 10638210. DOI: 10.1109/TVLSI.2009.2032289.

[92] E. Salazar-Villalobos, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Matrix Accelerator*, version v1.1.0, 2022. DOI: 10.5281/zenodo.6413238. [Online]. Available: https://doi.org/10.5281/zenodo.6413238.

[93] A. Rodriguez-Figueroa, L. G. Leon-Vega, and J. Castro-Godinez, *Flexible Accelerator Library: Approximate Convolution Accelerator*, version v0.1.0, 2022. DOI: 10.5281/zenodo.6413243. [Online]. Available: https://doi.org/10.5281/zenodo.6413243.

[94] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities", in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[95] Xilinx Inc., "Zynq-7000 SoC Data Sheet: Overview", *Xilinx*, vol. DS190, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[96] A. Horé and D. Ziou, "Image quality metrics: Psnr vs. ssim", in *2010 20th International Conference on Pattern Recognition*, 2010, pp. 2366–2369. DOI: 10.1109/ICPR.2010.579.

[97] T. Gervens and M. Grohe, "Graph Similarity Based on Matrix Norms", in *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, S. Szeider, R. Ganian, and A. Silva, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 241, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 52:1–52:15, ISBN: 978-3-95977-256-3. DOI: 10.4230/LIPIcs.MFCS.2022.52. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16850.

[98] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: From error visibility to structural similarity", *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004. DOI: 10.1109/TIP.2003.819861.

[99] Y. Lu, J. Zhang, S. Zheng, Z. Li, and L. Wang, *Low error-rate approximate multiplier design for dnns with hardware-driven co-optimization*, 2022. DOI: 10.48550/ARXIV.2210.03916. [Online]. Available: https://arxiv.org/abs/2210.03916.

[100] G. Zervakis, K. Tsoumanis, S. Xydis, N. Axelos, and K. Pekmestzi, "Approximate multiplier architectures through partial product perforation: Power-area tradeoffs analysis", in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15, Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2015, pp. 229–232, ISBN: 9781450334747. DOI: 10.1145/2742060.2742109. [Online]. Available: https://doi-org.ezproxy.itcr.ac.cr/10.1145/2742060.2742109.

[101] I. Kouretas and V. Paliouras, "Simplified Hardware Implementation of the Softmax Activation Function", *2019 8th International Conference on Modern Circuits and Systems Technologies, MOCAST 2019*, pp. 13–16, 2019. DOI: 10.1109/MOCAST.2019.8741677.

[102] S. Ullah, S. S. Murthy, and A. Kumar, "SMApproxLib: Library of FPGA-based Approximate Multipliers", *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018. DOI: 10.1109/dac.2018.8465845.

[103] M. e. a. Adelman, "Faster Neural Network Training with Approximate Tensor Operations", no. NeurIPS, 2018, ISSN: 2331-8422. arXiv: 1805.08079. [Online]. Available: http://arxiv.org/abs/1805.08079.

[104] D. Xu, Z. Zhu, C. Liu, Y. Wang, S. Zhao, L. Zhang, H. Liang, H. Li, and K. T. Cheng, "Reliability Evaluation and Analysis of FPGA-Based Neural Network Acceleration System", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–13, 2021, ISSN: 15579999. DOI: 10.1109/TVLSI.2020.3046075.

[105] J. Castro-Godinez, J. Mateus-Vargas, M. Shafique, and J. Henkel, "Axhls: Design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models", in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[106] G. A. Baker and P. Graves-Morris, "Padé approximants and numerical methods", in *Padé Approximants*, 2nd ed., ser. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1996, pp. 67–121. DOI: 10.1017/CBO9780511530074.005.

[107] P. Wynn, "On the convergence and stability of the epsilon algorithm", *SIAM Journal on Numerical Analysis*, vol. 3, no. 1, pp. 91–122, 1966. DOI: 10.1137/0703007. eprint: https://doi.org/10.1137/0703007. [Online]. Available: https://doi.org/10.1137/0703007.

Appendix A

Vectorisation Wrapper through Template Recursion

Algorithm 10 Vectorisation wrapper for convolution

```
template <int N, int NT, class ENGINE>
struct Vectorise {
  static void Execute(
      typename ENGINE::datatype
          input[NT*ENGINE::outputsize + ENGINE::kernelsize - 1]
               [ENGINE::windowsize],
      typename ENGINE::datatype
          kernel[ENGINE::kernelsize][ENGINE::kernelsize],
      typename ENGINE::datatype
          output[NT * ENGINE::outputsize][ENGINE::outputsize]) {

  /* Important! Inlining the execution allows parallelism */
#pragma HLS INLINE
  ENGINE op{};
  op.Execute(&input[ENGINE::outputsize * (N - 1)], kernel,
             &output[ENGINE::outputsize * (N - 1)]);
  /* Continue Loop - The next i = i - 1 */
  Vectorise<(N - 1), NT, ENGINE>::Execute(
      input, kernel, output);
  }
};

template <int NT, class ENGINE>
struct Vectorise<0, NT, ENGINE> {
  static void Execute(
      typename ENGINE::datatype
          input[NT*ENGINE::outputsize + ENGINE::kernelsize - 1]
                [ENGINE::windowsize],
      typename ENGINE::datatype
          kernel[ENGINE::kernelsize][ENGINE::kernelsize],
      typename ENGINE::datatype
          output[NT * ENGINE::outputsize][ENGINE::outputsize]) {
/* Important! Inlining the execution allows parallelism */
#pragma HLS INLINE
  /* Do Nothing (terminate loop) */
  }
};
```

