

École polytechnique de Louvain

Hardware-Software Co-Design of an FPGA-based Transformer for Embedded Machine Learning

Authors : **Arnaud GALLEZ, Antoine VANDEN CLOOSTER**
Supervisors : **Jean-Didier LEGAT**
Readers : **David BOL, Jean-Didier LEGAT, Adrian KNEIP**
Academic year 2021-2022
Master [120] in Electrical Engineering

Abstract

Today embedded devices are all around us. With the addition of Machine Learning (ML) in such devices, a wide range of applications is possible. These platforms however generate a great amount of data to be processed. A current challenge in IT consists of reducing the data circulating on communication networks. Indeed these grow year after year and represent a non-negligible worldwide energy consumption. *Edge computing* proposes to tackle this problem by handling data *at the edge* without resorting to computing servers. The research for efficient hardware accelerators is a good lead in this direction.

The Transformer network is a recent architecture that could handle multiple tasks at once, reducing the need for different model-targeted accelerators and co-processors. The multimodal training possibility of this architecture, coupled with the current need for smart sensors capable of handling data by themselves in an acceptable latency are the reasons that motivate this work. This thesis aims indeed at deploying a Multi-Head Attention (MHA) block from the Transformer architecture on the DE10-Nano embedded device. After having identified the bottlenecks of the MHA, a hardware accelerator is proposed to go over them.

From an initial Software Floating-Point (FP) implementation taking 121.7 ms for one inference, we go to an accelerated quantized Software-Hardware co-designed system taking 13.29 ms on the same input, accelerating the process by 89%. A software-only integer implementation is also presented, reducing the initial time by 79.73% and thus demonstrating the value of quantization.

Acknowledgements

First of all, we would like to thank our supervisor Pr. Jean-Didier Legat for his availability as well as the trust and freedom he has granted us during the year.

We would also like to thank Adrian Kneip for his co-supervision of the thesis. Furthermore we want to express our gratitude for his availability and precious advises along the year.

We would also like to acknowledge the last member of our jury, Pr. David Bol for accepting to read our thesis.

A particular thanks to our reviewers for their rigorous proofreading and constructive returns.

Finally we would like to thank our families and friends for their moral support throughout the year.

Contents

List of Figures	x
List of Tables	xiv
Introduction	1
I TinyML: CNN deployment onto an embedded platform	5
Introduction	7
1 Machine Learning and Embedded Deployment: A State-Of-The-Art	9
1.1 Machine Learning	9
1.1.1 What is a Neural Network	9
1.1.1.1 Generalities	9
1.1.1.2 Softmax	12
1.1.1.3 Convolutional Neural Network (CNN)	12
1.1.1.4 Quantization	15
1.1.2 Training a neural network	17
1.1.2.1 Supervised learning	18
1.1.2.2 Dataset split	20
1.1.2.3 Training with quantization	21
1.2 Embedded platform: DE10-Nano	21
1.2.1 HPS	21

1.2.2	FPGA	22
1.2.3	System and communication	22
1.3	Tensorflow Lite for Microcontrollers (TFLM)[26]	24
1.3.1	Principles and promises	25
1.3.2	Why TFLM is different from TFLite	27
1.3.3	How does it work[31]	28
1.3.4	From Tensorflow to TFLM model	30
2	CNN deployment onto DE10-Nano using TFLM	33
2.1	Experimental setup	33
2.2	Results	35
2.2.1	Training	35
2.2.2	Inference	36
2.2.2.1	Binary size	36
2.2.2.2	Timing	37
II	Transformers: Deployment and acceleration onto an embedded platform	41
	Introduction	43
3	Transformers	45
3.1	Transformer	45
3.1.1	Natural Language Processing (NLP)	45
3.1.2	Architecture	49
3.1.2.1	Input embedding and positional encoding	51
3.1.2.2	Encoder	54
3.1.2.3	Decoder	62
3.1.2.4	Final output processing	63
3.1.3	Types of models	63
3.1.4	Bottlenecks	64
3.1.5	Existing acceleration techniques	65

3.1.5.1	Quantization	66
3.1.5.2	Algorithmic techniques	67
3.1.5.3	Hardware techniques	72
4	From Transformer training to Software Embedded deployment	77
4.1	Training the transformer	77
4.1.1	Building the model	78
4.1.2	Dataset	78
4.1.3	Network and Training parameters	79
4.1.4	Simplifications and consequences	81
4.1.5	Result of the training	81
4.2	Deploying the Transformer onto DE10-Nano	82
4.2.1	Transformer with TFLM: an impossible story	82
4.2.1.1	Flex operations	82
4.2.1.2	Missing operations from TFLite	83
4.2.1.3	The end of TFLM	84
4.2.2	MHA with CMSIS-NN: a possible story	84
4.3	Results	85
4.3.1	Quantization on Softmax	85
4.3.2	Timing of MHA	86
5	MHA hardware acceleration	91
5.1	Hardware-Software system architecture	92
5.1.1	Tasks splitting	92
5.1.2	Global overview of the system	95
5.2	Dot Product acceleration	96
5.3	Digital circuit	103
5.3.1	Memories	104
5.3.2	Control unit	105
5.3.3	Linear Layer	106
5.3.3.1	Head module	108
5.3.3.2	Block_head module	109

5.3.3.3	Systolic_array module	109
5.3.3.4	S_a_init module	113
5.3.3.5	PE module	115
5.3.4	Pipelining	116
5.3.4.1	Pipelining timing	117
5.3.4.2	Logic	118
5.4	Final results	119
Conclusion		125
Bibliography		I
A Tensorflow & Keras		XI
A.1	Execution graph vs Eager execution	XII
A.1.1	Different model types	XII

Acronyms

ADC Analog-Digital Converter

ALM Adaptive Logic Module

API Application Programming Interface

CMSIS Common Microcontroller Software Interface Standard

CNN Convolutional Neural Network

CPU Central Processing Unit

DAG Directed Acyclic Graph

DSP Digital Signal Processor

FFT Fast Fourier Transform

FIR Finite Impulse Response

FP Floating Point

FPGA Field Programmable Gate Array

GLUE General Language Understanding Evaluation

GRU Gated Recurrent Unit

HDL Hardware Description Language

HPS Hard Processor System

IDE Integrated Development Environment

Int8 8-bits integer

IP Intellectual Property

LSB Least Significant Bit

LSTM Long Short Term Memory

LUT Look-Up Table

MHA Multi-Head Attention

ML Machine Learning

MLP Multi-Layer Perceptron

MM Memory Mapped

NLP Natural Language Processing

NN Neural Network

NRE Non-Recurring Engineering

OS Operating System

PE Processing Element

PIO Parallel Input/Output

PPA Power - Performance - Area

RNN Recurrent Neural Network

SA Systolic Array

TFLite Tensorflow Lite

TFLM Tensorflow Lite for Microcontrollers

List of Figures

1.1	Examples of Neural Network structures	10
1.2	RELU(x), Sigmoid(x) and Tanh(x) activation functions [11]	11
1.3	Convolutional Neural Network (CNN) global structure	13
1.4	Convolution operation between a kernel and an input feature map [12]	14
1.5	Maximum and average pooling processes	15
1.6	Importance of rightfully choosing the learning rate [18]	19
1.7	Overfitting [22]	20
1.8	Bridges available between the HPS and FPGA	23
1.9	Different parts of the board and their connections	24
1.10	Differences between Tensorflow versions	25
1.11	Different elements and their relations in the TFLM framework	28
1.12	Conversion of the Tensorflow model up to the TFLM one. This happens in two steps. First the model to the TFLite version, then the .tflite file is converted to a C++ array.	31
2.1	CNN structure	34
2.2	Example of mnist sample for the digit "7"	34
2.3	Losses and accuracies for the training of the CNN	36
2.4	Timing of the models summarized in table 2.2. Models G,H,I are to be measured with respect to the right axis, as their execution times are orders of magnitude lower than the ones on DE10.	38
3.1	RNN inner structure	47
3.2	Global overview of the transformer [1]	50

3.3	Dataflow in the Transformer architecture	51
3.4	Example of Scaled Dot-Product Attention : From sentence to embeddings	52
3.5	Embbded vectors[46]	53
3.6	Positional encoding	55
3.7	Scaled Dot-Product Attention	56
3.8	Example of Scaled Dot-Product Attention: bring context to the embedding	57
3.9	Multi-Head Attention	60
3.10	Bottleneck identification [58]	65
3.11	Column Balanced Block Pruning [62]	67
3.12	Dot product accelerator for sparse matrix with column balanced block-wise sparsity pattern	68
3.13	Encoder acceleration	69
3.14	Results of the Column Balanced Block Pruning	70
3.15	Flowchart of the multihead attention mechanism	71
3.16	Data flows for the proposed algorithms avoiding data marshalling operations	72
3.17	Systolic Array of shape (4×4)	73
3.18	Processing Element	74
4.1	Training curve	82
4.2	Behavior of the softmax function depending on the implementation for two different inputs.	86
4.3	Profiling of the execution with the time taken by the different steps for a full FP implementation (no DSP co-processors) of the MHA. . .	88
4.4	Broken down execution time for the int8 implementation of the MHA. .	89
4.5	Different paradigms to map the software execution onto a multi-core platform	90
5.1	Global overview of the hardware design	95

5.2	Overview of the complete system, the different parts and their inter- actions	96
5.3	Input sequence	99
5.4	SA for matrix multiplication acceleration	100
5.5	Accelerated addition	101
5.6	Influence of N on the system	102
5.7	Hierarchy of the digital circuit	103
5.8	Memories properties	104
5.9	Control Unit	105
5.10	<i>Linear Layer</i> module	107
5.11	<i>head</i> module	108
5.12	<i>Block_head</i> module	109
5.13	Boundary conditions of a systolic array for the case $I = 1, J = 1$. .	110
5.14	Boundary conditions of a systolic array for the case $I = 1, J > 1$. .	111
5.15	Boundary conditions of a systolic array for the case $I > 1, J = 1$. .	111
5.16	Boundary conditions of a systolic array for the case $I > 1, J > 1$. .	112
5.17	Matrix flattened in a register	112
5.18	Structure of the systolic array module	114
5.19	Structure of the <i>s_a_init</i> module	114
5.20	Architecture of the <i>pe</i> module	116
5.21	Pipelining in the <i>main</i> module	117
5.22	Area (Measured) and Latency (Measured and theoretical) depending on N. The theoretical curve comes from Section 5.2.	120
5.23	Breakdown of execution time when reading memory from FPGA before computing. <i>Waiting FPGA</i> denotes the time during which the FPGA computes and writes outputs to memory.	122
5.24	Breakdown of execution time when live - reading memory from FPGA during Softmax-matmul operation 'P.' stands for 'Projections'	123

5.25	Comparison of the results for the different implementations. <i>Read</i> stands for the implementation where the HPS reads the projections computed by the FPGA before processing the matrix multiplications. <i>No Read</i> designates when memories are live-read during matrix computations on the HPS.	124
A.1	Different Keras API with the amount of code needed for each	XIII

List of Tables

2.1	Sizes comparison depending on ops resolver and model type	37
2.2	Labelling of the different models and configurations	38
3.1	Main NLP tasks and their meaning. This list is not exhaustive. . . .	46
3.2	Results using the quantized RoBERTa architecture. No accuracy loss is observed, a slight increase in accuracy is even observed for the quan- tized version.	66
3.3	Speedup between the quantized model of I-Bert and its FP32 imple- mentation [15].	66
3.4	Results from [63]	72
3.5	a and b input vectors depending on the processing step of a (4×4) SA	75
4.1	Parameters introduced in Section (3.1.2) used for the network where ($E + D$) stands for <i>Encoder + Decoder</i>)	80
4.2	Results for the two presented techniques to approximate Softmax function	85
5.1	Key metrics to evaluate digital designs	92
5.2	Different possible system configurations with their respective advan- tages and drawbacks	94
5.3	Global parameters of the Transformer	97
5.4	Parameters of the Transformer for a head i	97
5.5	Summary of the different SA designs	102
5.6	linked I/O of each PE according to its position for the case $I = 1, J = 1110$	

- 5.7 linked I/O of each PE according to its position for the case $I = 1, J > 1$ 111
- 5.8 I/O of each PE according to its position the case $I > 1, J = 1$. . . 111
- 5.9 linked I/O of each PE according to its position for the case $I > 1, J > 1$ 112

Introduction

Today embedded devices are all around us. They can perform a wide variety of different applications such as sensing, communicating, entertaining, etc. With the addition of Machine Learning (ML) in such devices, possibilities are extended. Applications like autonomous vehicles, smart sensors and cities are now possible. However, implemented ML models are often designed to handle only one task. This means as new model has to be designed for each of them. Also, efficient hardware accelerators and coprocessors need to be re-engineered for every specific architecture. A network capable of handling multiple tasks could be interesting regarding the following points:

1. Some sensors could be merged together, leading to a lower production of integrated devices.
2. Hardware accelerators could be designed once and for all, leading to lower Non-Recurring Engineering (NRE) costs.
3. A hardware accelerator specifically designed for such network could be optimized compared to the use of a generic Digital Signal Processor (DSP) with regards to energy consumption, latency and chip area utilization.

The *Transformer* could be a solution to the presented points. This architecture was initially proposed by the Google Brain Team in 2017 [1]. It was first targeted for Natural Language Processing (NLP) tasks, i.e. tackling problems involving language often in the form of text. After having applied this architecture to these tasks some researchers tried to generalize the network to other applications as it often has been done with other architectures. For example the Convolutional Neural Network (CNN),

initially designed for image recognition tasks, has been applied to NLP or recommendation tasks [2][3]. This has been done for Transformers with for instance ViT [4] that solves vision tasks. Attempts were also made to train multi-task transformers such as ViLBERT, able to handle textual and vision tasks simultaneously like image captioning, visual question answering or caption-based image retrieval. UniT [5] is one of the most versatile Transformer networks that jointly trains seven different tasks with one architecture. It even performs sometimes better than models specifically trained for one of these tasks. Even if it does not require as much parameters as seven different networks, its size can be huge. Indeed it is common to observe Transformers with several millions or even billions of weights [1][5][6].

Another present challenge in IT consists of reducing the data circulating on communication networks. Indeed these grow year after year and represent a non-negligible energy consumption [7]. For battery-powered sensing networks (wildfires detection, endangered species monitoring, etc.) this limitation becomes critical as energy resources are scarce. A paradigm proposing to solve this problem is called *edge computing*. This philosophy calls for a processing of data *on device* without relying on computing servers.

The size of the Transformer leads to challenges for the deployment on edge devices as these last are constrained in memory and computing power. Also, inference on networks of such size can be very long. This latency causes problems in real-time embedded applications such as image recognition in autonomous vehicles or discussion translation.

The multimodal training possibility of the Transformer, coupled with the current need for smart sensors capable of handling data by themselves in an acceptable latency are the reasons that motivate this work. This thesis aims indeed at deploying a Transformer on the DE10-Nano embedded device [8]. After having identified the bottlenecks of the architecture, a hardware accelerator is proposed to go over them.

This work is divided in two main parts. Part I aims at studying the usability of

Tensorflow Lite for Microcontrollers (TFLM) on the DE10-Nano. TFLM is a library designed to ease the deployment and inference of ML models on microcontrollers. To assess this usability, a Convolutional Neural Network (CNN) is used as toy example. In this part:

- Chapter 1 gives the necessary background of ML as well as a quick overview of the CNN architecture. It then addresses the training process, by which a Neural Network (NN) is tuned to achieve a specific task. It lastly presents the used embedded platform and the TFLM library.
- Chapter 2 gives the results we obtain by deploying a CNN onto the DE10-Nano using TFLM.

The CNN can be ported to the DE10-Nano using TFLM and proves that TFLM is usable with the board. The Transformer however cannot be deployed on embedded devices using this library because its architecture is intricate to the point that TFLM cannot handle some of its operations.

For this reason only part of the network (called Multi-Head Attention (MHA)) is manually mapped using Vanilla C. Because of the network size, we choose to use integer-only operations as well as quantized weights and activations. The MHA is then accelerated using Software/Hardware (SW/HW) co-design. This is the focus of Part II that is organized as follows:

- Chapter 3 is dedicated to the Transformer architecture. It first presents the tasks that can be handled by this network, then develops its architecture in details. The bottlenecks are finally identified before giving existing acceleration schemes to tackle them.
- Chapter 4 addresses the Transformer implementation using only software. It follows a top-down approach, first introducing the training process then the steps that are followed to map the architecture on the board. As explained before, this is made only for part of the architecture. Quantization and timing results are finally presented.

- Chapter 5 finally presents our hardware acceleration scheme. It first explains the structure of the complete SW/HW co-design before getting down to the accelerator architecture in itself. The obtained results are then presented and compared to the software-only version of the algorithm.

A conclusion finally wraps up the work as well as giving further research tracks. All the codes of our implementations are available at [9].

Part I

TinyML: CNN deployment onto an embedded platform

Introduction

This part aims at testing the compatibility between Tensorflow Lite for Microcontrollers (TFLM) and the DE10-Nano board. A Convolutional Neural Network (CNN) trained for the digit recognition task is used as a toy example. This part is organized in two chapters.

Chapter 1 presents a State-Of-The-Art of the Machine Learning (ML) and embedded deployment concepts. It first introduces ML by explaining some basic concepts such as neurons, Softmax, quantization, Neural Network (NN) and Convolutional Neural Network (CNN). In a second time the training process of a NN is described, followed by a detailed presentation of the embedded platform used throughout the work i.e. the DE10-Nano. The specifications of its two main parts i.e. Hard Processor System (HPS) and Field Programmable Gate Array (FPGA) are given as well as of the communication between them. TFLM, a library designed by Google to execute ML algorithms onto embedded platforms, is finally presented.

Chapter 2 covers the actual deployment of the CNN architecture on the DE10-Nano. It first presents the experimental setup after which the results are given. A comparison is performed between different model, TFLM and board configurations.

Chapter 1

Machine Learning and Embedded Deployment: A State-Of-The-Art

This Chapter aims at presenting the concepts used in this work. It is divided in three sections that first address the basics of Machine Learning (ML), then present the embedded platform we use throughout the work and finally introduce the Tensorflow Lite for Microcontrollers (TFLM) library.