Appendix B

Convolution PE Description with Vectorisation

Algorithm 11 Final PE description with vectorisation and replacement

```
/* Matrix dimensions */
static constexpr int O = 2;
static constexpr int K = 3;

/* Define data types */
static constexpr int I = 1;
static constexpr int W = 16;
static constexpr int D = 4;
using DataType = ap_fixed<W, I>;

/* Define the number of PEs or execution units */
static constexpr int N = 4;

/* Define approximate operators */
using Multiplier =
    axc::arithmetic::lsbdrop::Multiply<DataType, W, I, D>;
using Adder =
    axc::arithmetic::lsbdrop::Add<DataType, W, I, D>;

/* Specialise the GEMMA PE without constructing it */
#ifdef USE_WINOGRAD
using Engine = ama::hw::convolvers::Winograd<DataType, K, O,
      Adder, Multiplier>;
#else
using Engine = ama::hw::convolvers::Spatial<DataType, K, O,
      Adder, Multiplier>;
#endif

/* Vectorise replicating the engine by N times */
/* Use the PE over image IX with kernel IK with output OY */
ama::hw::ParallelConvolver<N, N, Engine>::Execute(
    IX, IK, OY);
```

Appendix C

Function Approximation

This work has also made progress in function approximation, particularly, in exponential-based functions. This will be relevant for future work.

Softmax is an activation function optimisable by tweaking the exponential function $f(x) = e^x$, which is a bijective function whose domain is $\mathbb{R}$ (see Figure C.1). One of the possible optimisations is to define the function for a custom domain, which is a subset of $\mathbb{R}$, removing those elements not required for the DL operations, in particular, by the FCL, which is commonly preceding a softmax.

Remembering that the FCL can be expressed as (2.2) and assuming a numerical representation that supports a uniformly distributed discrete set within the domain $S =]-1{,}1[$, an element of the output vector can be expressed as

$$y_i = \mathbf{w}_i \cdot \mathbf{x} + b_i \tag{C.1}$$

where $\mathbf{w}_i$ is the i-th row vector from the matrix $\mathbf{W}$ and $\cdot$ is the dot-product between vectors, expressed as $\mathbf{w}_i \cdot \mathbf{x} = \sum_{j=1}^{k} w_{ij} x_j$. It means that each output element involves k products and k additions including the bias.

The computation is numerically vulnerable to the additions, risking overflows. The problem was previously addressed in 3.2, which keeps the output of every element of the FCL within $S \in]-1, 1[$. Knowing that the domain of y_i is constrained and given by S, S *can also give the exponential function domain.* Since S is a uniformly distributed discrete set, quantised in $\beta - 1$ bits, an immediate consequence is that the function can also be defined by a Look-up Table (LUT) with a number of points equal to the number of elements of the set without incurring an under or over discretisation. However, it will imply a resource utilisation greater than an application needs.

Considering that the input of the exponential function is restricted to S and suitable for LUTs, it is possible to explore more approximations to lower the resource consumption and speed up the processing. This research considers the Taylor series, Padé approximants, and piece-wise interpolation within the approximations.

C.0.1 Taylor series

A Taylor series consists of a function approximation given by the infinite sum of elements that are expressed in terms of the target function's derivatives at a single point. For the exponential function, the Taylor series centred in $a = 0$ is

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \ldots, \forall x \in \mathbb{R} \tag{C.2}$$

where a is the point where the function's derivative is centred [71], and it converges everywhere.

C.0.2 Padé approximant

The Padé approximant is a function that approximates through the reason of two polynomials [106]. For a function $f(x)$, there is an unique approximant of order m,n

$$R_{m,n}(x) = \frac{P_m(x)}{Q_n(x)} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m z^m}{b_0 + b_1 x + b_2 x^2 + \cdots + b_n x^n}, x \in \mathbb{C} \tag{C.3}$$

where P and Q are polynomials of degrees no more than m and n. When $n = 0$, the Padé approximant becomes the same as the m-order Taylor series. Wynn's algorithm is one of the methods to compute the Padé's approximant [107].

C.0.3 LUT-based Piece-wise Interpolation

This method consists in sampling the function in uniformly-distributed points and computing the best-fit polynomial between the points. For instance, a linear polynomial requires two points to compute, whereas a quadratic requires three [72]. Figure C.1 shows how a linear interpolation fits the e^x function by taking eight samples and performing linear interpolation.

The computation of the segments can be either computed at runtime or at compute time. At runtime, the slope and intercept are computed as

$$m_p = \frac{y_{p_1} - y_{p_0}}{x_{p_1} - x_{p_0}}, b_p = y_{p_1} - m_p x_{p_1} \tag{C.4}$$

such that $f_p(x) = m_p x + b_p, x_{p_0} \leq x \leq x_{p_1}$, where $(x_{p_0}, y_{p_0}), (x_{p_1}, y_{p_1})$ are the points before and after the point of interest x_p, respectively. In this case, the computation of the point requires: 1) storing the points in a LUT, 2) computing the linear equations, and 3) computing the value of interest. At compute time, instead, it computes 2) offline and stores the slopes and intercepts in the LUT, shortening the path from 1) to 3).

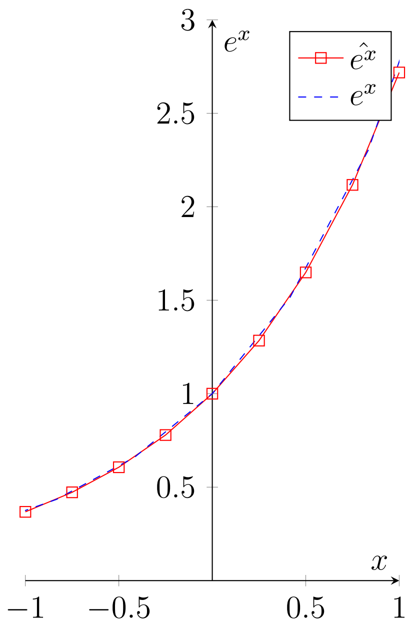


Figure C.1: Piecewise representation by doing eight samples within the domain S and applying a linear interpolation

Moreover, for the sake of avoiding unwanted divisions while computing the indices of the slope-intercept pairs required for the computation, the number of points can be a power of two, such that the division becomes a bit-shift in such a way that

$$p = x' \gg P \implies m_p = M[p], b_p = B[p] \tag{C.5}$$

where P is the number of points (power of two), x' is the quantised value of x in fixed-point, M and B are the LUTs for the slope and intercept, respectively.

Appendix D

Operator Approximation

This appendix shows the results after benchmarking the approximate operators.

D.1 Resource consumption

This section shows the resource consumption estimated by Vivado HLS. As the approximation increases, resource consumption decreases concerning the exact version of the operation, suggesting a gain in area.

The data types utilised are `half`, a 16-bit floating-point provided by Vivado. Moreover, `q16` and `q8` are data types using fixed-point numbers with a data width of 16 and 8 bits, respectively. These data types were selected to evaluate the approximations after bit pruning and logic change in the less significant bits (LSB).

The results are obtained after sampling 100 to 1 million samples for error quantification.

The following figures show how the `half` data type presents a greater consumption in contrast than the approximate values, even by using fixed-point arithmetic.

D.1.1 Approximate Addition

Figure D.1 shows the resource consumption for the addition by modifying the data type. Fixed-point arithmetic consumes less resources than the half-precision floating-point (`half`).

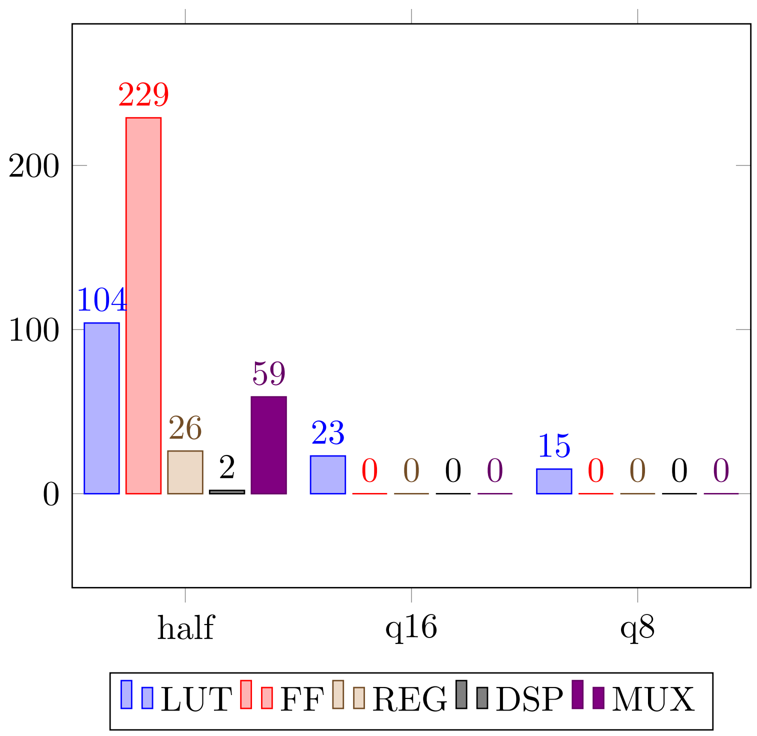


Figure D.1: Addition resource consumption without introducing approximations

Figures D.2 and D.3 show how the approximations reduce the resource consumption as the number of approximate bits increases.