1.1 Machine Learning

1.1.1 What is a Neural Network

1.1.1.1 Generalities

In Machine Learning (ML), one designs a *network* constituted of connected *neurons* composed of *weights* and *activation functions*. These are represented in Figure 1.1a. The operations performed by a neuron are the following and expressed in Equation 1.1:

1. Taking an input vector $\mathbf{x} \in \mathbb{R}^P$, it computes a weighted sum of the inputs. Each vector entry is multiplied with an associated *weight* belonging to a weight

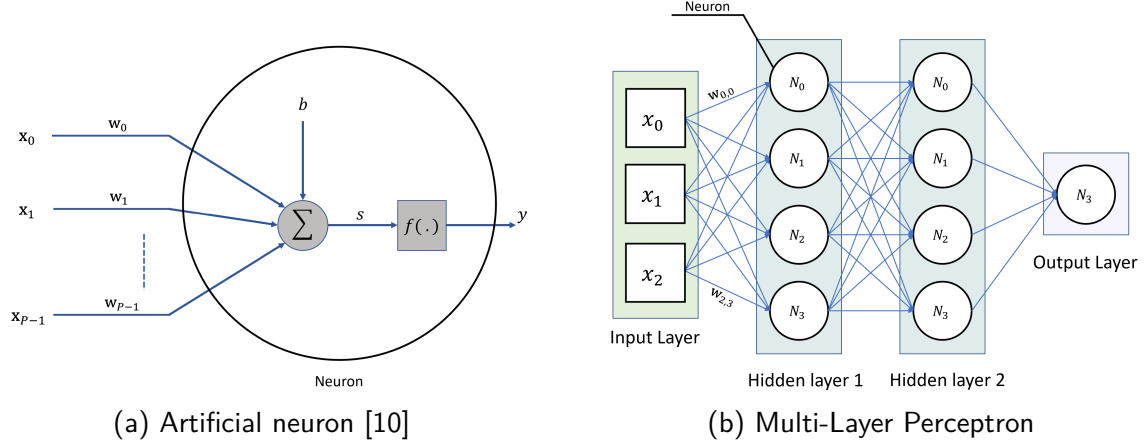


Figure 1.1: Examples of Neural Network structures

vector $\mathbf{w}$.

2. A bias b is added to the output of this weighted sum.
3. The result of the two previous steps is passed through a non-linear *activation function*.

$$s = \sum_{p=0}^{P-1} x_p \cdot w_p + b \quad (1.1)$$

$$y = f(s)$$

Where x_p and w_p are elements of the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}$ respectively. The result of the weighted sum s is called the *activation*, $f(\cdot)$ is called the *activation function* and y is the output.

Activation function

The activation function is put after the weighted sum to add non-linearity to the network. If the said network was composed only of weighted sums, it would not be able to *learn* non-linear relations. Three of the most commonly used activation functions

are depicted in Figure 1.2 and their expressions are given in Equation 1.2.

$$\begin{aligned}\text{ReLU}(x) &= \max(0, x) \\ \text{Sigmoid}(x) &= \frac{1}{1 + \exp^{-x}} \\ \text{Tanh}(x) &= \tanh(x)\end{aligned}\tag{1.2}$$

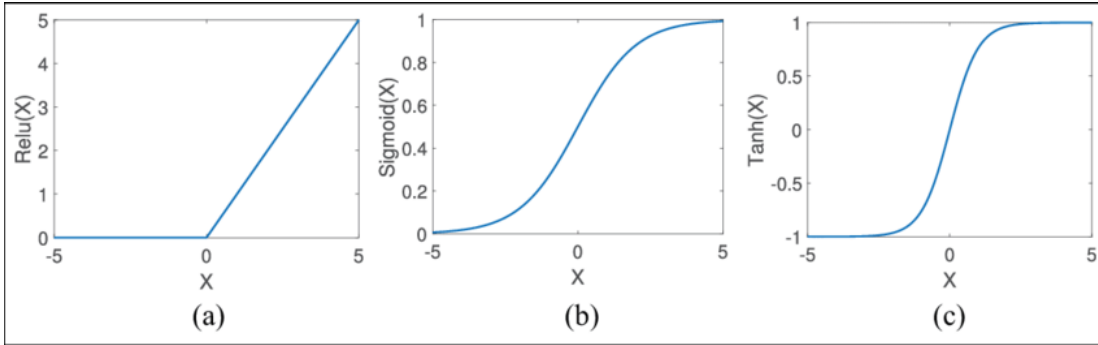


Figure 1.2: $\text{ReLU}(x)$, $\text{Sigmoid}(x)$ and $\text{Tanh}(x)$ activation functions [11]

Inference and training

The idea of NN is inspired by the human brain. Depending on the experiments a human has been through, its brain reinforces some neural connections to be able to handle similar experiment if needed. Neural networks work the same way. By providing examples to the network it is able to update and fine-tune its weights to achieve the task it has been designed for. This is called the *training process*. During training the network predictions are only used to update the weights, not to be directly used. This process is further explained in Section 1.1.2.

When training is complete one can use the network for the task it has been designed for, in a process called *inference*. This designates the action of giving inputs to the network and using the output for real-life applications.

Multi-Layer Perceptron (MLP)

One neuron is not enough to learn complex relations and create interesting applications. This is where the *network dimension* comes into play. As depicted in Figure

1.1b multiple neurons can be assembled to create a *layer* (the colored boxes). These layers can then be stacked one after another to create a *network*. In Figure 1.1b all the neurons of a layer are connected to all the neurons of the next one. For this reason this kind of arrangement is called *Fully Connected Layers (FCL)*. When multiple FCLs are connected together they form a *Multi-Layer Perceptron (MLP)*. This type of network can learn complex relations between features, each feature being a $\mathbf{x}$ entry. As an example, if one has to predict the number of shares of an online news media article, the networks would be able to link several features such as the length of the title, the time at which it was published, the number of figures, etc.

1.1.1.2 Softmax

A particular activation function not covered in the last section is the *Softmax* function. The latter takes as input a vector $\mathbf{z} \in \mathbb{R}^{(I)}$ composed of the outputs of the I neurons belonging to the previous layer. Its expression is the following:

$$Softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=0}^{I-1} e^{z_j}} \quad (1.3)$$

where $0 \leq i < I$. This expression is interpreted as giving a probability distribution of its inputs.

1.1.1.3 Convolutional Neural Network (CNN)

One of the main limitations the MLP encounters is due to the way its inputs are presented. The $\mathbf{x}$ vector is indeed *flat* meaning it only has one dimension. For some types of inputs this can be harmful as position information could be relevant. This is the case for images for instance. CNNs go over this limitation by applying *convolutions*. Figure 1.3 depicts a use case where different boat types have to be discriminated. The CNN can handle this by assigning probabilities to each boat *class*.

By alternating between *Convolution* and *Max pooling* layers one can create a *feature extractor* (Figure 1.3). This last one feeds its outputs into a fully-connected layer, responsible for the classification. In this kind of network there is no need to specify

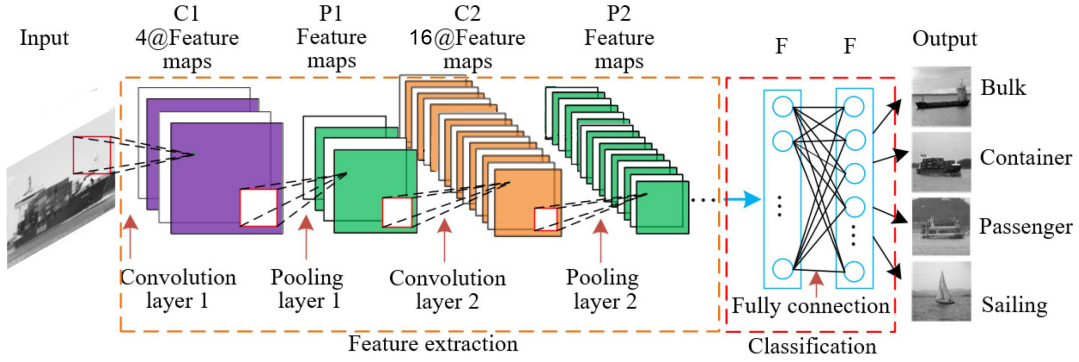


Figure 1.3: Convolutional Neural Network (CNN) global structure

features in advance as it was the case for MLP, the input image can be directly fed as is.

Convolution

The convolution is a mathematical operation expressed in equation 1.4 and visually represented in Figure 1.4 for a 2D case:

$$y[i, j] = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} k[m, n] \cdot x[i - m, j - n] \quad (1.4)$$

$0 \leq i < I, 0 \leq j < J, 0 \leq m < M, 0 \leq n < N$. Where $\mathbf{K} \in \mathbb{R}^{(M \times N)}$ is termed *kernel*, $\mathbf{X}$ is the 2D-input of a convolutional layer and $\mathbf{Y} \in \mathbb{R}^{(I \times J)}$ is its output.

Figure 1.4 presents three steps of the convolution operation with an input $\mathbf{X} \in \mathbb{R}^{(5 \times 5)}$ a kernel $\mathbf{K} \in \mathbb{R}^{(3 \times 3)}$ and an input $\mathbf{Y} \in \mathbb{R}^{(5 \times 5)}$. In this image, the input feature map is represented in blue, the kernel in grey and the output feature map in green. Each kernel is composed of trainable weights.

The kernel size as well as the number of different kernels in one layer are parameters that can be chosen by the network designer. In the first layer the input is simply the input image. For the next ones they are the output of the previous layer and are called *feature maps*. In Figure 1.3 the feature maps are represented in purple, green and orange. One can observe that the first convolution layer creates 4 output maps (in purple), meaning 4 different kernels were trained in this layer. Each of them performs

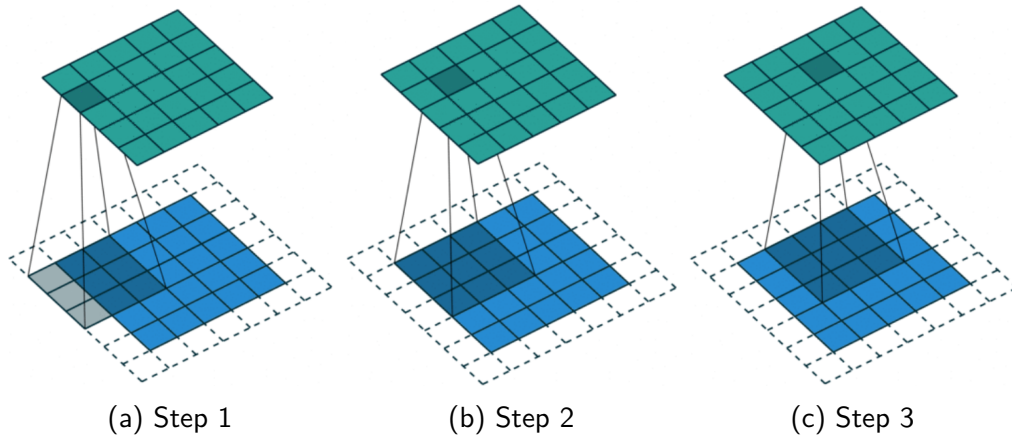


Figure 1.4: Convolution operation between a kernel and an input feature map [12]

a convolution on the layer input.

Pooling layer

Features maps output by convolutional layers are high-dimensional. This could lead to a very heavy network if the number of layers is large. To limit this phenomenon *Pooling layers* are inserted. Their role is to reduce the dimensions of each feature map by applying a *Pooling* operation. It consists in dividing the feature maps into (usually square) windows having the same dimensions and applying an operation on them. Each window is reduced into a single data by for instance retaining the maximum value of each window or computing the average value of the data. This is depicted in Figure 1.5 for a 4x4 feature map and windows of size 2x2. The pooling operation also helps the network to be more robust against small variations of input images, meaning the network can discriminate two different boats even if they are not placed on the same spot in the image [13].

Again, window size and pooling strategy are hyperparameters that can be tuned by the designer.

Softmax

At the end of the *Feature extraction* part of the network (Figure 1.3) the features maps are *flattened*, i.e. their 2D structure is reduced to one dimension by stacking

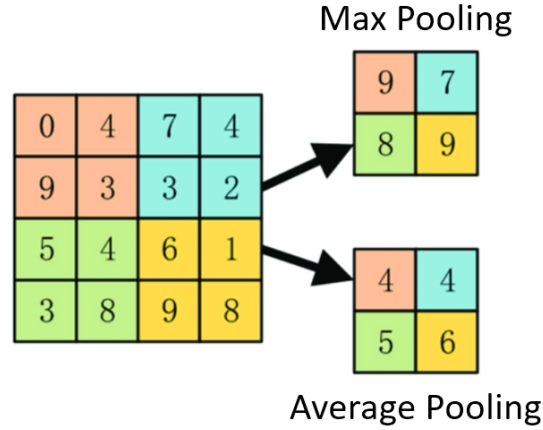


Figure 1.5: Maximum and average pooling processes

the columns. This stacked version is passed through a Fully Connected layer (see previous section) followed by a Softmax. This chain forms what is called a classifier as it assigns a greater coefficient to the most likely classes.

1.1.1.4 Quantization

When deploying to microcontrollers, two factors become of utmost importance: executable file size (a.k.a. binary size) and power consumption (latency could also be cited, depending on the application). Quantization is a mean of reducing all of these metrics by *quantizing* the weights or/and the inputs and outputs of the network, i.e. representing these numbers on less bits than their initial width. Neural networks can cope very well with this optimization [14][15].

Int-only arithmetic is the follow-up of quantization. In this paradigm, operations are performed using only *integers* instead of Floating Point (FP) numbers. The basics of these techniques and the associated challenges are explained hereunder.

Scaling

The main idea behind quantization is the *scaling*, where FP values are mapped onto integers ones using the following equation:

$$q = Q(x, b, S) = \lfloor \frac{clip(x, -\alpha, \alpha)}{S} \rfloor$$

Where:

- $\lfloor . \rfloor$ is the floor operation (that can also be another rounding mechanism as ceil or round to nearest)
- α is a clipping parameter (to remove outliers)
- $S = \frac{\alpha}{2^{b-1}-1}$ is called the scaling factor
- b is the bitwidth of the quantized numbers, i.e. the number of bits the quantized numbers will be represented on
- q is the quantized version of x

$$\text{clip}(x, a, b) = \begin{cases} a & x < a \\ b & x > b \\ x & \text{elsewhere} \end{cases} \quad a, b, x \in \mathbb{R}, a < b \quad (1.5)$$

The α parameter and $\text{clip}(x, a, b)$ function are useful to get a maximum precision in the quantized space. Indeed, if one has a weight distribution with 95% of the weights in an interval and outliers outside this last, it can be interesting to clip (i.e. set to $\pm\alpha$) the weights outside this interval. This way the size of the full range is reduced. With a smaller range to quantize, one can get a greater resolution, i.e. higher separation between close values. To assess the initial data range, multiple techniques are possible: look at only one layer, or for the whole network at once.

The method presented above is called *Uniform Symmetric quantization*. *Non uniform quantization* is also possible. In this case positive and negative values are treated separately, and the accuracy is thus not the same in the positive and negative ranges. To come back to the original representation of the number, one can multiply the quantized version by S and get $Sq \approx x$.

Softmax and non-linear operations

Unlike matrix multiplications or convolutions, other common NN operations (like LayerNorm, Softmax, etc.) are not linear operations. For linear operations, one can put the scaling factor before the expression, and retrieve the original representation by

performing dequantization. When multiple linear operations follow each other, the scaling factor is adapted after each of them, making it possible to perform $Sq \approx x$ at the end of the execution to retrieve the original value. For non-linear operations however, this is not correct anymore:

$$Linear(Sq) = S \cdot Linear(q) \text{ but } Softmax(Sq) \neq S \cdot Softmax(q)$$

The non-linear operations implementation are thus more intricate. Moreover, functions like Softmax use exponential functions, that grow so rapidly it is difficult to use with integer-only arithmetic.

The following paragraph addresses this problem with the Softmax function. Different approaches exist to go over this limitation: compute the value of a function for a certain range of data and store it in a Look-Up Table (LUT) [16], approximate them with polynomials as proposed in [15], or use modified (like base-2) exponentials. The LUT approach is fast regarding the computation, but causes an overhead on the binary size, as the LUT has to be stored on the device.

Rescaling

When using integer-only arithmetic, one has to be careful with the operations it uses to avoid *overflow* or *saturation* mechanisms. These problems arise when the range of the used numbers are exceeded. For example, an 8-bit signed integer can store integer values between -128 and 127. If one wants to multiply two 8-bits integers, it first has to accumulate the result on a 16-bit integer before scaling it back down to int8 by discarding the 8 Least Significant Bit (LSB). This gives rise to an accuracy loss, but allows to go on with int8 computations.

1.1.2 Training a neural network

The first step to go through for neural networks deployment is the training. In this step, one has to provide inputs to fine tune the weights of the network.

1.1.2.1 Supervised learning

Supervised learning is a learning paradigm that happens in three steps:

1. The forward pass where inputs are provided to the network, which makes a prediction.
2. Using a loss function, one can compute the difference between the network output and the ground truth.
3. The network weights are updated using the loss function value as well as a backpropagation algorithm (equation 1.7).

The entire flow can thus be seen as an optimization problem, where the loss function has to be optimized with respect to the weights. To find the minimum of a differentiable function, one has to compute its gradient, i.e. its derivative w.r.t each of its variables. The weights are thus updated as follows [17]:

$$\Delta w_j^t = l_r \cdot \frac{\partial L(\mathbf{w}^t)}{\partial w_j^t} \quad (1.6)$$

$$w_j^{t+1} = w_j^t - \Delta w_j^t \quad (1.7)$$

where w_j denotes a weight, l_r the *learning rate* and L the loss function. Δw_j^t is termed *delta* and t indicates the time step. The minus sign in Equation 1.7 is there to update the weight in the direction it decreases the loss, i.e. in opposite direction of the gradient, that shows the steepest *increase*. In the literature l_r is also mentioned as η . Equations 1.6 and 1.7 are known as the *backpropagation algorithm*.

Learning rate

When updating the weights, one has to choose a *learning rate*, this means a rate at which the weights will be updated. Figure 1.6 illustrates the importance of rightfully choosing this parameter. In the first case, the network will take a very long time to converge to its optimum, and the third one may not converge to the minimum of the loss function. Please note that this figure is a simplified version of a loss function, as it is in reality a function of thousands of parameters (the network weights). In practice, the learning rate will be updated through training, to perform bigger steps

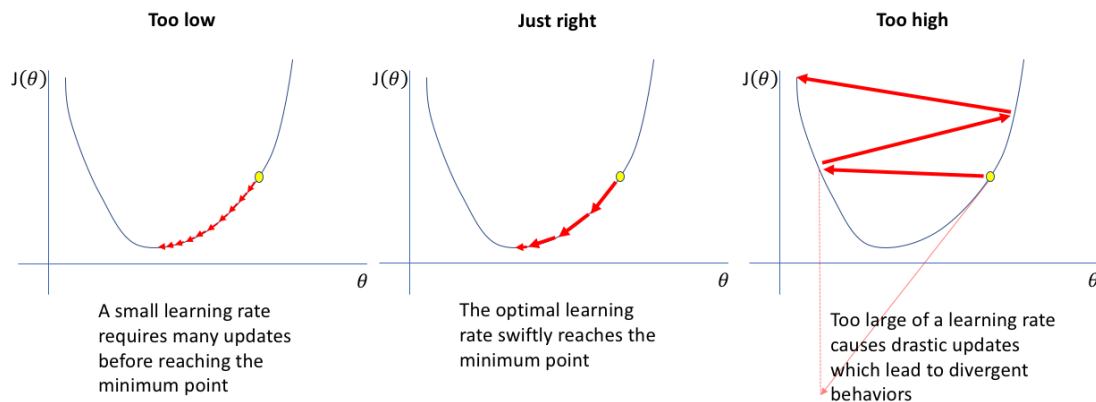


Figure 1.6: Importance of rightfully choosing the learning rate [18]

when far away from the minimum and smaller ones when approaching it as depicted in the middle schematic of Figure 1.6.