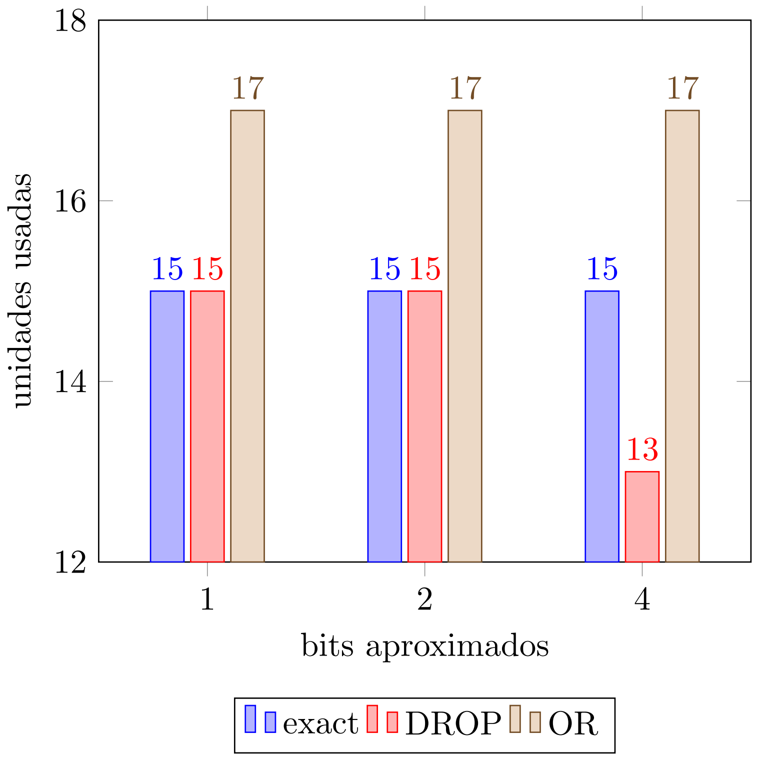


Figure D.2: LUT consumption in an 8-bit adder

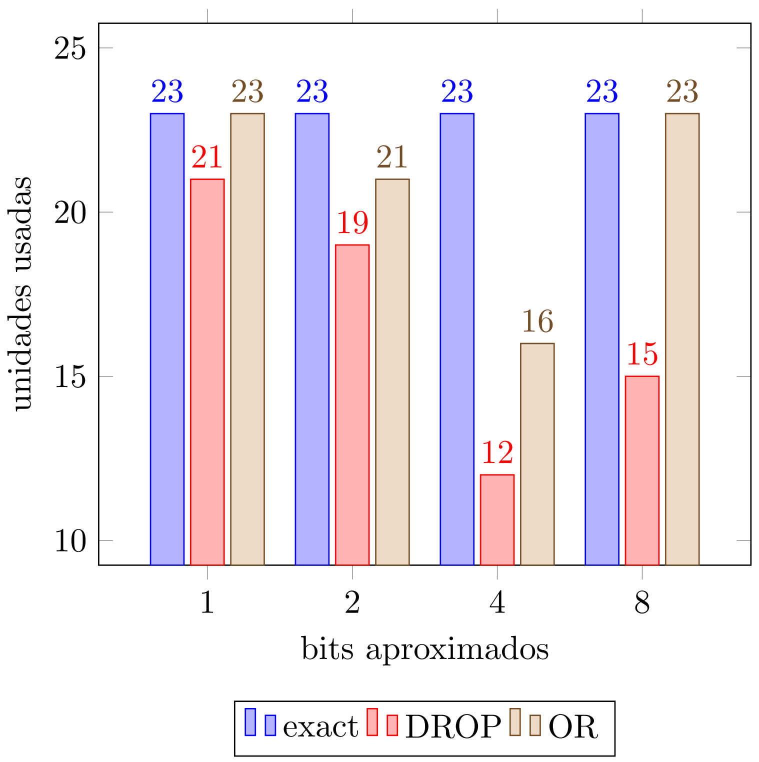


Figure D.3: LUT consumption in a 16-bit adder

D.1.2 Approximate Multiplication

Figures D.4 and D.5 show the same tendency to reduce resource consumption as the approximation increases. In the 16-bit case, there is an exception, utilising not only LUTs but FFs. Figure D.6 shows the FF usage by multiplication. In the exact 8-bit configuration, there is no FF usage. Nevertheless, when using 16-bit data width and 8-bit approximation, the consumption becomes the same as in the 8-bit configuration for the LSB drop, suggesting a correct behaviour.

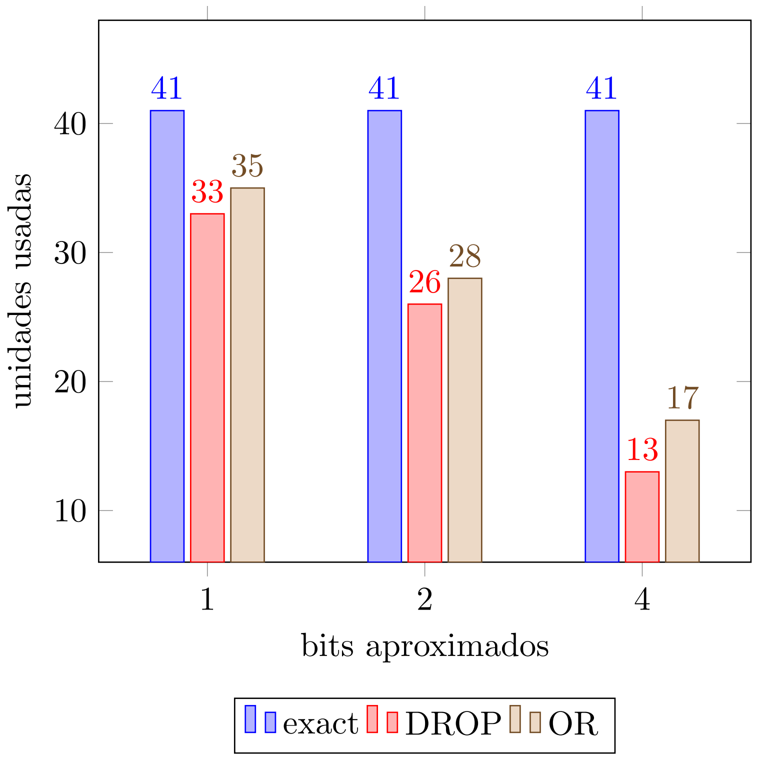


Figure D.4: LUT consumption in an 8-bit multiplier

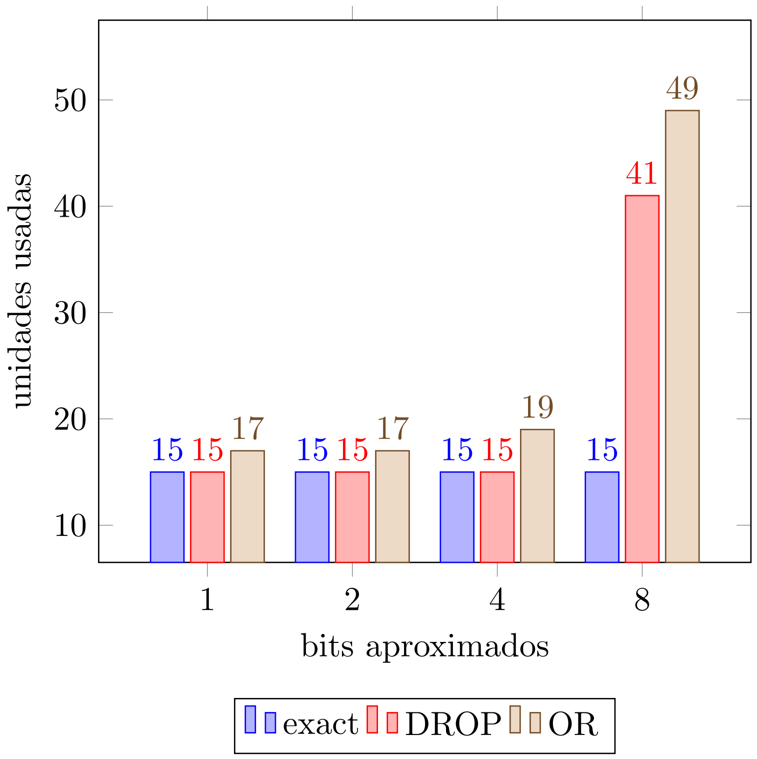


Figure D.5: LUT consumption in a 16-bit multiplier

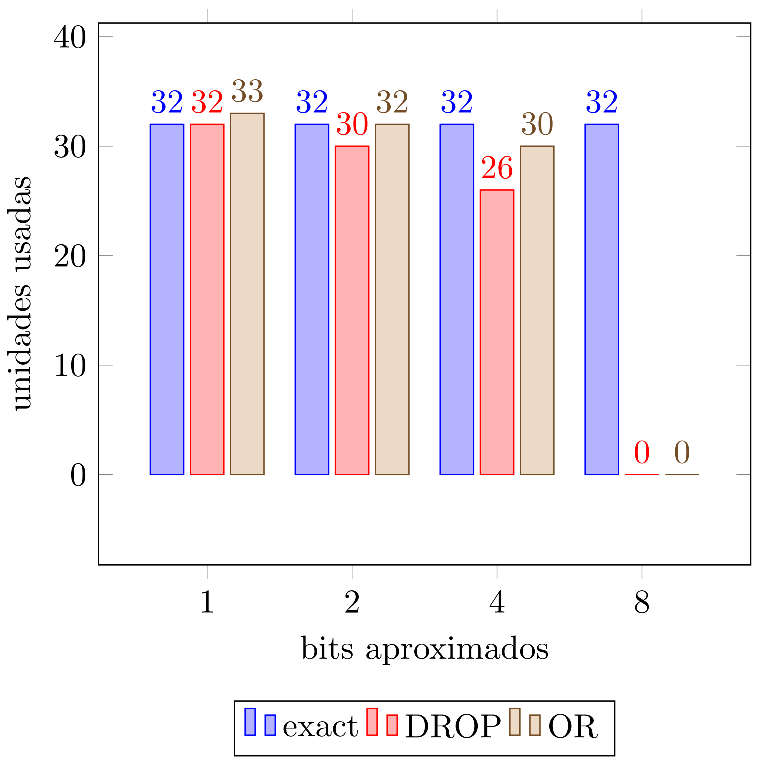


Figure D.6: FF consumption in a 16-bit multiplier

D.2 Latency

D.2.1 Approximate Addition

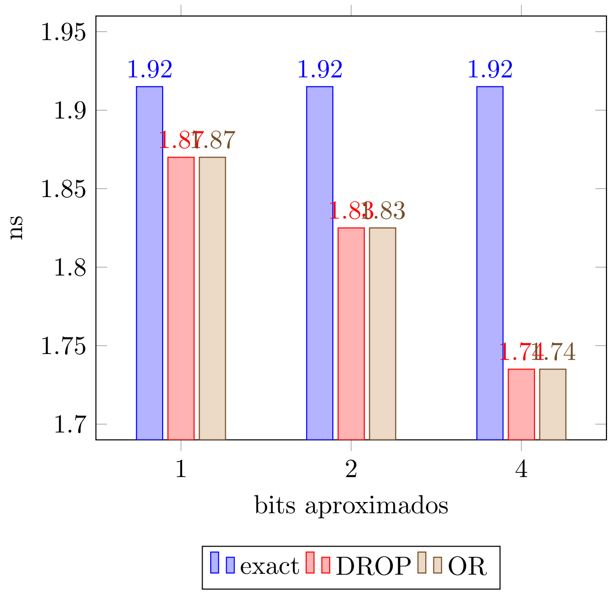


Figure D.7: Latency of an 8-bit *Adder*

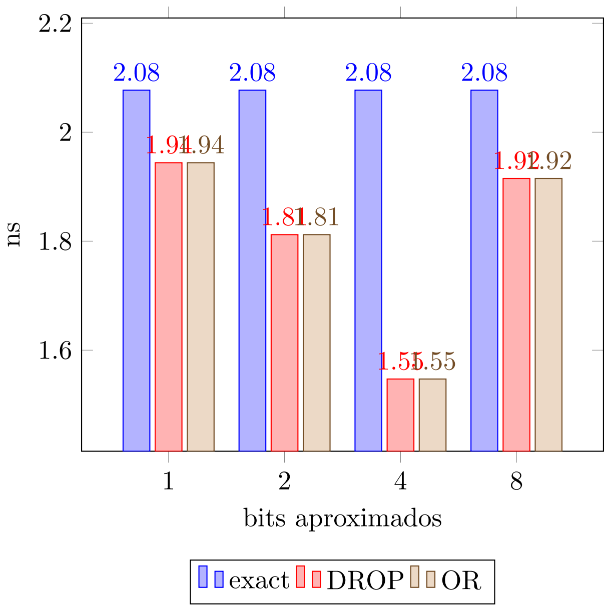


Figure D.8: Latency of a 16-bit *Adder*

D.2.2 Approximate Multiplication

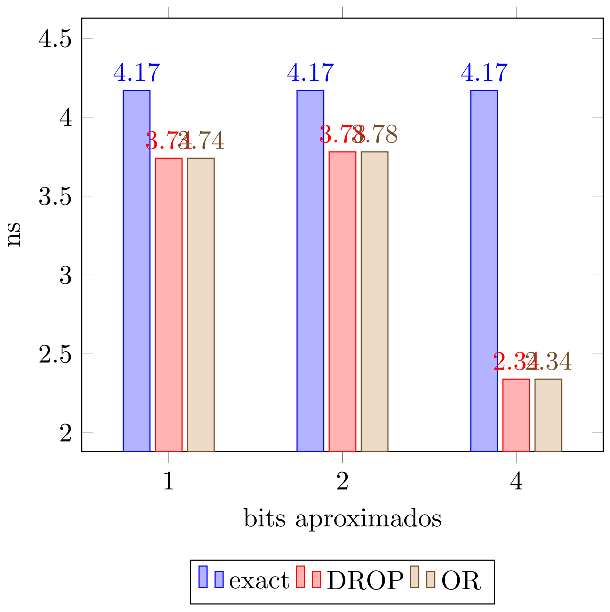


Figure D.9: Latency of an 8-bit *Multiplier*

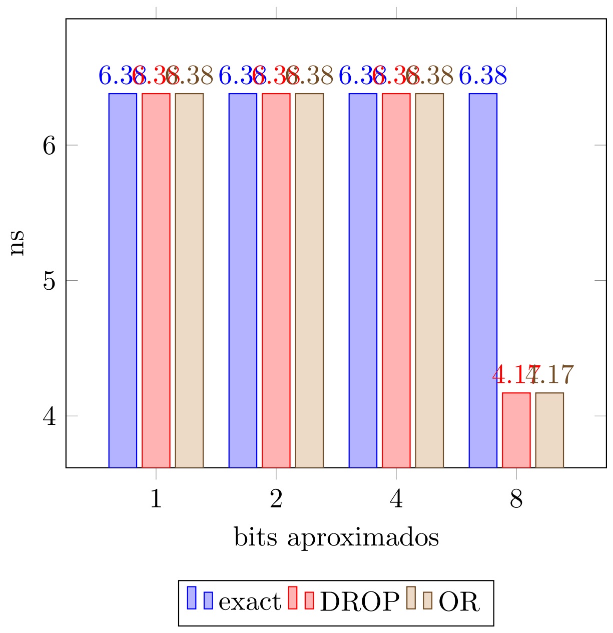


Figure D.10: Latency of a 16-bit *Multiplier*

D.3 Approximation Error

This section shows the error introduced by the approximations. Apart from the approximations, the fixed-point representation is an approximation itself, introducing errors (quantisation errors). These errors have been quantified and represented through the blue curves in the error plots. They work as a reference for further approximation errors.

It is critical to consider that the floating-point also introduces errors but they will be considered as the baseline or *exact* version in this analysis.

Using 500 samples, the bin length is 0.2%, and the error introduced every 5 bins is 1%. Figures D.11, D.12 and D.13 show that increasing the data size, the error decreases close to the quantisation error. In the 16-bit case, the error is as small as the quantisation error, fitting within the first 0.2% bin.