Optimizer

Equation 1.7 depicts the general idea behind weights updating during training. Many algorithms have been developed on top of this idea through the years and compiled under the name "optimizers". For instance Adam[19], one of the most common optimizers, stores a learning rate for every weight individually. Depending on the network structure and computation resources, one can choose the most suited algorithm.

Epoch and Batch

The process of training a neural network can be quite compute-intensive, as many partial derivatives have to be computed (equation 1.7 would be applied to all of the weights of a network that sometimes contains millions of them). To counteract this limitation, *batches* are often used. They are a collection of a number of data samples. When training, the loss will be computed for each element of the batch and accumulated (instead of directly modifying the weights). Once the whole batch has been processed, weights are updated according to these accumulated deltas (difference between ground truth and network output). This is described in Equation 1.8 and leads to a suboptimal step, but the computational gain gives an overall increase in training

speed. [20][21]

$$w_j^{t+1} = w_j^t - l_r \cdot \sum_k \frac{\partial L(\mathbf{w}^t)}{\partial w_j^t} \quad \forall k \in \text{batch} \quad (1.8)$$

An epoch is simply the technical term to express when all the training data has been processed. To train a network, one has to perform many epochs depending on the number of parameters and the optimizer that was used.

1.1.2.2 Dataset split

When training a neural network, the designer should carefully split its dataset in multiple parts: training, validation and test data. This separation ensures that the network will be able to generalize well to inputs it was not trained on. Indeed, if one uses all its data to train the network and does so for too long, the network could *overfit* the data, i.e. learn its training set "by heart" and not be able to correctly predict new input data.

The training set is the one used to update the network weights. It is the one discussed in the preceding sections about backpropagation, etc.

The validation set is used *during training*. It does not serve as input to train the network, but the loss function is regularly computed on this data to check if the loss function does not increase after some epochs. The overfitting phenomenon is indeed often characterized by a loss function that monotonically decreases then increases again on this data split, while the loss continues to decrease on the training data.

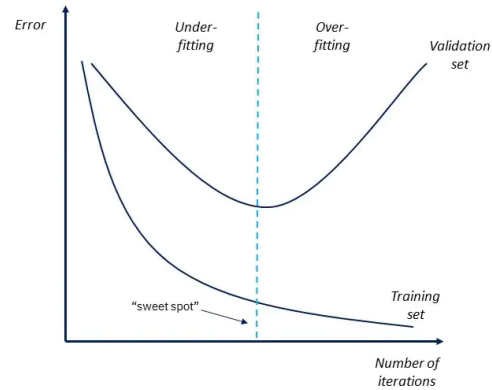


Figure 1.7: Overfitting [22]

The test set is only used at the end, to assess the accuracy of the trained network. This data was never seen by the network, and is thus a simulation of the task the network will have to tackle in its deployment phase. A common data split is the 70% training - 20% validation - 10% test.

1.1.2.3 Training with quantization

When one wants to use quantization with its network it needs to choose how to implement it during training. Two different paradigms are possible: *post-training* and *quantization-aware* quantization.

In the first scheme the network is fully trained using floating point numbers and quantized after the process is finished. In the second one, the model knows in advance it has to be quantized, and trains directly with quantized weights.

Post-training quantization is easier to apply but could generate a loss in accuracy on the trained network. *Quantization-aware* however can generate a more accurate network but can sometimes be difficult to perform depending on the network structure.

1.2 Embedded platform: DE10-Nano

The DE10-Nano is a development board commercialized by Terasic [8] and which main component is a Cyclone V device [23]. The Cyclone V is a chip containing a Hard Processor System (HPS) and a Field Programmable Gate Array (FPGA) section, developed hereunder. In this part of the work only the HPS section is used. In Part II both parts and the communication between them are utilized.

1.2.1 HPS

A HPS is a *computer core* that is directly engraved on the silicon chip, by opposition to a *soft core* that is implemented into reconfigurable logic (see next paragraph). In the DE10-Nano case, the HPS is composed of 2 ARM Cortex-A9 [24] clocked at 800 MHz. This processor is based on the armv7-a architecture and has Neon (SIMD), floating

point and DSP coprocessors. As our goal is to target small embedded applications, these coprocessors will sometimes be disabled in the next chapters to study the gain they provide and get closer to a less powerful system such as the Cortex-M series. The HPS is programmed using languages such as C or C++. Also, we execute our algorithms using only one core instead of two. Another constraint we impose ourselves is not to rely on any Operating System (OS) like Linux. Linux can easily be used on the DE10-nano platform, but would not be onto other smaller ones so we kept our application *baremetal*, i.e. the program instructions are directly executed by the processor.

1.2.2 FPGA

A FPGA is composed of logic elements that can be reconfigured to implement a digital circuit designed using a Hardware Description Language (HDL) such as Verilog. This part can be used to produce a proof-of-concept and verify that a circuit works as expected or as a direct deployable circuit. In our case, we used the board as the first usage. The Cyclone V we use contains 110k logic elements, as well as 6 Mb of reconfigurable memory. It also embeds 112 Digital Signal Processor (DSP) blocks that can accelerate the computation of DSP tasks (such as High Precision MACs, Finite Impulse Response (FIR) filters, Fast Fourier Transform (FFT), etc.). This part computes at a 50 MHz frequency.

1.2.3 System and communication

Now knowing the different parts that compose the system one has to take a look at a global view of the board and more particularly on how these parts interact with each other.

The different possibilities to communicate between HPS and FPGA are presented in Figure 1.8. In our work, we use the Lightweight bridge with the *Avalon Memory Mapped (MM) slaves* [25]. This Intellectual Property (IP), i.e. a hardware block designed by a vendor, allows the HPS to access elements in the FPGA fabric with a simple memory read, hence the name.

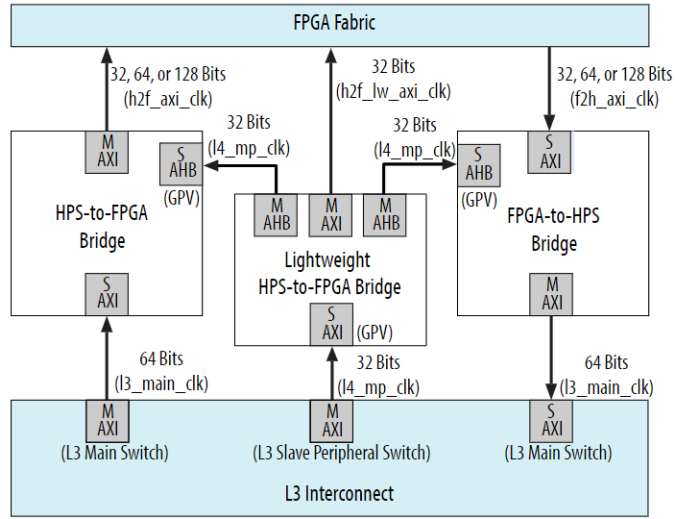


Figure 1.8: Bridges available between the HPS and FPGA

This means that if we need a memory to be implemented in the FPGA portion of the board, we can then interface it using the MM slave and read/write it from the HPS by a simple C memory read/write. Figure 1.9 depicts in details how the different subsystems and protocols interact with each other. As one can see the Cortex-A9 can reach the L3-interconnect *via* its L2 cache. The L3 will then master the Lightweight HPS-to-FPGA Bridge via its slave peripheral switch.

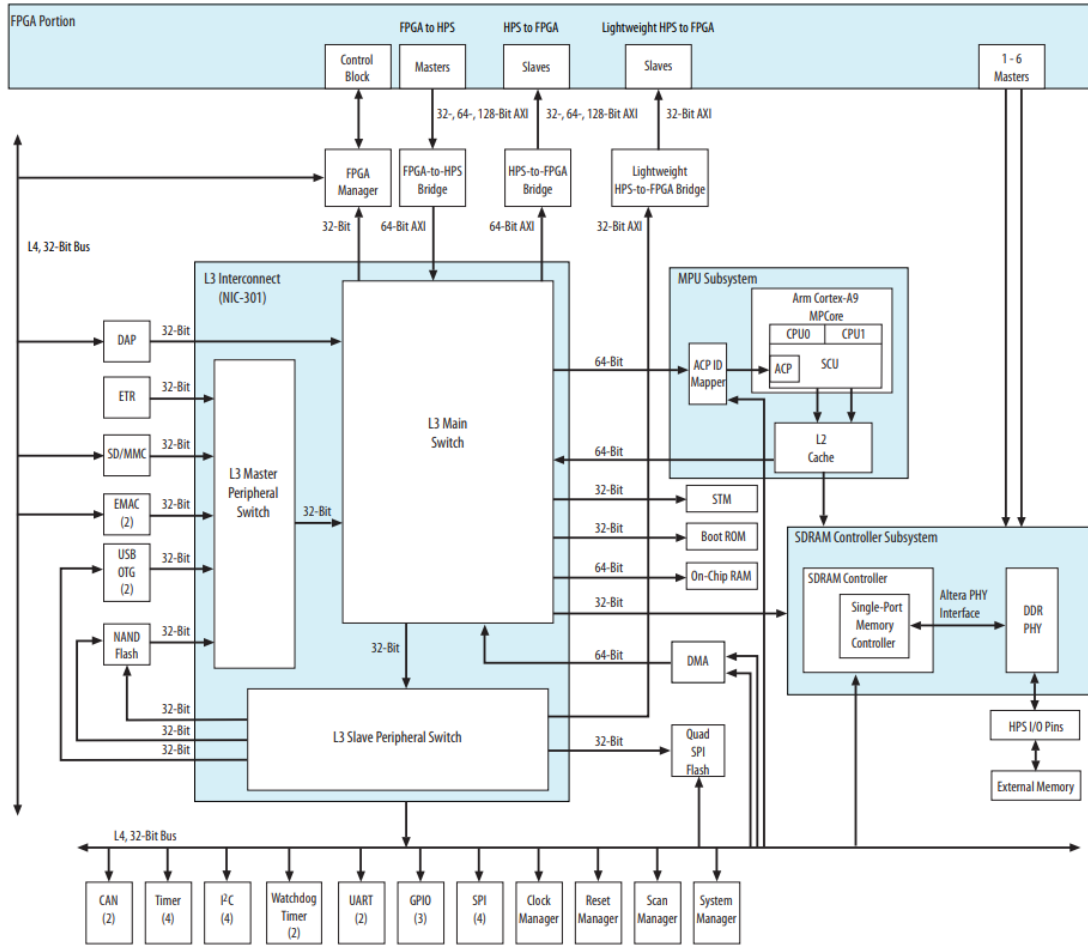


Figure 1.9: Different parts of the board and their connections

1.3 Tensorflow Lite for Microcontrollers (TFLM)[26]

As explained in Section 1.1, ML workflow happens in two steps: training and inference. The first one is often performed on computers or servers. To complete this step one can use a Python library called *Tensorflow* which is one of the many ML libraries that exist. Appendix A gives supplementary informations about this library.

The inference can be processed on any device (computer, server, embedded platform, etc.). In this work the DE10-Nano presented earlier is used for this step. To deploy a network on this kind of platform, different ways can be followed:

1. One can code the network and inference loop from scratch, recovering the weights trained earlier. For instance if the network has been coded and trained in Python, one has to translate the model structure into a programming language the board can understand.
2. Another approach is to use a library that can automatically convert the trained model into something the board can execute by itself. In this case the model translation is handled by the library. This is what Tensorflow Lite for Microcontrollers (TFLM) proposes and what is presented hereunder.

Tensorflow Lite for Microcontrollers (TFLM) is a derivative of Tensorflow Lite (TFLite), which is itself a derivative of Tensorflow (Figure 1.10). TFLite was announced in 2017 as an extension of Tensorflow, specifically designed for edge devices as phones and microcontrollers. In April 2021, the library portion was separated from the rest and called TFLM, as its usage and development targets became too different from TFLite and its targetted platforms.

This section explains the working principles of TFLite and TFLM

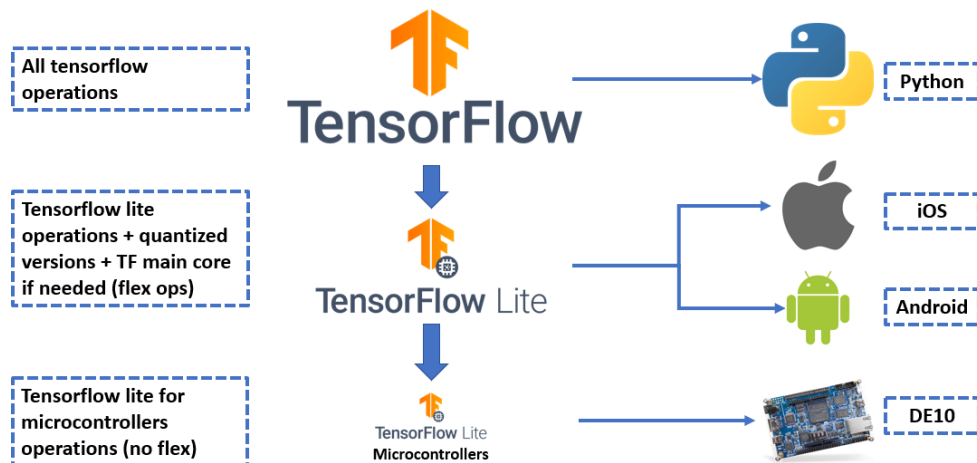


Figure 1.10: Differences between Tensorflow versions

1.3.1 Principles and promises

As explained in section A.1, a Machine Learning (ML) algorithm can be seen as a Directed Acyclic Graph (DAG). Simply put, a DAG is a graph with its nodes corre-

sponding to operations, acting on input *tensors*, producing outputs that will be used by the next node. The *Directed Acyclic* keywords come from the fact that data flows from one input to the final output. Example for such a graph is presented in Figure 2.1a. This paradigm will be used by Tensorflow to transform the Python code into a usable file format for other languages. This is the corner stone of TFLite. From this file format, one can directly deploy models on embedded platforms, using specific Application Programming Interface (API)s. For examples, it is possible to use the models on Android and Apple smartphones using [27] or [28].

Serialization

When a computer program is created, it needs to be able to exchange data with other ones. This is why a protocol must be designed to store the objects and data of that computer program. This process is called *serialization*. It takes objects and data as inputs, converts them into a binary representation (in bytes) and writes them in memory. When the same program (or another one) wants to access the data that were previously written, it deserializes it and reconstructs the object in its own memory. For this to work, both sides need to agree on the serialization protocol. Through the years, many protocols were designed with different trade offs between encode/decode time and file size metrics. The most well known are JSON and XML that create a human-readable serialized file at the cost of the mentioned metrics. Tensorflow uses *flatbuffers* presented in the next paragraph.

FlatBuffers

In many computer application (like games, network data exchange, etc.), access to data in memory needs to be fast and efficient. For example, a game could need to access tens of data structures per second, and it needs to do so while keeping a respectable framerate for the player. For this purpose, Google designed **Flatbuffer**[29], a serialization scheme that can directly access the data in memory without parsing the entire file and reconstructing all the objects [30]. This operation is thus really fast and memory-efficient and can be used to store a ML DAG. This *.tflite* file format is really convenient as the runtime in-memory representation is the same as the serialized format, and data does not need to be unpacked, i.e. objects do not have

to be reconstructed beforehand. This *tflite* file can be transformed into a C/C++ array using a python script. To encode the data, one needs to define a *schema* that describes how the data should be handled. The one used by TFLite is described at page 382 of [31].

Interpreter and project generation

The goal behind TFLM is to ease the deployment of ML models onto microcontrollers. To do so, one should be able to easily convert a classic Tensorflow model to its TFLM implementation. To achieve this, two main paradigms are possible:

1. Build a converter that creates a complete compiled binary file with the functions, layers, weights, etc. of the model, directly from the Tensorflow implementation.
2. Build an interpreter able to take any model in an understandable format and make the execution *interpreted*. This means that the model will not run directly, but will rely on another piece of code to execute its instructions.

The Tensorflow team decided to build a trade off between the two approaches that they call *project generation*. In this paradigm, all the needed source files (and these only, meaning that if some operations are not needed, they are not present in the binary) are copied into a file structure, usable in any Integrated Development Environment (IDE). To make this happen one has to run a python script provided by the library [32]. This makes the library portable, as the source files can be exported to any build system of the user's choice and thus used with a wide variety of different devices.

1.3.2 Why TFLM is different from TFLite

As stated above, TFLite was primarily designed to handle mobile platforms such as smartphones. But compared to them, some platforms such as microcontrollers lack a lot of resources. For example, many of them do not rely on an Operating System (OS), and are more constrained in memory than the mentioned devices. The Tensorflow team thus began to design a new version of TFLM that met the following criteria [31]:

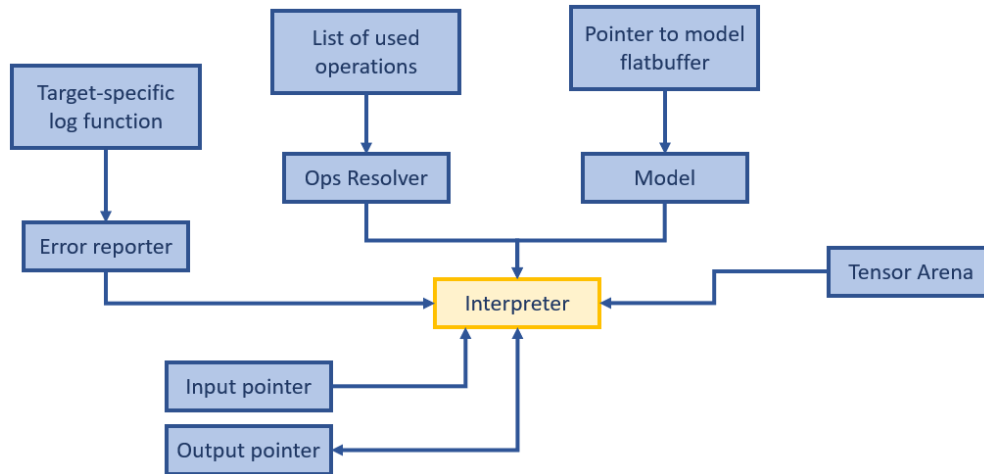


Figure 1.11: Different elements and their relations in the TFLM framework

- No OS dependency
- No dependency on C/C++ standard libraries (except for math lib)
- No Floating Point (FP) hardware expected
- No dynamic memory allocation
- Support for C++11
- A 32-bits processor

1.3.3 How does it work[31]

Now that the ideas and paradigms behind TFLM are well understood, let's take a deep dive in its inner working. The main idea is the following: An *interpreter* uses an *Error reporter*, *Tensor Arena*, input and output *tensor pointers* and an *Ops resolver* to *interpret* the *model*. All of these terms will now be explained, and their interactions are summarized in Figure 1.11.

Error reporter

All platforms are not the same, and do not contain the same I/Os. To tackle this problem, TFLM creates a generic Log function. It is up to the user to modify it depending on where the Log output should be sent. For the DE10-nano, the `printf()` function is already implemented and thus used. For other platforms, one might need to use

external ports/Leds, external screen, etc. available on his particular platform.