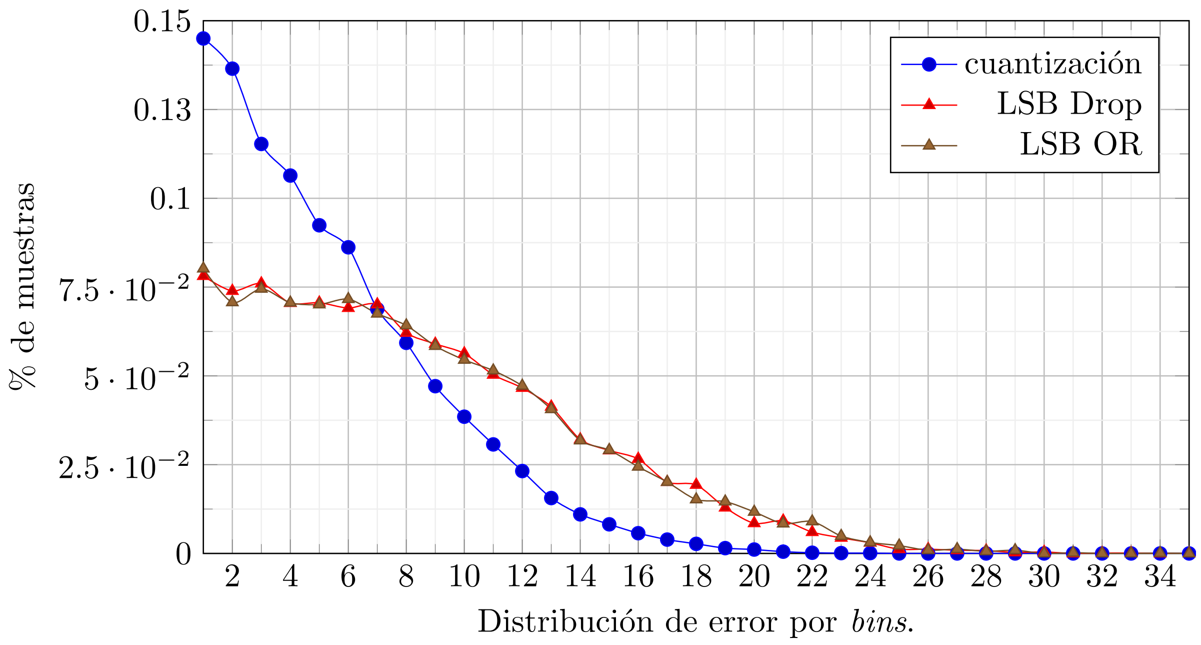


Figure D.11: Error distribution with 8-bit data width

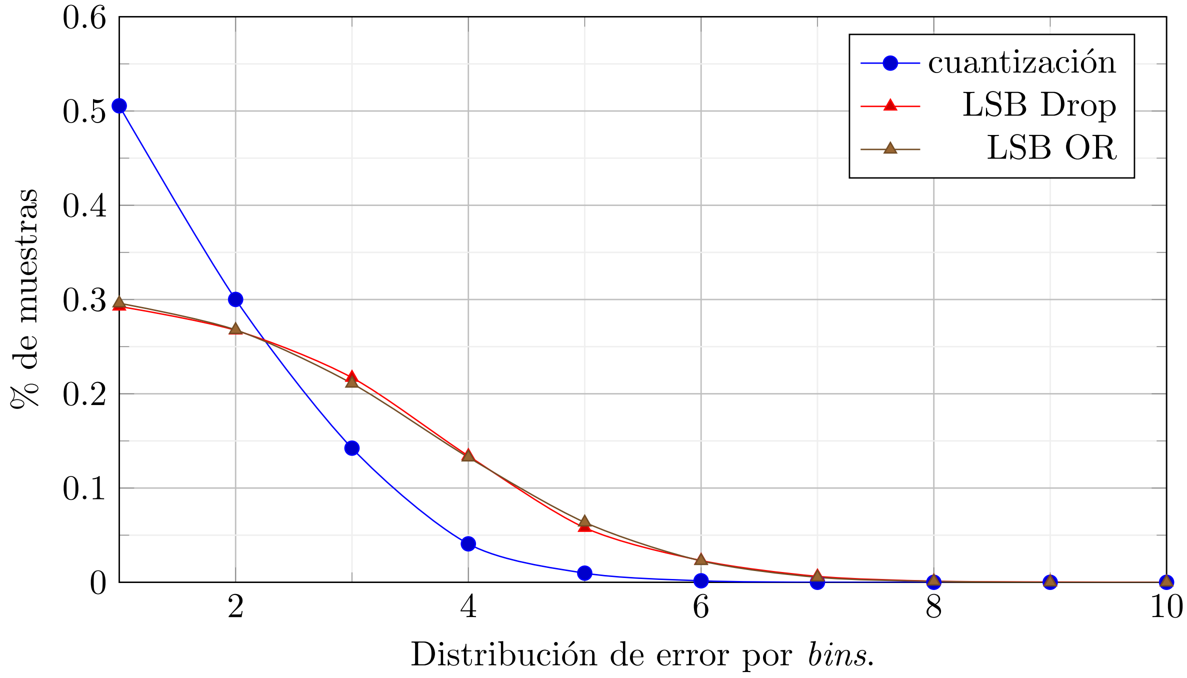