Tensor arena

Since the TFLM library does not expect the platforms on which it runs to have dynamic memory allocation features, it is up to the user to statically preallocate some memory where the interpreter will have room to store input, output and intermediate results. This preallocated memory is termed *Tensor arena*.

Tensor pointers

The interpreter takes a pointer pointing to the input data. As the model contains the tensor shapes, the input is simply an array that is reshaped by the model. The interpreter also takes a pointer to the output, i.e. where it will store the final tensor of the model. This pointer can then be read and further handled by the user once an inference is complete.

Ops resolver

As stated before, TFLM uses Operations (Ops) to work. This allows for a very modular design, where ops can be added, updated, and it lets the user select the ones it needs. Therefore, two choices are possible: the *AllOpsResolver*, that includes all the operations available in TFLM, or the *MicroMutableOpResolver*, where the user is free to add only the operations his network needs. When taking side for the second approach, one should be careful to include all the operations his network needs. However, this should decrease the binary size as not all operations are compiled and integrated into the deployable file.

Model

The ML model run by the interpreter is stored in the Flatbuffer file. As explained in the section related to this file format, the Flatbuffer does not need to be parsed as the memory representation is the same as the runtime representation. What is effectively passed to the interpreter is the root of the DAG.

Interpreter

The interpreter is the core element of TFLM. It takes all of what is presented above and interprets the model dataflow, using the inputs. It will read the Flatbuffer operation after operation, and access the tensors it has to operate on, until the final output.

1.3.4 From Tensorflow to TFLM model

Once the network is trained with Tensorflow it can be *converted* to TFLite and TFLM. This is where TFLM becomes very handy because one only has to run a command to transform a full Tensorflow model into a *flatbuffer* that can be interpreted on the targetted board. To achieve this conversion the converter breaks down the model into atomic operations like the ones that can be seen in Figure 2.1a. Some common layers such as Convolutional, FCLs, Max Pooling, etc. are directly implemented as operations. For other ones however, they might need to be broken down to simpler ones such as addition, matrix multiplications, etc. These operations are then called on the board depending on the deployment environment. The ops have indeed their implementation version for each deployment environment, let it be TFLite or TFLM. For instance if one deploys an iOS application the operations will be coded in Swift, for Android in Java, and for microcontrollers in C++. The conversion flowchart is illustrated in Figure 1.12.

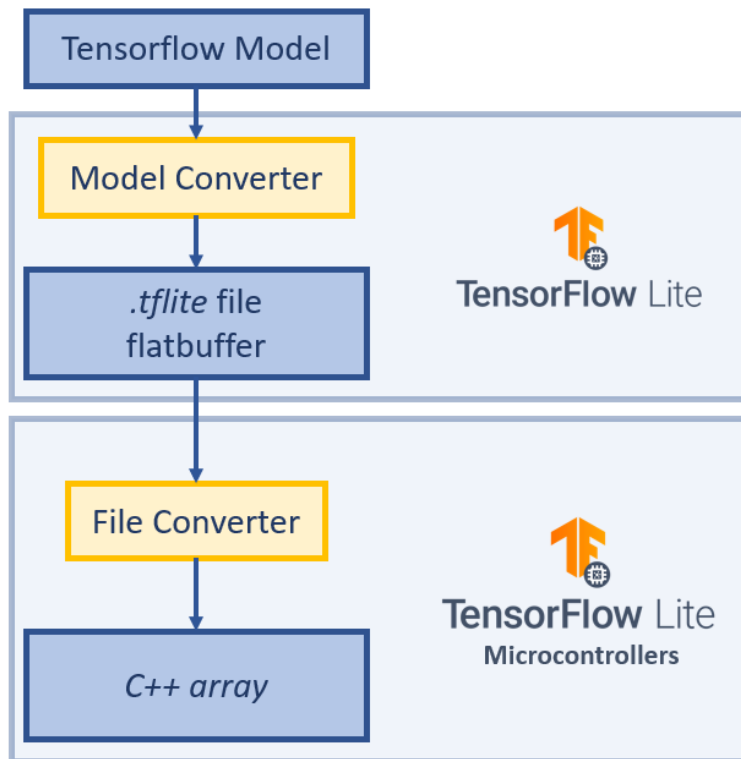


Figure 1.12: Conversion of the Tensorflow model up to the TFLM one. This happens in two steps. First the model to the TFLite version, then the *.tflite* file is converted to a C++ array.

Chapter 2

CNN deployment onto DE10-Nano using TFLM

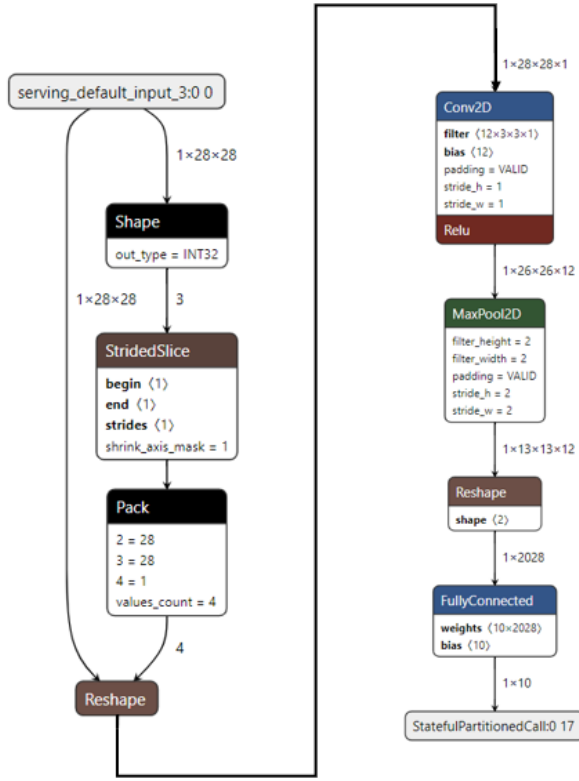
This section aims at testing the usability of TFLM on a DE10-Nano. It uses a CNN model in different configurations (quantized / not quantized) to assess the accuracy and inference timing of each of them. The presented task consists of recognizing hand-written digits (numbers from 0 to 9).

First the experimental setup is presented, describing the used model and its variations, as well as the dataset and DE10-Nano configurations. The results are then described.

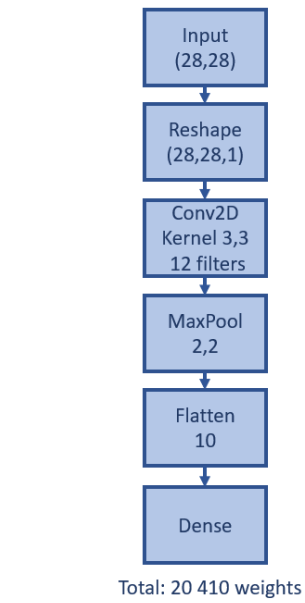
2.1 Experimental setup

Model

Parts of the code we use comes from the Tensorflow website [34]. The model is described in Figure 2.1. As one can observe it is not very deep as it is composed of only one convolutional and one Max pooling layer. The keyword *Dense* means here *Fully-Connected*. At the output of this dense layer, a Softmax is applied. The network is composed of ≈ 20000 trainable weights in total.



(a) DAG of the CNN, seen in Netron [33]



(b) Architecture of the used CNN

Figure 2.1: CNN structure

Dataset

For this example, we use the MNIST dataset [35]. It contains 70 000 images of shape (28×28) of handwritten digits (from 0 to 9) with their associated label. Figure 2.2 shows a sample taken from the dataset. This dataset is often used as benchmark for computer vision applications comparison.

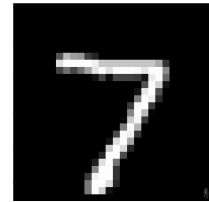


Figure 2.2: Example of mnist sample for the digit "7"

Compiler optimizations

The DE10-Nano contains the Neon floating point unit [36], capable of performing

speed up on FP operations. To stay fair with the Int8 implementation and assuming that most of battery-powered microcontrollers do not have access to this kind of technology, we divide the results between the case where this coprocessor is used and the case it is not (by forcing the compiler not to use these special instructions executing on the coprocessors).

All experiments are conducted using O3-optimization, except for the size optimized experiments, where the -Os flag is used and the coprocessor disabled for the FP implementation. On the one hand, the O3-optimization compiler flag tells the compiler to optimize the program for speed, at the cost of a greater binary size. The -Os flag on the other hand optimizes the compilation to get a binary size as small as possible, at the cost of execution speed.

Binary size

As mentioned in Section 1.3 it is possible to reduce the size of the executable file by selecting only the operations a network needs, instead of compiling the whole library. Also as explained in Section 1.1.1.4 the executable size can be reduced by the use of quantization. These optimizations will be tested in Section 2.2.2.1.

Configurations summary

Table 2.2 summarizes the different tested model configurations. The quantized versions are obtained by performing *post-training quantization*.

2.2 Results

2.2.1 Training

The training results can be observed in Figure 2.3. As one can see the network quickly converges to its final accuracy. Indeed after 4 epochs the training and validation accuracies increase at a lower rate than the previous epochs. After 8 epochs the validation loss slightly increases, depicting the typical behavior of *overfitting*. For these reasons the network that is used hereunder is trained for 8 epochs. This leads

to an accuracy of 98% on the *test set*.

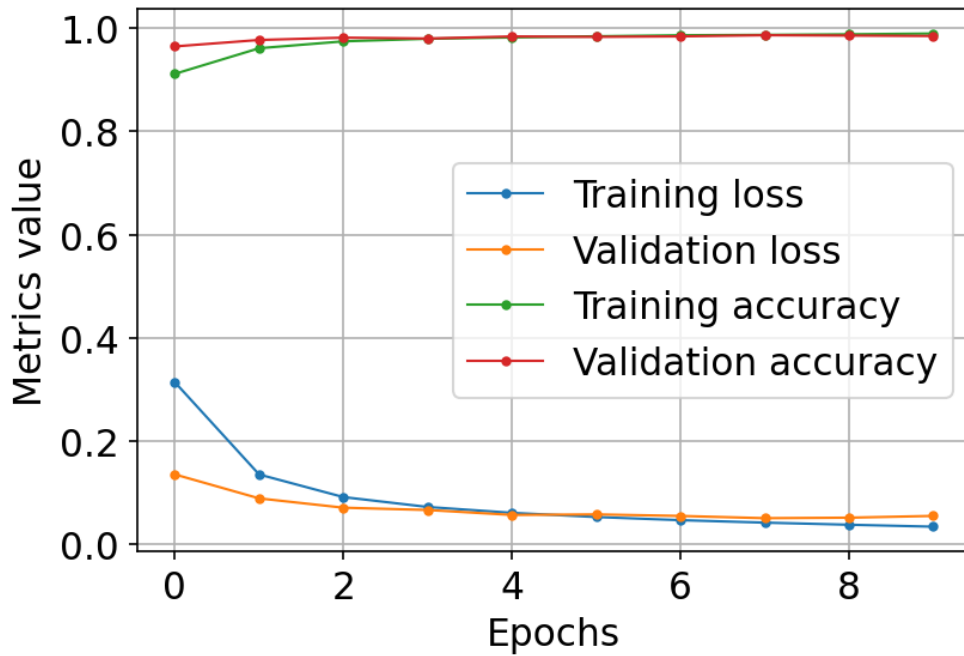


Figure 2.3: Losses and accuracies for the training of the CNN

2.2.2 Inference

2.2.2.1 Binary size

Using only needed operations

The reduction in code size obtained by selecting only the needed operations in the ops resolver is hard to observe on the DE10 due to a difficult parametrization of the compiler.

However, a quick experiment we conducted in the Arduino Integrated Development Environment (IDE) proves the advantage of using a MicroMutable resolver. Arduino is a firm commercializing microcontrollers. TFLM can be used on these boards, in a different IDE than the one we use to build for and communicate with the DE10-Nano. Results for this comparison are summarized in Table 2.1a for a 32-bits model. As one can see, using a MicroMutable op resolver reduces the binary size by 108 [kB] (i.e.

	AllOps	MicroMutable
Code size [Bytes]	333 784	224 952

(a) Code size depending on used ops resolver

	FP model	Int8 model
Size [Bytes]	414	119
Accuracy [%]	98.02	98.01

(b) Model size comparison

Table 2.1: Sizes comparison depending on ops resolver and model type

32%). In our case the MicroMutable Resolver only needs 7 operations compared to the ≈ 130 ones present in the AllOpsResolver.

Quantization

The size reduction obtained by quantizing the model into a int8 representation is presented in Table 2.1b. As expected, the size is divided by ≈ 4 (3.55). On top of this inputs can directly be stored in the quantized format. This allows a further reduction of memory allocation since inputs are now taking less space. For sensing applications, one can use an Analog-Digital Converter (ADC) of lower resolution (8 bits instead of 32). Concerning the accuracy, the difference is nearly negligible, passing from 98.02 to 98.01%.

2.2.2.2 Timing

Results of the conducted experiments are shown in Figure 2.4. The bars presented on the left side (blue) are the models running on DE10, whereas the other ones (red) are the ones evaluated on computer, in Python.

As shown in Figure 2.4, quantization does not give the expected results. For the difference between models A and B, this can easily be explained by the use of a dedicated floating point unit, that divides the execution time by a factor 3.2. But model C does not call these speedups and stays faster by 189[ms] than quantized model A. This difference with theory could be explained by the Altera toolchain that compiles efficient code for the cortex A9, even without a coprocessor. Indeed, the Cortex-A9 is based upon armv7-a architecture that contains speedups for floating point operations. On top of this this architecture is inherently based on 32 bits.

	Model	Platform	Compiler	Coprocessor	Inference type
A	int8	DE10	O3	YES	N/A
B	fp32	DE10	O3	YES	N/A
C	fp32	DE10	O3	NO	N/A
D	int8	DE10	Os	NO	N/A
E	fp32	DE10	Os	NO	N/A
F	fp32 keras	PC	N/A	N/A	Loop
G	fp32 keras	PC	N/A	N/A	Batch inference
H	fp32 tf-lite	PC	N/A	N/A	Loop
I	int8 tf-lite	PC	N/A	N/A	Loop

Table 2.2: Labelling of the different models and configurations

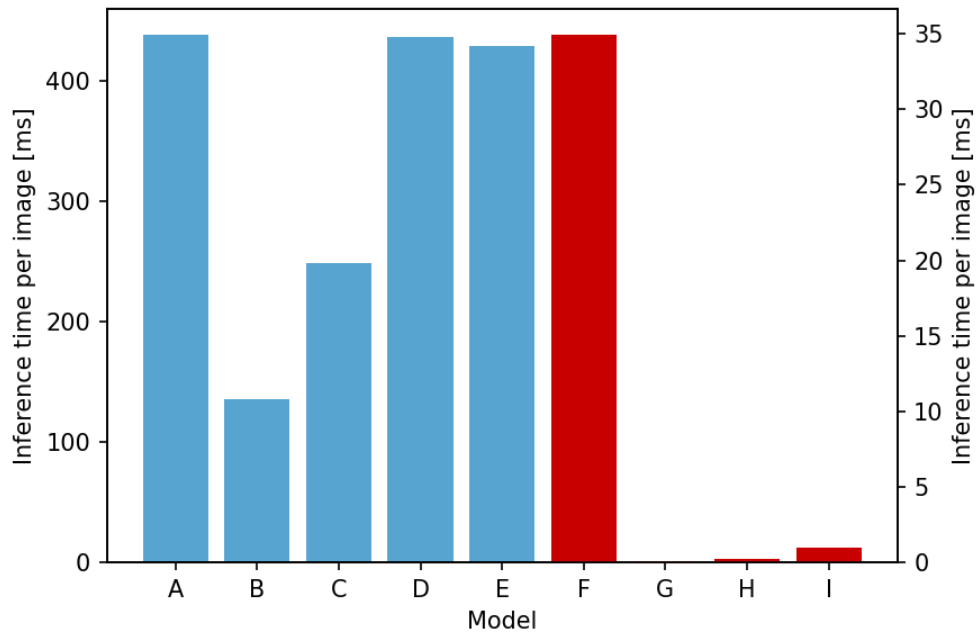


Figure 2.4: Timing of the models summarized in table 2.2. Models G,H,I are to be measured with respect to the right axis, as their execution times are orders of magnitude lower than the ones on DE10.

When optimizing the binary size, both implementations perform in the same range of numbers, with a advantage of 2% for the FP model.

When comparing these execution times with the computers, there is no match. Indeed, even the slowest model on computer (F) remains 3.88x faster than the fastest DE10

one (B). Again, we observe a faster inference onto a FP model than an int8 one. It is also noticeable how a TFLite model (H) performs better than a fully saved Keras model (F), when using a naive loop. This advantage is quickly lost when using the dedicated batch inference functions (G).

One should also keep in mind that we are here measuring the timing and sizes only, and that a crucial metric is here not presented: power consumption. The DE10-Nano board consumes a lot of power and it is difficult to assess which part of the consumption is due to the program itself and which one to the functioning of the board. Moreover, measuring current on such boards is not an easy task.

Part II

Transformers: Deployment and acceleration onto an embedded platform

Introduction

The second part of this work focuses on the training and deployment of a Transformer on the DE10-Nano.

As a reminder this architecture can be used for different tasks but contains a huge number of weights (millions to billions). This leads to an architecture with a high versatility and interesting edge applications but on which inference can be long and energy consuming.

For this reason, we focus on designing a hardware accelerator capable of speeding up the computation and reducing energy consumption on embedded devices.

We use the DE10-Nano as a prototyping board to design a SW/HW co-design using the embedded HPS and FPGA. For the software part of the design, a first attempt to map the architecture is made using TFLM. But as the Transformer has an intricate architecture, the library cannot be used to map the model.

For this reason, we choose to map only the Multi-Head Attention (MHA) using Pure C, quantization and integer-only computation. The MHA is then further accelerated by using a hardware accelerator of our design.

This part is divided into three chapters:

- Chapter 3 gives a detailed description of the Transformer architecture and its position in the Natural Language Processing (NLP) field.
- Chapter 4 draws up all the processes between the Transformer training to the

MHA software embedded deployment. At the end of this chapter a discussion is held on the implementation of different Softmax quantization algorithms. The MHA execution time is measured and compared between a full FP and an int8 implementation.

- Chapter 5 finally introduces the SW/HW co-design. It first presents the tasks distribution between the HPS and FPGA then describes the designed acceleration algorithm and the whole digital circuit. The execution time and the allocated resources of the hardware system are exposed as well as the final full system execution time, with a comparison to the software-only implementation.

Chapter 3

Transformers

This chapter aims to be a state-of-the-art of the Transformer architecture. It first addresses the different tasks that can be handled by the Transformer networks and the dominating models in the NLP field before the creation of the studied architecture. It next exposes the inner working principles of the Transformer, its bottlenecks and the existing acceleration schemes to tackle them.

3.1 Transformer

3.1.1 Natural Language Processing (NLP)

Transformers were initially designed in 2017 by the Google AI research team [1]. This architecture was designed to tackle what is called *Natural Language Processing (NLP)* problems. This field of Machine Learning tries to develop architectures to serve several tasks such as language translation, text summarizing, text classification, question answering. Some of the possible NLP applications are mentioned and explained in table 3.1. Some of them can be tested by everyone and for free on the Hugging face website [37]. As we can observe in Table 3.1, NLP tasks are used through a wide variety of applications to automate some repetitive and boring tasks.

Task name	Task description
Language translation	Translation between different languages, similar to the Google Translate service [38].
Text summarizing	Provided with a text as input, a text summarizing algorithm can produce a summary of the latter in plain english. This is useful to summarize court audiences, board meetings, etc.
Text classification	Such algorithms can group texts having a content alike to each other by assigning tags. This is useful in a company context to forward mail to only some employees based on the content or route some complaint to the appropriate customer support.
Question answering	Given a text, a trained question answering model can answer questions about it in plain English. For example, if you give the Wikipedia page of UCLouvain to such a model and then ask "Where is UCLouvain located", it can answer "In Louvain-La-Neuve".
Sentiment analysis	This kind of model is used to analyze the sentiment (happiness, anger, sadness, positive, negative, etc.) of a text. This can be used in customer reviews aggregation for example, where the main sentiment of a group of people is gathered through their reviews.
Spam detection	Sufficiently trained, these models can clean our mailboxes from fraudulent or spammy content by analyzing the sentences structures, etc.
Named Entity Recognition	This can be used to extract informations from a text such as names, places, etc. and classify them in predetermined categories.
Text generation	When reading the beginning of a sentence, this kind of network can suggest the next word(s) or sentence(s). For instance, GPT-2[39] is used as autocomplete model for Gmail.
Automatic Speech Recognition	This task is very similar to what voice assistants performs: taking a voice signal as input and transcribe it to text.

Table 3.1: Main NLP tasks and their meaning. This list is not exhaustive.

Before the Transformer

Until the publication of the transformer in 2017, the main networks used to tackle language understanding problems were the Recurrent Neural Network (RNN) and later the Long Short Term Memory (LSTM). These are briefly explained in the next

section to compare their weaknesses and strengths with the Transformer, and justify the importance of performing researches on the Transformer architecture.

RNN

A NLP task is different from other classical machine learning tasks (such as computer vision, regressions, classifications, etc.) in the sense that they have to take some context or history into account. Indeed, when a sentence is spoken by someone, words order, relations between subjects and verbs, etc. are important. For example, in the sentence "I eat an apple", the network should be able to associate the verb *eat* with the subject *I*.

Also, all the sentences are not equal in length but the network should be able to handle them all. As a counter example, it is not the case for a Convolutional neural network (CNN), mainly used in vision, where the input image size is fixed and images have to be pre-processed to meet this criterion.

These challenges led to the creation of RNNs, capable of keeping a *context vector* on top of the current input. As shown in Figure 3.1 the RNN is composed of two parts: a

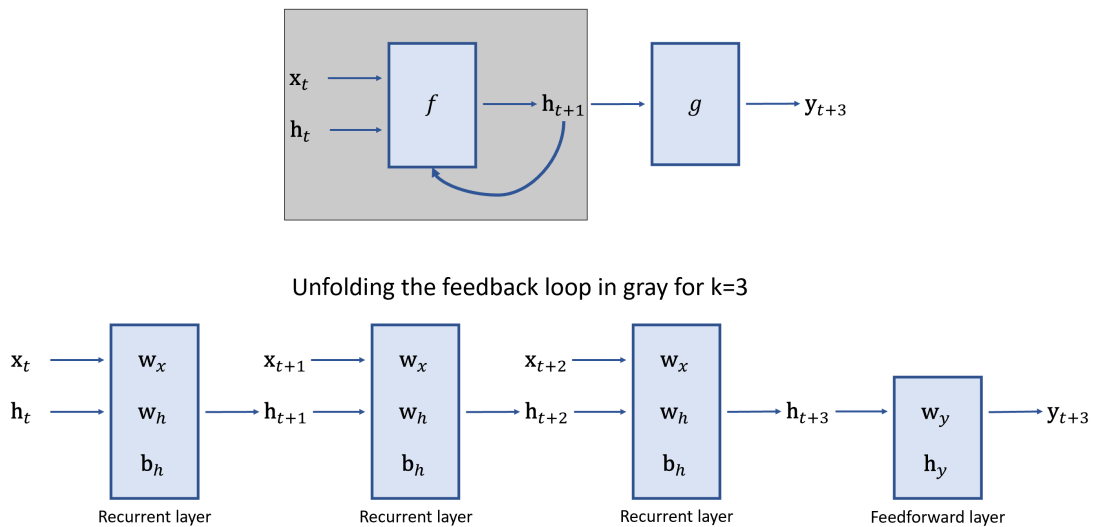


Figure 3.1: RNN inner structure

feedback loop (grey box) and a linear layer (g). The first one is a short notation for what is *unrolled* on the bottom line. As we can see, this kind of network has to store

two sets of weights, one that is applied to the $\mathbf{h}$ vector and one for the $\mathbf{x}$ vector. $\mathbf{x}$ is the input at different time steps, while $\mathbf{h}$ is the *context vector*, supposedly capturing the dependence between elements of $\mathbf{x}$ while propagating to the right. After applying the dot product between the weights and their respective inputs, they are summed up and passed through an activation function. The weights are shared across time, i.e. they are the same for all time steps. This allows to apply the network to a variable length sequence.

Given its structure the training of a RNN is very slow since it needs to first compute the output through each time step of the unrolled network, then compute the backpropagation (see Section 1.1.2) through each time step in a process called *Back Propagation Through Time (BPTT)*. This can be long if the sequence length is long itself. The network also suffers from vanishing gradient problems (where the gradient of the last time steps has very little impact on the final weight update), as well as from exploding gradients (where some weights become very large compared to the other ones). These two problems are quite common in machine learning training and have different solutions that are beyond the scope of this thesis [40]. RNNs also struggles to learn long term dependencies. If the sequences are too long, it could fail to associate words that are far apart. For example, if the sentence "*I live near the Atomium, my city is Brussels*" is given to the network which is then asked to find the word "*Brussels*" by itself, it could fail to retain the importance of the word "*Atomium*" that contains significant informations.

In order to tackle these problems some solutions were designed such as LSTM and Gated Recurrent Unit (GRU)[41], but they also suffer from several problems inherent to the structure of a recurrent network [42]. They are not further discussed in this work.

The transformer to the rescue

As seen in previous sections, RNN's main drawback is their *recurrent* nature which adds a lot to the training complexity and complicates the learning of long term dependencies. Transformer networks tackle this problem by using the *Attention mechanism*, which is explained in details in Section 3.1.2.2. The main advantages and drawbacks that

transformers brought are the following:[43]

- + No use of temporal dependence for encoding. As explained in following sections, encoding sentences are processed as a whole vector. This means that training is much easier to parallelize and is much faster since we do not need to encode the whole input sequence word by word like it is the case in RNN. This is especially true for *encoder only models* (see Section 3.1.3).
- + Use of simple and well-known operations such as activated linear layers, layer normalization and matrix multiplications. Only the positional encoding step differs from this rule (see Section 3.1.2.1).
- The network size is getting bigger with transformers, with some of the biggest models having up to 1.75 trillion (10^{12}) parameters[44].
- During batched training, the transformer cannot accept arbitrary size sequences anymore: when training the model a size is chosen and all the sequences have to be padded afterwards. This adds a preprocessing step and limits the length of the input sentences. During inference however, an arbitrary sequence length can be processed.
- + A model becoming state-of-the-art in NLP tasks for its increase in accuracy compared to LSTM and RNN [1][45]
- + As stated in this work introduction, the Transformer is a generalizable network that can be used for other tasks than language ones.

3.1.2 Architecture

The architecture of the Transformer is shown in Figure 3.2. It is composed of two main parts called the *Encoder* and the *Decoder*. These parts are composed of encoder and decoder *layers* respectively. These layers are repeated N times in series. To understand the global dataflow of this architecture, let us take an example where the network has to translate a sentence from French into English. The process is depicted in Figure 3.3.

The French sentence is fed as input of the encoder that creates a high-level representation which is then fed to the decoder. As second input, the decoder takes its

previous outputs. For example, the sentence "<sos> Les manches du manteau sont longues <eos>" is fed as input to the encoder. "<sos>" and "<eos>" respectively denote "Start Of Sequence" and "End Of Sequence". The first input of the decoder will be <sos> symbol. After having processed this symbol and the encoder output, the decoder outputs the first word: "The". Then, the next decoder inputs will be the encoder output (the same as before), and the sequence "<sos> The". This process is repeated until the "<eos>" token is predicted, giving "<sos> The sleeves of the coat are long <eos>" after 9 loops.

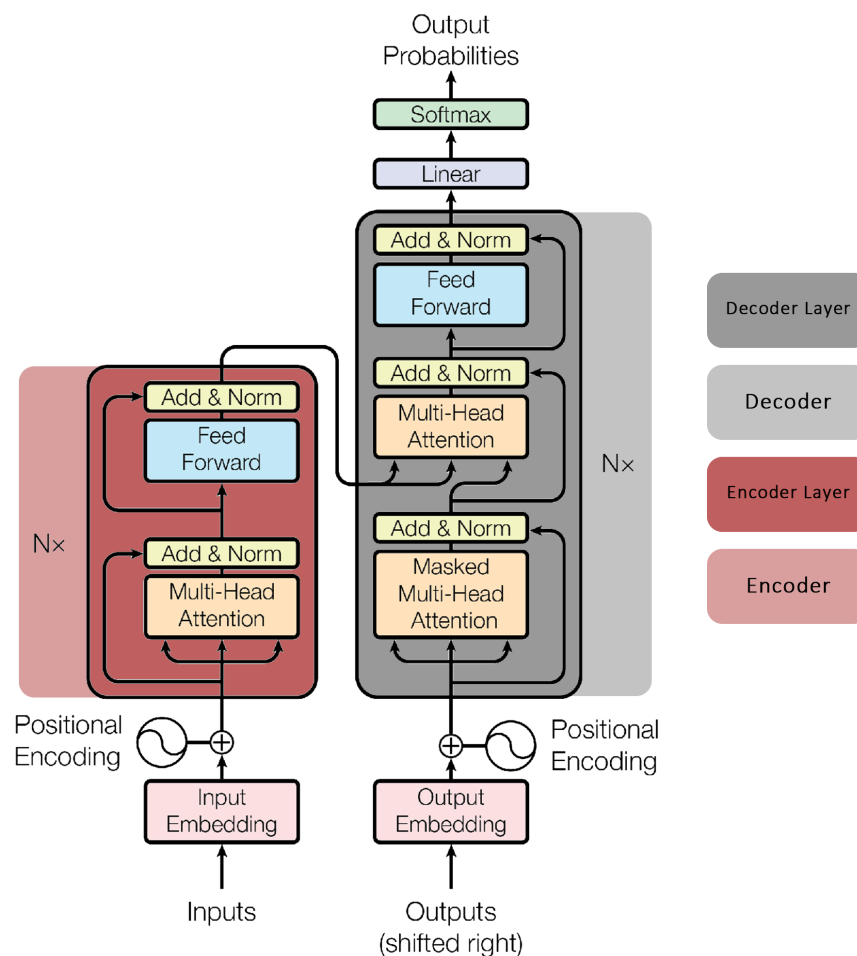


Figure 3.2: Global overview of the transformer [1]

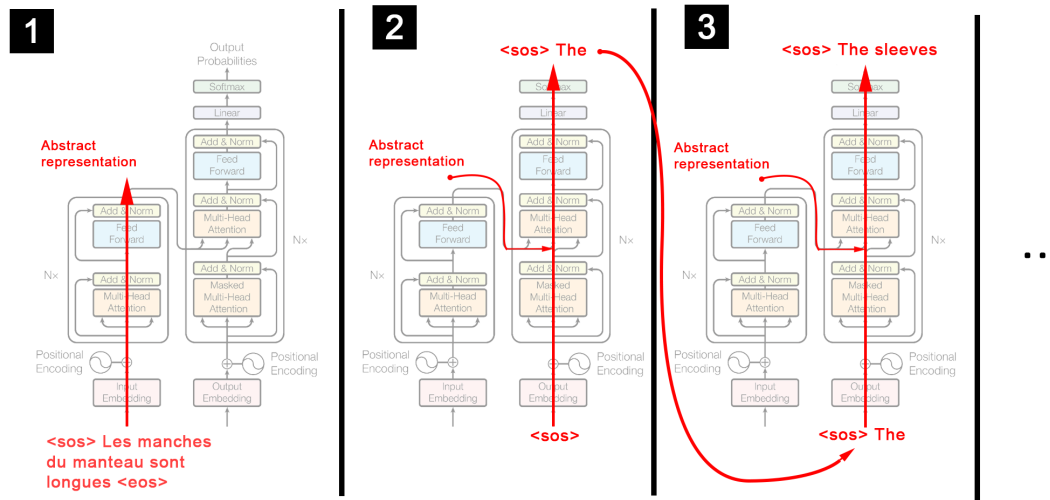


Figure 3.3: Dataflow in the Transformer architecture

3.1.2.1 Input embedding and positional encoding

Word embedding

As one can understand, French words cannot be directly fed to the neural network because computers only understand numbers. To be able to pass a word as input one has to *embed* it, i.e. convert it into a vector. The mapping between a word and its vector representation is called *embedding* and is presented hereunder. Each vector represents a word of the translated sentence. The translation between a word and its vector representation is shown in Figure 3.4.

Each input word (e.g. **Man** in Figure 3.4) is mapped through a trained dictionary into a *token*. Once the word is tokenized a trained embedding layer is used to represent the input token as a vector $\in \mathbb{N}^{d_{model}}$. The goal of this embedding is to group the words belonging to the same category (color, gender, country...). Figure 3.5 shows the idea behind the embeddings vectors. The left side of the figure shows vector representations of *Male*, *king*, *woman* and *queen* with $d_{model} = 3$. The masculine names such as *man* and *king* are represented in the same X value ($X = X_1$) whereas feminine names as *woman* and *queen* are represented in another X value ($X = X_2$). On the right side of the figure, the idea is the same with the grammatical tense of

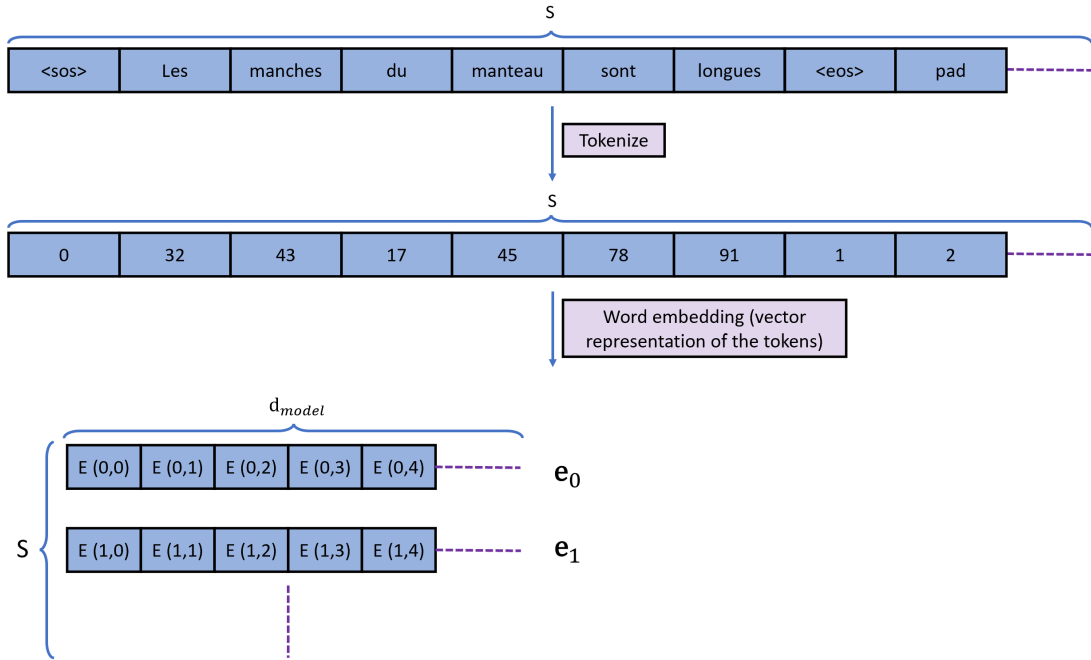


Figure 3.4: Example of Scaled Dot-Product Attention : From sentence to embeddings

a verb. In the example, verbs conjugated in the same tense stands in the same Y axis.

Sub-words embedding

Our design uses the sub-word embedding. The principle is the same as word embedding but the tokens represent sub-words of length l_{sub} instead of full words. If one takes the example of an input sequence $\langle man \rangle$ with $l_{sub}=3$, the sub-words dictionary will take into account the sub-words: $\langle ma, man, an \rangle$ where " $\langle$ " and " $\rangle$ " are special characters that have to be included to indicate the beginning and the end of an input sequence. One can train the system to create a dictionary of size L . It will retain the L most used sub-words in the training sentences and will map them into tokens. Each input sentence is represented into a sequence of S sub-words. As in word embedding, the sub-words will be represented in a vector of size d_{model} [47].

The input embedding block outputs an *Embedding Matrix* $\mathbf{EM} \in \mathbb{R}^{(S \times d_{model})}$ for each input sentence.

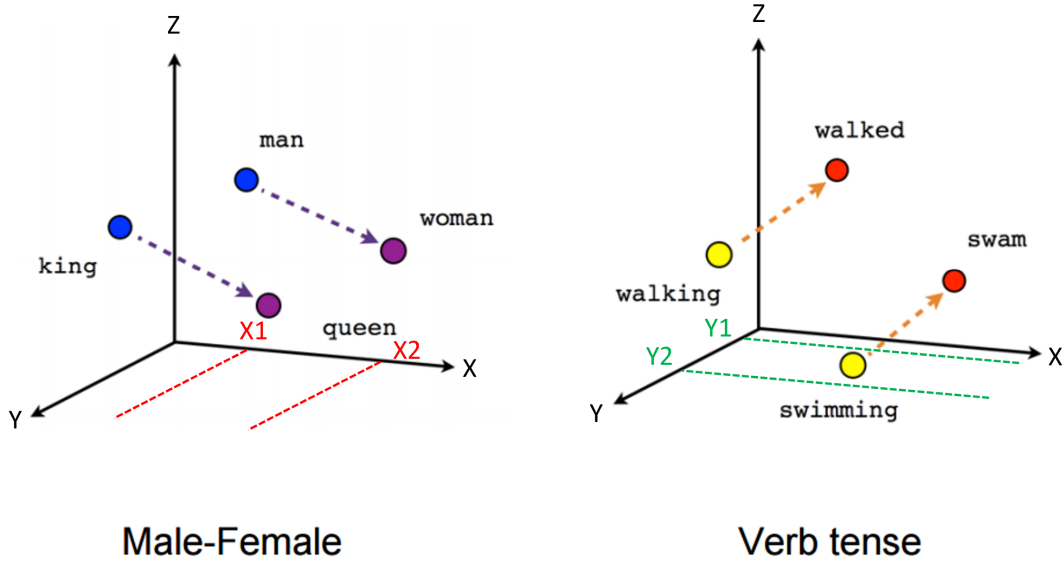


Figure 3.5: Embbeded vectors[46]

Positional encoding

When using the Transformer, positions of words are lost as everything is projected independently of their position in the sequence. Indeed, the Transformer does not use any recurrence or any convolution like other NLP algorithms (as explained in Section 3.1.2). To mitigate this and retrieve some position information, a layer called *Positional Encoding* is added. This block creates a matrix called *Positional Encoding Matrix* $\mathbf{Z} \in \mathbb{R}^{(S \times d_{model})}$ that is added to $\mathbf{EM}$ in order to bring this information about position (Equation 3.3).