Figure D.12: Error distribution with 10-bit data width

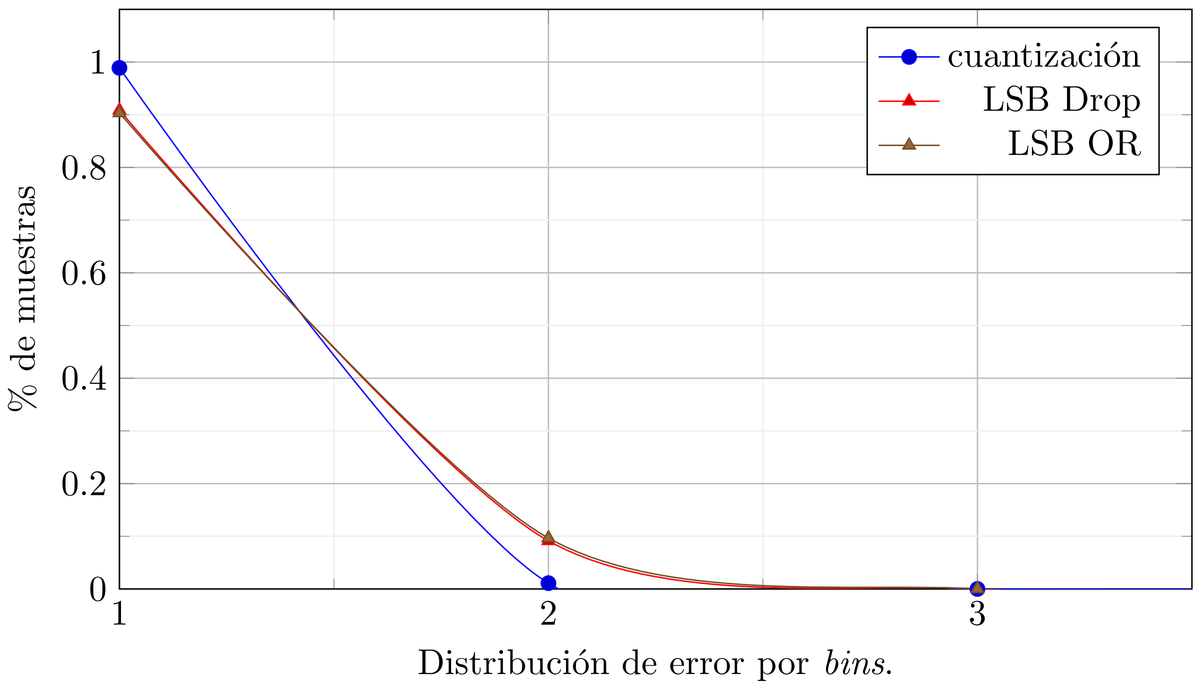


Figure D.13: Error distribution with 12-bit data width