The challenge is to determine what value to put in $\mathbf{Z}$. A simple idea could be to add the sub-word index to each embedded vector. However that may distort the embedding information if the sequence is long [48]. Indeed for long sequences, a position index could be so great that embedding information would be lost. What [1] proposed is to work with frequencies, and more specifically with the sine and cosine functions. The $\mathbf{Z}$ of an input sequence of length S with embedding vectors of size d_{model} is computed as follows:

$$\mathbf{Z}(pos, 2i) = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (3.1)$$

$$\mathbf{Z}(pos, 2i + 1) = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \quad (3.2)$$

Where :

- $0 \leq pos < S$ is the sub word index in the sequence.
- $0 \leq i < \frac{d_{model}}{2}$. $2i$ and $2i + 1$ are the dimension index of the embedding representation of a sub-word $\in \mathbb{R}^{d_{model}}$.
- 10000 is a design choice.

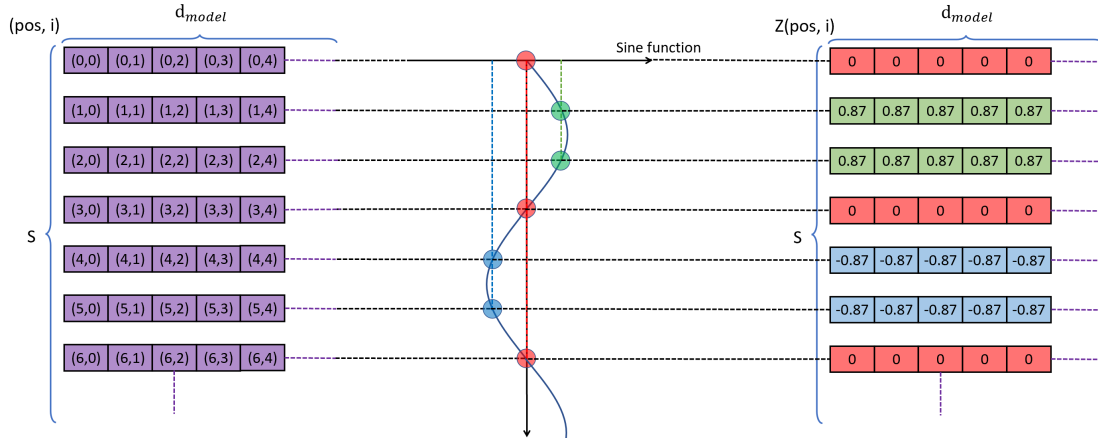
There are advantages with working with frequencies. The output $\in [-1; 1]$ and therefore avoids all the possible distortion. One problem still remains if the only parameter of the periodic functions is the sub-word index pos . As shown in Figure 3.6a, because of the periodicity, several sub-words in the sequence could get the same $\mathbf{Z}(pos, i)$. See blue, red and green clusters in Figure 3.6a. This is why the functions defined in Equations 3.1 and 3.2 also depend on the dimension i of each embedding vector $\in \mathbb{R}^{d_{model}}$. This is shown in Figure 3.6b. It allows to produce different $\mathbf{Z}(pos, i)$ for all the sub-words in the sequence. A sine function is used for the even dimensions i and a cosine function for the odd ones. Figure 3.6b shows how the dependency of i avoids to produce the same $\mathbf{Z}(pos, i)$ for different sub-words in the sequence and for different dimensions i of the sub-word embedding representation[49].

The input of the system $\mathbf{E} \in \mathbb{R}^{(S \times d_{model})}$ is the sum of the *Embedding Matrix* and the *Positional Encoding Matrix* :

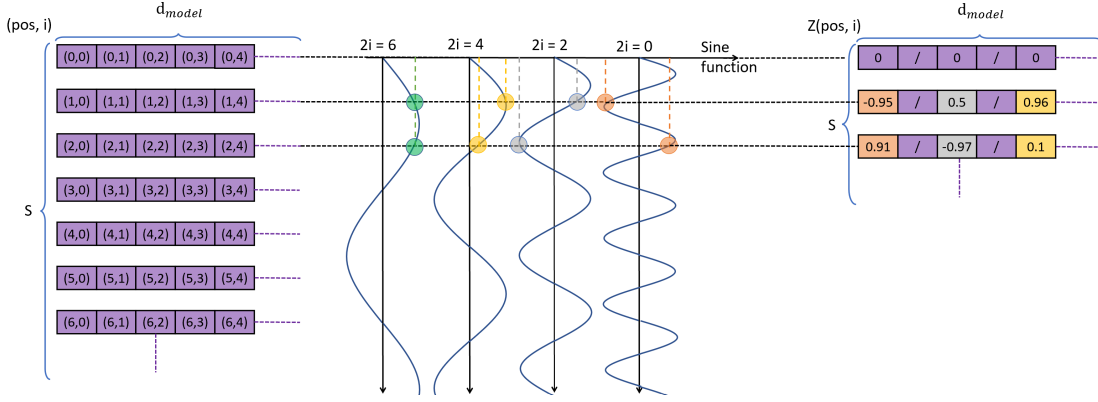
$$\mathbf{E} = \mathbf{EM} + \mathbf{Z} \quad (3.3)$$

3.1.2.2 Encoder

An *Encoder* is used to create an abstract representation of a sequence. This representation will later be used by the decoder to output the expected result. Figure



(a) P without the dependency of the embedding index i



(b) P with the dependency of the even embedding index $2i$

Figure 3.6: Positional encoding

3.2 shows the sub-layers of the encoder. A complete Encoder is composed of N_x *Encoder layers*, each of which includes a *Multi-Head Attention (MHA)* and a *Feed Forward* layer. Each layer followed by an *Add and Norm* block. All the block are explained hereunder. As explained in Section 3.1.2.1, the encoder takes the matrix $\mathbf{E}$ as input.

Scaled Dot-Product Attention

The MHA block presented in Figure 3.2 is itself composed of multiple parts shown in Figure 3.9. The main block of this sub-layer is the *scaled dot-product attention* that is developed hereafter and shown in Figure 3.7.

The Attention concept was first introduced by [50] and is used to link the sentence tokens with each other, i.e. to provide context. When creating sentences indeed, one has to keep subject, verbs, complements, etc. in the right order, use the correct words and correct conjugation in the right context.

Scaled Dot-Product Attention

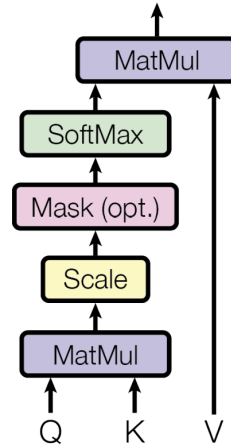


Figure 3.7: Scaled Dot-Product Attention

For instance if we want the French translation of the sentence "Les manches du manteau sont longues", the system should be able to translate "manches" by "sleeves" and not "half", since we are speaking about clothes and not football. The Attention mechanism is designed to tackle such problems.

Figure 3.8 shows the utility behind all the different layers of the scaled-dot product attention. In Figure 3.8 one wants to compute the contextualized version of the third embedded vector e_3 of the example sentence.

First, a dot product is computed between the input embedding we want to contextualize e_i (in the example the input embedding is e_3) and all the embedding vectors of the sentence e_j . The result is a score vector s_i of shape $(1 \times S)$. One has to remind that the related tokens have similar embedding representations, leading to a higher

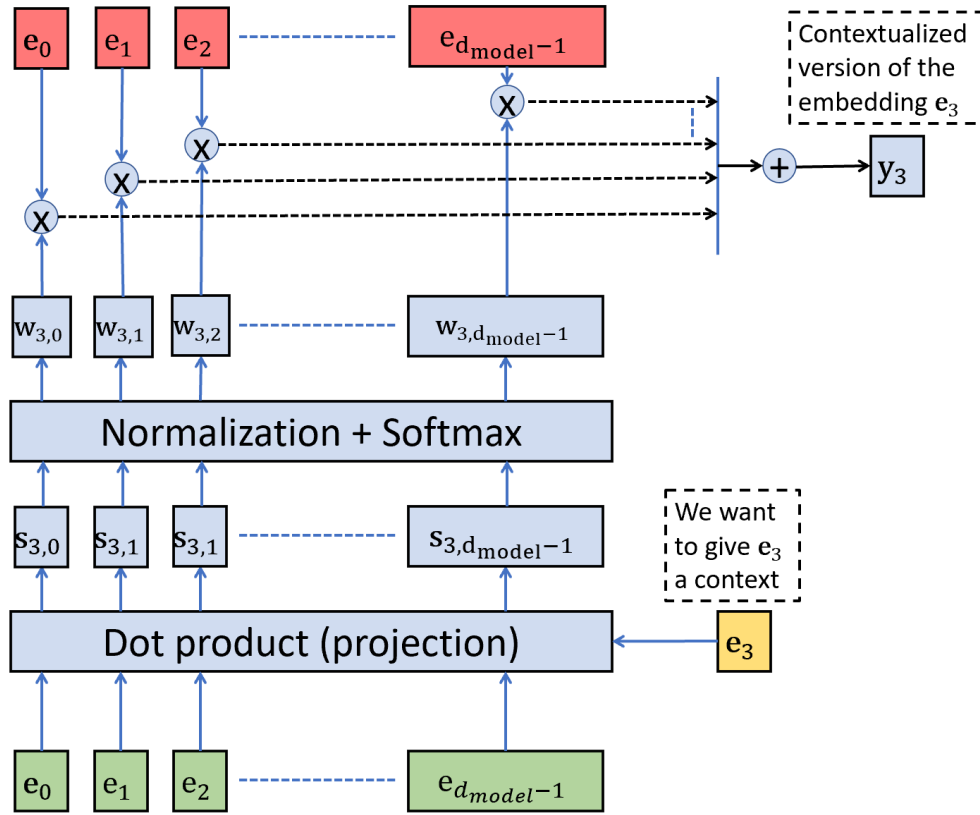


Figure 3.8: Example of Scaled Dot-Product Attention: bring context to the embedding

score s_{ij} .

$$s_{ij} = \mathbf{e}_i \cdot \mathbf{e}_j \quad (3.4)$$

Where $0 \leq i < S, 0 \leq j < S$. To obtain the attention weight vector $\mathbf{t}_i$ of shape $(1 \times d_{model})$, a scaling is first applied to the scores :

$$\mathbf{d}_i = \frac{s_i}{\sqrt{d_{model}}} \quad (3.5)$$

The scaling operation is done to avoid a vanishing gradient during training, due to large values obtained with the dot product. This is important for the following operation,

the *Softmax* :

$$\mathbf{t}_i = \text{Softmax}(\mathbf{d}_i) \quad (3.6)$$

$$t_{ij} = \frac{\exp(d_{ij})}{\sum_{z=0}^{S-1} \exp(d_{iz})} \quad (3.7)$$

Where $\mathbf{d}_i$ is the scaled score vector $\in \mathbb{R}^{(1 \times S)}$ and $\mathbf{t}_i$ is the attention weight vector $\in \mathbb{R}^{(1 \times S)}$. Attention weights are not to be confused with the trainable weights of the network. They simply are the results of Equation 3.6. This is done to obtain normalized attention weights with $\sum_{j=0}^{S-1} t_{ij} = 1$. Finally, all the obtained attention weights are applied to the embedded vectors to obtain the weighted sum $\mathbf{y}_i \in \mathbb{R}^{(1 \times d_{model})}$:

$$\mathbf{y}_i = \sum_{j=0}^{S-1} \mathbf{e}_j \cdot w_{ij} \quad (3.8)$$

The output can be interpreted as the weighted sum of all the embedded vectors of the sequence having bigger or smaller weight according to their similarity in the embedding space and their position in the sentence.

Queries, Keys, Values

In the preceding section, the same embedded vectors are used in the Attention mechanism. It is possible to bring new parameters allowing to bring a bit more of context to the embedded vectors by creating what is called the *Queries*, *Keys* and *Values*. These matrices are the results of matrix multiplications between the input matrix $\mathbf{E}$ and 3 different trained matrices $\mathbf{W}_q \in \mathbb{R}^{(d_{model} \times d_q)}$, $\mathbf{W}_k \in \mathbb{R}^{(d_{model} \times d_k)}$, $\mathbf{W}_v \in \mathbb{R}^{(d_{model} \times d_v)}$, where d_* denotes projection dimensions [51]. This projection is performed through *linear layers*:

$$\mathbf{Q}_j = \mathbf{e}_j \mathbf{W}_q \quad (3.9)$$

$$\mathbf{K}_j = \mathbf{e}_j \mathbf{W}_k \quad (3.10)$$

$$\mathbf{V}_j = \mathbf{e}_j \mathbf{W}_v \quad (3.11)$$

where $0 \leq j \leq L$ and $Q_j, K_j, V_j \in \mathbb{R}^{(1 \times d_{model})}$. The *Queries* (Q) is the input matrix we want to contextualize, the *Keys* (K) is the input matrix on which the *Queries* are projected and the *Values* (V) is the input matrix on which the weights are applied. It gives:

$$\mathbf{Q} = \mathbf{E} \mathbf{W}_q \quad (3.12)$$

$$\mathbf{K} = \mathbf{E} \mathbf{W}_k \quad (3.13)$$

$$\mathbf{V} = \mathbf{E} \mathbf{W}_v \quad (3.14)$$

This is why the inputs are labeled Q, K and V in Figure 3.7. The attention mechanism takes these 3 matrices as inputs and produces an output computed as follow:

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax\left(\frac{\mathbf{Q} \mathbf{K}^T}{\sqrt{d_{model}}}\right) \mathbf{V} \quad (3.15)$$

To ensure correct dimensions in the $\mathbf{Q} \mathbf{K}^T$ multiplication, $d_k = d_q$. For the remaining of this work, $d_k = d_q = d_v = d_{model}$.

Multi-Head Attention

The *Multi-Head Attention* (MHA) consists in parallelizing the *Scaled Dot-Product Attention* in H heads. Instead of projecting $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ at once, computation will be divided in H *Attention heads*. Each Head is responsible for multiplying a subset of columns $\mathbf{W}_q^i$, $\mathbf{W}_k^i$ and $\mathbf{W}_v^i$ of the initial projection matrices with the inputs $\mathbf{E}_1$, $\mathbf{E}_2$ and $\mathbf{E}_3$ respectively. These subsets each contain $P = \frac{d_{model}}{H}$ columns. The MHA architecture is shown in Figure 3.9 and can be expressed as follows:

$$MultiHead(\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3) = Concat(\mathbf{head}_1, \dots, \mathbf{head}_h) \mathbf{W}^O \quad (3.16)$$

$$= \mathbf{MAO} \quad (3.17)$$

Where:

- **MAO** stands for **Multi-Head Attention Output**
- $\mathbf{E}_1$, $\mathbf{E}_2$ and $\mathbf{E}_3$ are the inputs of the MHA. They are projected into $\mathbf{Q}$, $\mathbf{K}$ and

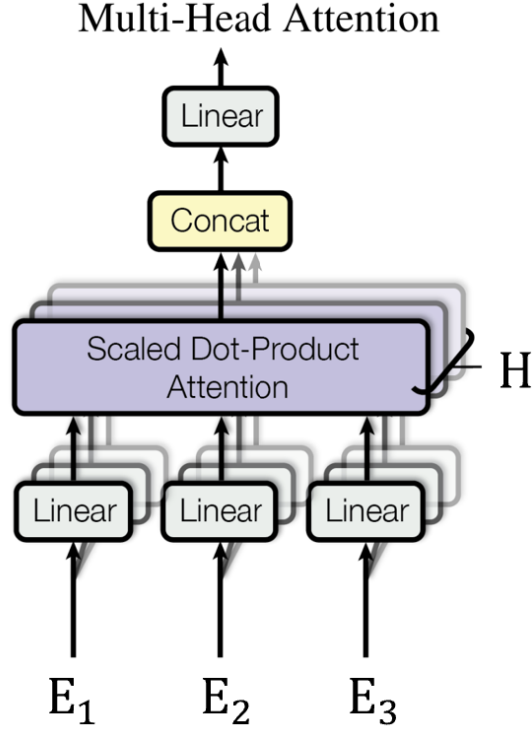


Figure 3.9: Multi-Head Attention

V matrices. In the first Encoder layer E_1 , E_2 and E_3 are equal to E . In the following ones, the inputs are the output of the preceding Encoder Layer. In the Encoder E_1 , E_2 and E_3 are always equal to each other. This is not the case for the Decoder where E_2 and E_3 come from the Encoder output and E_1 from the previous MHA block (Figure 3.2).

- $\text{head}_i = \text{Attention}(\mathbf{Q}^i, \mathbf{K}^i, \mathbf{V}^i) \in \mathbb{R}^{(S \times P)}$, $P = \frac{d_{\text{model}}}{h}$.
- $\mathbf{Q}^i, \mathbf{K}^i, \mathbf{V}^i \in \mathbb{R}^{(S \times P)}$ are respectively $E_1 W_q^i$, $E_2 W_k^i$, $E_3 W_v^i$.
- $W_q^i, W_k^i, W_v^i \in \mathbb{R}^{(d_{\text{model}} \times P)}$ are subsets of columns of the trained weight matrices $W_q, W_k, W_v \in \mathbb{R}^{(d_{\text{model}} \times d_{\text{model}})}$.
- $W^O \in \mathbb{R}^{(d_{\text{model}} \times d_{\text{model}})}$. The multiplication between the $\text{Concat}(\text{head}_1, \dots, \text{head}_h)$ and W^O represent the Linear block in Figure 3.9. This step is used to reshape the concatenation of the *Scaled Dot-Attention* from each head to obtain MAO $\in \mathbb{R}^{(S \times d_{\text{model}})}$.

Add and normalize

An addition between the output of the *Multi-Head Attention* and its input is done to avoid vanishing gradient problems during training. A normalization is applied in order to set the variance to 1 and the expected value to 0. This operation stabilizes the learning process [52][53]. The normalization of a matrix $\mathbf{M} \in \mathbb{R}^{(I \times J)}$ is defined as:

$$norm(\mathbf{M})_{i,j} = \gamma_j \cdot \frac{m_{i,j} - \mathbb{E}[\mathbf{M}]}{\sqrt{\text{Var}[\mathbf{M}] + \epsilon}} + \beta_j \quad (3.18)$$

Where $m_{i,j}$ is an element of the matrix $\mathbf{M}$, $0 \leq i < I$ and $0 \leq j < J$. The vectors γ , $\beta \in \mathbb{R}^{d_{model}}$ are trainable weights. The parameter ϵ is used for numerical stability in the case where the variance tends to 0. The *norm* layer is expressed as follows:

$$\mathbf{X} = norm(\mathbf{E}' + \mathbf{MAO}) \quad (3.19)$$

$\mathbf{E}'$ denotes the input of an Encoder Layer. For the first Encoder Layer, $\mathbf{E}'$ is equal to $\mathbf{E}$. For the following layers $\mathbf{E}'$ is the output of the previous layer.

Feed-Forward Layer

A 2-layer fully connected Feed-Forward Network (FFN) is added to add non-linearity and reshape the encoder layer output in case $d_k(=d_q)$ and d_v are not the same. The block can be expressed as:

$$FFN(\mathbf{X}) = RELU(\mathbf{XW}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \quad (3.20)$$

$$= \mathbf{FF} \quad (3.21)$$

Where :

- $\mathbf{W}_1 \in \mathbb{R}^{(d_{model} \times d_{ff})}$ and $\mathbf{W}_2 \in \mathbb{R}^{(d_{ff} \times d_{model})}$ are a trained weight matrix
- d_{ff} is the dimension of the hidden layer.
- $\mathbf{b}_1 \in \mathbb{R}^{(d_{ff} \times 1)}$ and $\mathbf{b}_2 \in \mathbb{R}^{(d_{model} \times 1)}$ are bias vectors.
- *RELU* stands for *Rectified Linear Unit*. It is an activation function and is expressed as $f(x) = \max(0, x)$

The block outputs a matrix $\mathbf{FF} \in \mathbb{R}^{(S \times d_{model})}$.

Add and normalize

Another residual connection is inserted for the same reason stated before, and the output of the *Encoder* (EO) is expressed as follows:

$$\mathbf{EO} = \text{norm}(\mathbf{FF} + \mathbf{X}) \quad (3.22)$$

$\mathbf{EO} \in \mathbb{R}^{(S \times d_{model})}$ and is the input of the next encoder layer, except for the last one that is fed to the Decoder. The fact $\mathbf{EO}$ is the same size as the Encoder layer input allows for an arbitrary number of Layers to be stacked.

3.1.2.3 Decoder

The decoder part on the right in Figure 3.2 works on the same principle as the encoder up to some minor changes. As one can observe Attention is performed not only on the input of the decoder, but also with the output of the encoder stack. These parts are called *Masked self-attention* and *Encoder-decoder attention* respectively. In the second case, $\mathbf{E}_1$ and $\mathbf{E}_3$ come from the encoded representation in the Encoder while $\mathbf{E}_2$ comes from the Masked self Attention.

To translate a sentence (or any other sequence to sequence problem), the operations are simple: first, the input sequence is given to the encoder, that produces an encoded representation. Then, the first token of the decoded sequence, i.e. $\langle \text{sos} \rangle$ (start of sequence) is given in the decoder input. Using this token the decoder processes the data and what it received from the encoder. This outputs the first word of the sequence. Then, this first word and the $\langle \text{sos} \rangle$ token are given to the decoder input and the loop goes on, up until the $\langle \text{eos} \rangle$ (end of sentence) token is reached. This means that the data has to come back at the input of the decoder several times, which can cause an exploding training time (for each sequence of the dataset, the decoder will have to infer multiple times). To tackle this problem [1] proposed *masked self attention* as first attention mechanism in the decoder.

Masked Self-Attention

The idea is the following: one could pass the complete output (the "final true answer") to the decoder at the same time as the encoder, then compute probabilities for each word without having to wait for previous outputs. The problem with this approach is that self-attention would attend positions it normally can't: if my sentence to translate is "Le ciel est bleu" and the answer "The sky is blue", the self-attention mechanism should not be able to attend positions "is" and "blue" while computing the probability of "sky", since it is supposed to predict it later. To prevent this from happening, a mask is applied just before the Softmax operation (Figure 3.7) where the values at future positions are multiplied with a large negative number. The Softmax operation then sets them naturally to zero.

Teacher forcing

As explained in previous section, the decoder has to output words one after another at inference time. During training however, a technique called *teacher forcing* is used where the true input of the preceding words are given as input to the decoder. Indeed, let's think of a sentence to translate "The sky is blue" to "Le ciel est bleu". If the decoder outputs "sol" instead of "ciel" because it made a wrong prediction for this position, the rest of the sentence will likely be wrong as well. To avoid this the correct word is fed for every position during training, no matter what the decoder outputs. However, the fact that it made a wrong prediction is integrated to the loss function to compute backpropagation.

3.1.2.4 Final output processing

The output of the decoder stack is fed to a linear layer and finally to a Softmax function that act as a classifier. This classifier outputs probabilities for the words in the dictionnary, and the most likely one has to be chosen as next output.

3.1.3 Types of models

The Transformer network is based upon a encoder-decoder architecture. As for CNNs, different model types exist depending on the usage of one or both the compo-

nents.

- **Encoder-only models** are used when one needs to learn an abstract representation of something, then directly use this representation in a classifier. Sentence classification and question answering are good illustration [54]
- **Decoder-only models** are more common in tasks such as text completion and generation. For example, Gmail text completion is performed using the GPT model, a decoder-only transformer. [55]
- **Encoder-decoder models**, or Sequence to Sequence (seq2seq) are used in tasks where a sequence is given as input and another sequence is expected as output such as translation and text summarization [56]

Since the original article we read [1] focuses onto seq2seq models, this is the kind of model we try to train in Section 4.1.

3.1.4 Bottlenecks

The main bottleneck of the Transformer architecture is the attention mechanism because of its *quadratic computation time* $O(S \cdot d_{model}^2 + S^2 \cdot d_{model})$. The attention complexity is computed as follows: [57]

- **Q, K, V** are created with a matrix multiplication of shapes $(S \times d_{model}) \times (d_{model} \times d_{model})$ leading to a computational complexity computation of $O(S \cdot d_{model}^2)$.
- **QK^T** is computed and produces a matrix multiplication of shapes $(S \times d_{model}) \times (d_{model} \times S)$ leading to a computational complexity computation of $O(S^2 \cdot d_{model})$.
- **$(QK^T)V$** is computed and produces a matrix multiplication of shapes $(S \times S) \times (S \times d_{model})$ leading to a computational complexity computation of $O(S^2 \cdot d_{model})$ as well.

By combining these computational complexities, one obtains a computational complexity of $O(S \cdot d_{model}^2 + S^2 \cdot d_{model})$. The computational complexity dramatically

increases when the input sequence grows or when the embedded vectors are represented in a higher dimension. [58] confirms this bottleneck identification :

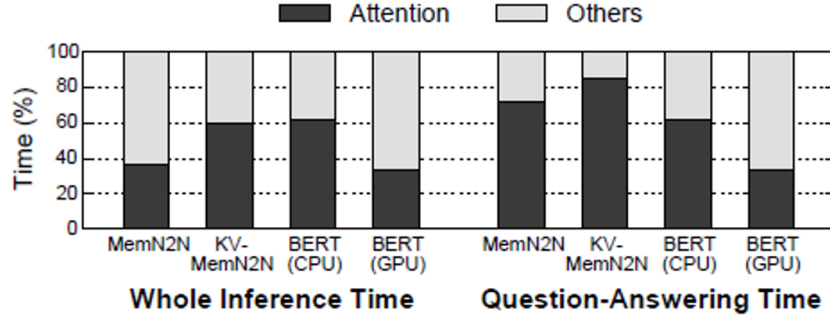


Figure 3. Portion of the time accountable for attention mechanism of NN workloads (for the total inference time and for the query response time).

Figure 3.10: Bottleneck identification [58]

An important point to note in Figure 3.10 is the time percentage taken by Attention on a CPU, which is the target of this work as very few embedded platform have a GPU.

One can also point the Feed-Forward layer as a bottleneck because of the huge matrix multiplication leading to a computational complexity of $O(S \cdot d_{model} \cdot d_{ff})$:

- $\mathbf{XW}_1$ and $RELU(\mathbf{XW}_1 + \mathbf{b}_1)\mathbf{W}_2$ are computed and produce a matrix multiplication of shapes $(S \times d_{model}) \times (d_{model} \times d_{ff})$ and $(S \times d_{ff}) \times (d_{ff} \times d_{model})$ leading to a complexity computation of $O(S \cdot d_{model} \cdot d_{ff})$.

3.1.5 Existing acceleration techniques

Once the Transformer architecture has been presented and its bottlenecks identified, it is possible to think about ways of accelerating the process. First the quantization technique is addressed as an acceleration scheme. Two algorithms are then presented and partially used in Chapter 5. Other algorithms are possible such as [58] but are not used in our accelerator and are therefore not presented. Those algorithm can also be further accelerated using dedicated hardware as presented in the second part of this

section.

3.1.5.1 Quantization

The transformer being a network with a huge number of weights, quantization is an essential technique to apply. The focus of this work is indeed to port this architecture to embedded platforms and some of them only support int8 operations [59].

[15] shows the Transformer reacts well to quantization (sometimes even outperforming the FP implementation). Table 3.2 depicts the results obtained by the I-Bert[15] team after quantizing a pretrained Transformer model called RoBERTa [60]. The table compares performances based on the General Language Understanding Evaluation (GLUE) benchmark that performs multiple language understanding tasks [61]. Table 3.3 presents the speed up that can be obtained with this technique.

(a) RoBERTa-Base												
	Precision	Int-only	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg.
Baseline	FP32	✗	87.8	87.4	90.4	92.8	94.6	61.2	91.1	90.9	78.0	86.0
I-BERT	INT8	✓	87.5	87.4	90.2	92.8	95.2	62.5	90.8	91.1	79.4	86.3
Diff			-0.3	0.0	-0.2	0.0	+0.6	+1.3	-0.3	+0.2	+1.4	+0.3

(b) RoBERTa-Large												
	Precision	Int-only	MNLI-m	MNLI-mm	QQP	QNLI	SST-2	CoLA	STS-B	MRPC	RTE	Avg.
Baseline	FP32	✗	90.0	89.9	92.8	94.1	96.3	68.0	92.2	91.8	86.3	89.0
I-BERT	INT8	✓	90.4	90.3	93.0	94.5	96.4	69.0	92.2	93.0	87.0	89.5
Diff			+0.4	+0.4	+0.2	+0.4	+0.1	+1.0	0.0	+1.2	+0.7	+0.5

Table 3.2: Results using the quantized RoBERTa architecture. No accuracy loss is observed, a slight increase in accuracy is even observed for the quantized version.

SL BS	128				256				Avg.
	1	2	4	8	1	2	4	8	
Base	2.42	3.36	3.39	3.31	3.11	2.96	2.94	3.15	3.08

Table 3.3: Speedup between the quantized model of I-Bert and its FP32 implementation [15].

3.1.5.2 Algorithmic techniques

Column Balanced Block Pruning[62]

When working with large matrices it is interesting to analyze the sparsity of the matrix. Sparse matrices are mainly composed of zero's (unlike a dense matrix) and may be processed optimally. Indeed, a lot of multiplications can be ignored as their result will be null. Not computing these leads to a reduction of the computational complexity and allows to store only a part of the initial matrices. To avoid multiplications with zeros, the *pruning* technique is used. This technique consists of deleting elements of the matrix according to a criterion. To apply an efficient pruning technique (i.e. deleted elements are organized in some way) one has to design a *pruning structure*. This latter is composed of a matrix segmentation as well as a criterion that will quantify which segments to delete.

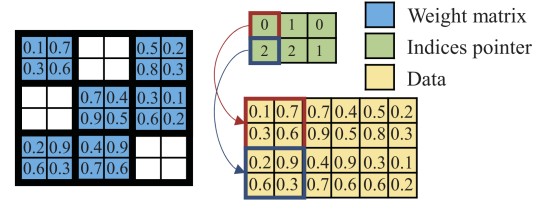
Algorithm 1: Column balanced block-wise weight pruning algorithm.

```

1 Input: weight matrix  $\mathbf{W}$ , matrix width  $n$ , matrix height  $m$ ,
  block row  $r$ , block column  $c$ , percentile  $perc$ 
2 Output: pruned weight matrix  $\mathbf{W}_p$ 
3 Set  $\mathbf{W}_p = \mathbf{W}$ 
4 Set column division  $j = n / c$ , row division  $j' = m / r$ 
5 Divide  $\mathbf{W}_p$  into  $j$  matrices:  $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_j$ 
6 foreach  $\mathbf{W}_i$  in  $\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_j$  do
7   Divide  $\mathbf{W}_i$  into  $j'$  matrices
8   Calculate  $L_2$  norm of each block
9   Setting the value in  $perc$  of the lowest  $L_2$  norm of blocks
    of  $\mathbf{W}_i$  as zero
10 end
11  $\mathbf{W}_p = \text{concatenate}(\mathbf{W}_1, \mathbf{W}_2, \dots, \mathbf{W}_j)$ 

```

(a) Algorithm of Column Balanced Block Pruning



(b) An example of CSCB format for column balanced block-wise sparse matrix, where the block size is 2×2

Figure 3.11: Column Balanced Block Pruning [62]

Column Balanced Block Pruning is one of these pruning structures. Its algorithm is shown in Figure 3.11a. The goal is to transform a weight matrix $\mathbf{W}$ of shape $m \times n$ into a pruned matrix $\mathbf{W}_p$. To do so, $\mathbf{W}_p$ is first set to $\mathbf{W}$. Then, $\mathbf{W}_p$ is divided into j sub-matrices of equal shape $(m \times \frac{n}{j})$, called column matrices. Each column matrix is divided into j' blocks of equal shape $(\frac{m}{j'} \times \frac{n}{j})$, called block matrices. For each block matrix a L_{2norm} ($\|\cdot\|_2$) is computed. Depending on the pruning ratio $perc$, the block(s) in the column matrix with the smallest L_{2norm} is (are) set to zero. The pruning ratio is the percentage of blocks that will be deleted in each column matrix.

The $L2_{norm}$ of a matrix $\mathbf{M} = m_{k,l}$ is computed as follows:

$$||\mathbf{M}||_2 = \sqrt{\sum_{kl} m_{k,l}^2} \quad (3.23)$$

The blocks with the smallest norm are the ones with the smallest influence on the final result. This is why the algorithm sets the values of these blocks to 0. These blocks aren't considered for the matrix product (Figure 3.12). Indeed, as shown in Figure 3.11b, a new matrix is created, holding indices of the non-zero blocks in each column matrix. This forms the Compressed Sparse Column Block (CSCB) matrix (in yellow in the Figure) that only contains non-zero blocks.

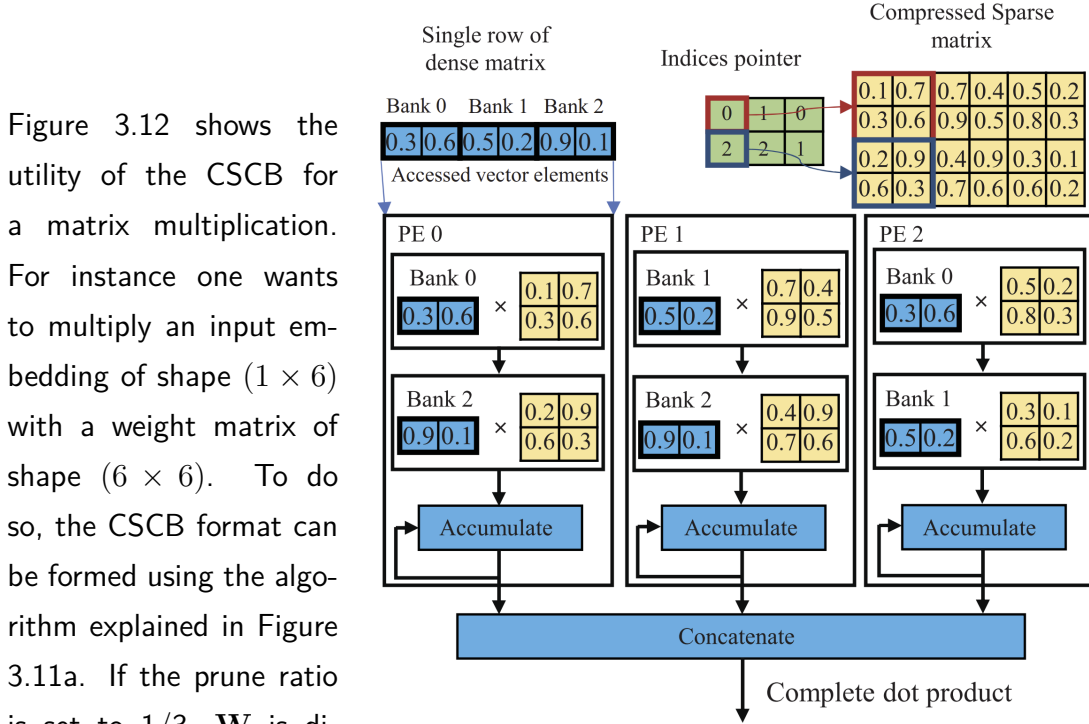


Figure 3.12: Dot product accelerator for sparse matrix with column balanced block-wise sparsity pattern

The dot product acceleration consists in multiplying the input vector with only the non zero blocks of the weight matrix. To do that, the input vector is divided into

banks of shapes $(1 \times \text{Block_rows})$, here (1×2) . The banks are multiplied only with the non-zero blocks in each Column Matrix. This is done in Processing Elements (PE) where each matrix multiplication of shape $(1 \times 2) \times (2 \times 2)$ is accumulated. At the end, the solutions of each PE are concatenated to obtain the complete dot product.

Figure 3.13

shows where the dot product acceleration can be performed. It is typically where weights matrix are stored in memory: Q, K, V matrix creation, the reshaping after the dot product attention and the linear layer.

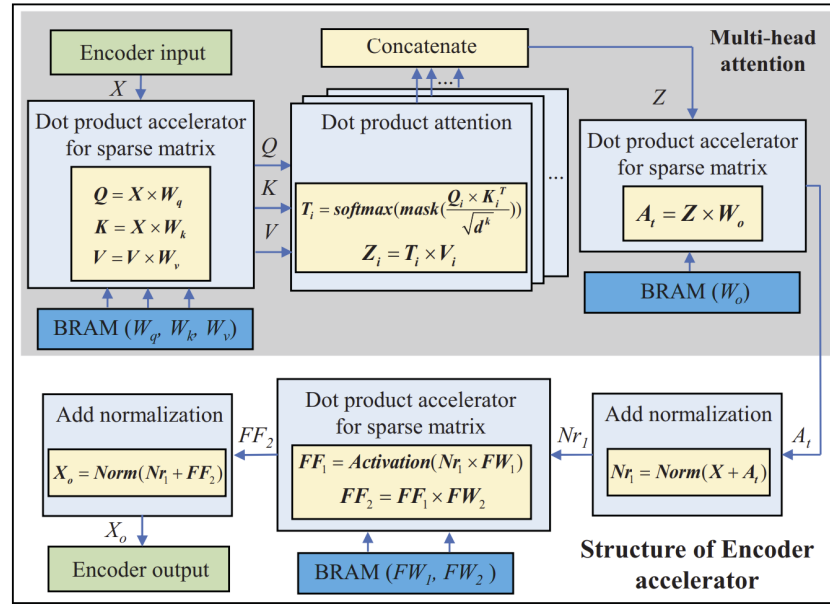
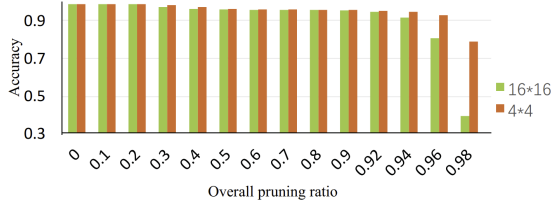


Figure 3.13: Encoder acceleration

The Column Balanced Block Pruning is pre-processed before storing the indices pointers and the Compressed Sparse matrix memory. In Figure 3.14a, one can observe the influence of the pruning ratio for block matrices of shape (4×4) and (16×16) on the next-word prediction accuracy. The results show that for a pruned ratio of 0.9, the accuracy of the Transformer model is 0.9512 for block matrices of shape (16×16) and 0.9535 for the shape (4×4) . In Figure 3.14b, a comparison of the Transformer model inference speed is done. The FPGA accelerator is 10.95 times faster than the CPU and 2.08 times faster than the GPU.

A Microcontroller is All You Need[63]

The algorithm presented in [63] serves as basis of our implementation of the Multi-



(a) Overall pruning ratio

Hardware	latency(ms)	Throughput (FPS)
Intel i5-5257U (2.7 GHZ) CPU	113.40	282.2
Jetson TX2 GPU	21.54	1485.6
Xilinx Alveo U200 FPGA board	10.35	3091.8

(b) Comparison between CPU and FPGA (prune ratio = 0.9, block size = 16 * 16 and batch size = 32)

Figure 3.14: Results of the Column Balanced Block Pruning

Head Attention (MHA). Figure 3.15 depicts the operations involved in the MHA mechanism, as well as how they are connected to each other. As one can observe many reshape and transpose operations are involved. These operations are known as *memory marshalling* and only serve to present the data in the correct order to the next operation. However they cause a performance and energy consumption overhead [64] and can be avoided with a smart design that directly outputs the shapes and data order the following layer needs. [63] proposes such a scheme, based on the shapes after reshape and transpose operations presented in figure 3.15.

As one can see, **Q** and **K** have a final shape that is different from **V**. This means that the algorithm should be different to produce the different data layout. What [63] proposes is depicted in figure 3.16a. The first part, called *Weight-Reuse linear (WRL)* is used to project the Values with a $H \times P \times S$ data layout ($H[P[S]]$ loop). In this paradigm the input is "stationnary", meaning it is updated only in the outermost loop while the weights change for every increase in the S index, which is the innermost loop.

The second one is called *Input-Reuse Linear (IRL)* and is used for the two other projections (**Q** and **V**) and outputs a $H \times S \times P$ data layout ($H[S[P]]$ loop). This is the opposite as the WRL as the weights are only updated in the H loop, and the input is changed at every S loop increment.

In both WRL and IRL cases it is possible to parallelize the computation in the outermost dimension, i.e. the "Heads" dimension. When having multiple cores at disposal, one can parallelize the computation. This means to execute the loop on different

cores for different heads. For the 2 matrices multiplications, the algorithms presented

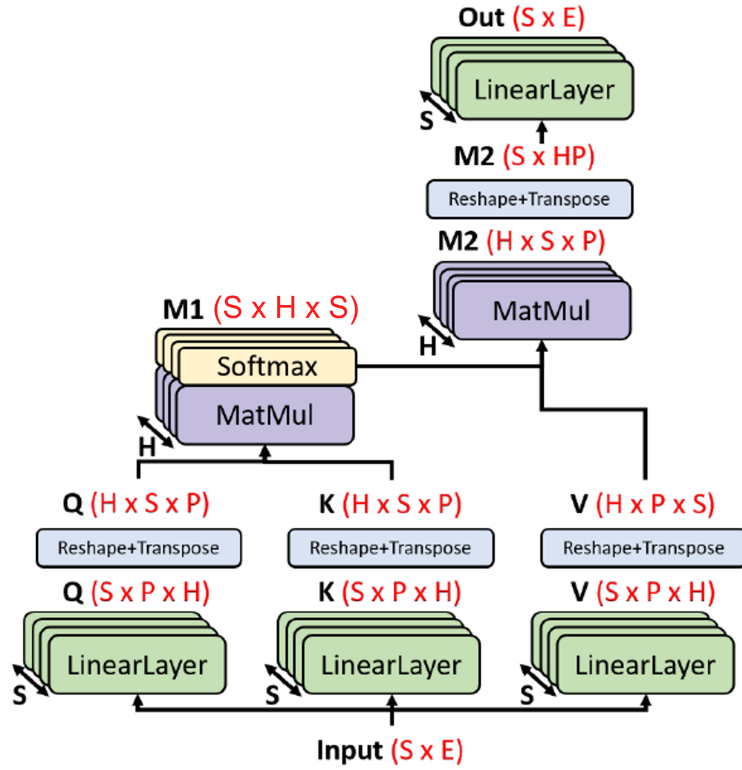
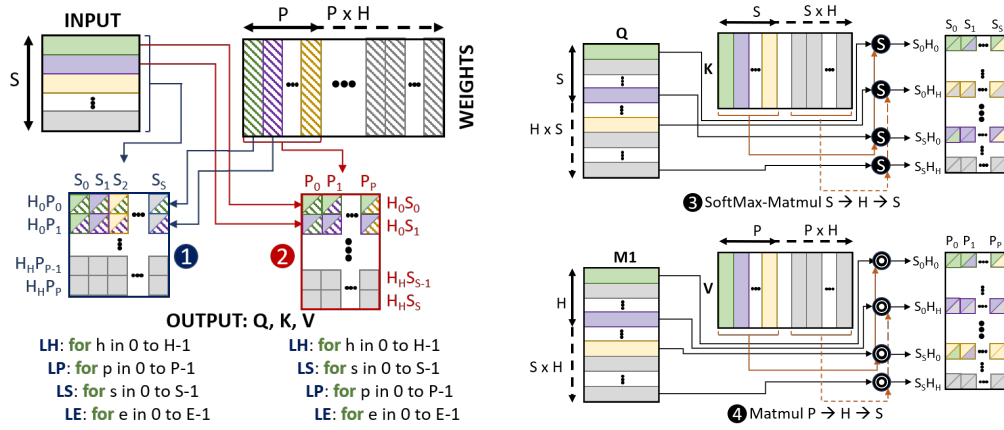


Figure 3.15: Flowchart of the multihead attention mechanism

on figure 3.16b are used. For the first case, the Softmax operation is fused within the multiplication. This means that the Softmax is applied on the output vector of the matrix-vector computation, then stored in memory. This is more efficient than processing the whole matmul and then reading the matrix line by line to apply the Softmax. This algorithm fulfills its promises by decreasing the number of cycles, time and energy needed to perform inferences. This is summarized in Table 3.4 for a Cortex-M4. This Table highlights that even using a specialized NN library like CMSIS-NN (presented in Section 4.2.2) without being careful with the reshape operations leads to worse results than what [63] proposes.



(a) Algorithms for Q, K and V projections (b) Algorithms for Softmax and Attention matrix multiplications

Figure 3.16: Data flows for the proposed algorithms avoiding data marshalling operations

Central Processing Unit (CPU) Power [mW]/Freq. [MHz]	1xCortexM4 10mW / 80 MHz		
Kernels	CMSIS-NN + Reshape	Presented work	Relative change
Cycles	103.59M	59.4M	-43%
Time/Inference [ms]	1294.92	741.93	-43%
Energy[mJ]	12.95	7.42	-43%
MACs/cycle	0.10	0.18	+80%
Throughput [GMACs/s]	0.008	0.014	+75%
En. Efficiency [GMACs/s/W]	0.81	1.41	+74%

Table 3.4: Results from [63]

3.1.5.3 Hardware techniques

Algorithmic techniques previously introduced can be further accelerated by making use of dedicated hardware like a Systolic Array (SA). This is presented hereunder.

Systolic Array[65][2]

As stated before, the Transformer main operations are matrix products. Here is an

example for $\mathbf{A} \in \mathbb{R}^{(m \times n)} \times \mathbf{B} \in \mathbb{R}^{(n \times p)}$:

$$\begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ \vdots & \ddots & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,n-1} \end{bmatrix} \cdot \begin{bmatrix} b_{0,0} & \cdots & b_{0,n-1} \\ \vdots & \ddots & \vdots \\ b_{m-1,0} & \cdots & b_{m-1,n-1} \end{bmatrix} = \begin{bmatrix} c_{0,0} & \cdots & c_{0,n-1} \\ \vdots & \ddots & \vdots \\ c_{m-1,0} & \cdots & c_{m-1,n-1} \end{bmatrix} \quad (3.24)$$

$$c_{ij} = a_{i,0}b_{0,j} + a_{i,1}b_{1,j} + \cdots + a_{i,n-1}b_{n-1,j} = \sum_{k=0}^{n-1} a_{ik}b_{kj} \quad (3.25)$$

An efficient way to compute this operation in hardware is to use a *Systolic Array* (SA). It consists of a structure of processing elements and allows parallel computations as well as accesses to local resources. The systolic array is composed of identical Processing Elements (PE) that can store, process and transfer data. It is represented in Figure 3.18a. All the PEs in the SA are linked to their direct neighbours as shown in Figure 3.17.

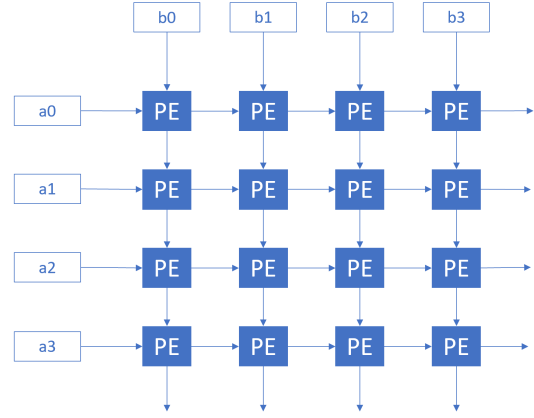


Figure 3.17: Systolic Array of shape (4×4)

As shown in Figure 3.18b, a PE is composed of two input registers R_a , R_b and a Multiply-accumulate (MAC) unit with its output accumulated in a register R_c . Its inputs in_a and in_b are transferred to its neighbors at the next clock cycle. To do so, registers out_a and out_b update their value at each clock cycle, storing the value of the inputs. The MAC unit operates the following operation :

$$c \leftarrow c + (a * b) \quad (3.26)$$

Since a request to memory is very expensive in terms of time and energy, local registers in the PEs avoid the data transfer between the main memory and the PEs. This way, inputs are transferred from PE to PE and use data stationnarity in time and space. The flow of the inputs to achieve this is shown in Table 3.5 for the following matrix

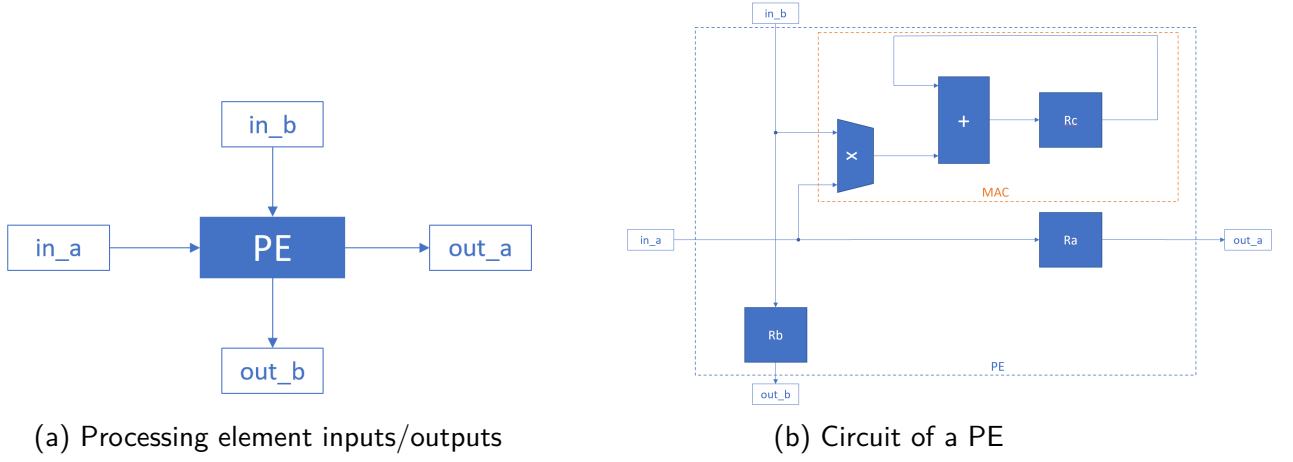


Figure 3.18: Processing Element

multiplication example:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,3} & a_{1,4} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \cdot \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,3} & b_{1,4} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = \begin{bmatrix} c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ c_{1,0} & c_{1,1} & c_{1,3} & c_{1,4} \\ c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \quad (3.27)$$

At each step, the element of the **a** vector will flow through the SA from the left side of the SA to the right side of the SA and the **b** vector from the top side to the bottom side. Since all the values of **a** must reach each column of the SA and that all the values of **b** must reach each row of the SA, the SA needs:

$$\max(r_A + c_A - 1, r_B + c_B - 1) \quad (3.28)$$

steps to finish the process. r_A, r_B, c_A and c_A denote the number of rows and columns of **A** and **B** respectively. Equation 3.28 is illustrated in Table 3.5

	Step 1	Step2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	...
a0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	0	0	0	0	0
a1	0	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	0	0	0	0
a2	0	0	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	0	0	0
a3	0	0	0	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	0	0

b0	$b_{0,0}$	$b_{1,0}$	$b_{2,0}$	$b_{3,0}$	0	0	0	0	0
b1	0	b_{01}	$b_{1,1}$	$b_{2,1}$	$b_{3,1}$	0	0	0	0
b2	0	0	$b_{0,2}$	$b_{1,2}$	$b_{2,2}$	$b_{3,2}$	0	0	0
b3	0	0	0	$b_{0,3}$	$b_{1,3}$	$b_{2,3}$	$b_{3,3}$	0	0

Table 3.5: **a** and **b** input vectors depending on the processing step of a (4×4) SA

Equation 3.25 shows what results must be accumulated in the each PE positioned at (i, j) . At the end of the computation, these results are stored and read in an output register. To be able to compute a matrix multiplication, the SA must fit with the shape of the result matrix $\mathbf{C} \in \mathbb{R}^{(m \times p)}$.

Chapter 4

From Transformer training to Software Embedded deployment

This chapter aims to address the software of the Transformer deployment onto the DE10-Nano.

First the training process of the network is described. The next section exposes the problems met when trying to deploy the Transformer using TFLM. The limitations of the library are highlighted in this section. As a reminder the dropout of TFLM led to a custom implementation of the MHA using C language. This is what is presented in the section following TFLM. The outcomes of the MHA port are finally presented. It first shows our results regarding the Softmax function quantization then deals with the timing results. These lasts are compared between a FP and an int8 implementation.

4.1 Training the transformer

The goal of this section is to create and train a prototype network that can be implemented on the embedded platform. Restrained resources (laptop CPUs or free version of Google Colab [66]) prevent the full training of a network as big as what is presented

in [1]. As a matter of comparison, this last is trained for three days with 8 GPUs in parallel. For these reasons, simplifications regarding dataset and network parameters are introduced in Sections 4.1.2 and 4.1.3.

4.1.1 Building the model

As explained in Sections A.1.1 and 3.1.2, the transformer has to be implemented either in subclassing or functional model. The second one is chosen in this work as it eases the conversion from Tensorflow to the TFLM version. It also opens the possibility to use tools such as model analyzer, that greatly simplifies model investigation by showing the sizes and the interconnections of all the layers.

The model presented hereunder is an Encoder-Decoder (see Section 3.1.3) Transformer capable of translating sentences from French to English.

As a reminder, the objective of this work is to deploy and accelerate the model onto an embedded platform and not to design it. For this reason an existing working model with a working structure and training loop can be used. The model has to respect some properties:

1. Use the functional API, as it eases the serialization and thus the conversion to TFLM.
2. Be a sequence-to-sequence model
3. Be written in a clear enough code to be understood and easily tunable

These conditions lead to the choice of [67]. This model is targeted for a chatbot application and is directly applicable for language translations. Indeed, one can adapt the used dataset and train the Transformer as is since a chatbot is also a seq2seq problem.

4.1.2 Dataset

The presented application requires a French-English translation dataset to be trained. *Europarl-v7* dataset [68] compiles proceedings from the European Parliament from

1996 to 2011. Europe being a multilingual entity, any session is transcribed and translated into the 24 official languages of the union. The French-English dataset is used as these are the languages of the main readers of this thesis. This latter is composed of 2 000 000 translated sentences.

In order to reduce training time the dataset is adapted to contain sentences with a maximum length of 150 subwords (see Section 3.1.2.1). It is also truncated to 20 000 sentences.

4.1.3 Network and Training parameters

Network

Going on with the idea of accelerating the training process, network parameters introduced in Section (3.1.2) are stripped to their minimum viable values. They are summarized in Table 4.1 and compared to the ones of [1] and [67]. Table 4.1 also gives their number of trainable weights in the Encoder and the Decoder as well as their binary sizes. As shown in [69], d_{model} is the parameter that influences the number of weights the most. Indeed, the number of trained weights in the *Encoder* can be computed as follows:

$$\begin{aligned}
 \text{Number of trainable weights} &= N \cdot \left[\underbrace{4 \cdot (d_{model} \times d_{model})}_{MHA} \right. \\
 &\quad + \underbrace{(d_{model} \times d_{ff}) + (d_{ff} \times d_{model}) + d_{ff} + d_{model}}_{FFN} \\
 &\quad \left. + 2 \cdot \underbrace{(d_{model} + d_{model})}_{norm} \right] \\
 &= N \cdot [4 \cdot d_{model}^2 + 2 \cdot (d_{model} \times d_{ff}) + 5 \cdot d_{model} + d_{ff}]
 \end{aligned} \tag{4.1}$$

And for the Decoder:

$$\begin{aligned}
\text{Number of trainable weights} &= N \cdot \left[2 \cdot \underbrace{4 \cdot (d_{model} \times d_{model})}_{MHA} \right. \\
&\quad + \underbrace{(d_{model} \times d_{ff}) + (d_{ff} \times d_{model}) + d_{ff} + d_{model}}_{FFN} \\
&\quad \left. + 3 \cdot \underbrace{(d_{model} + d_{model})}_{norm} \right] \\
&= N \cdot \left[8 \cdot d_{model}^2 + 2 \cdot (d_{model} \times d_{ff}) + 7 \cdot d_{model} + d_{ff} \right] \\
&\quad (4.2)
\end{aligned}$$

The amount of trainable weights in the *Encoder* and the *Decoder* of [1], [67] and of our architecture are given in Table 4.1.

	Vaswani and al.[1]	Model from [67]	Our Model
N	6	2	4
d_{model}	512	256	64
H (num heads)	8	8	8
d_{ff}	2048	512	512
Trainable weights (E + D)	$44.10 \cdot 10^6$	$2.63 \cdot 10^6$	$0.73 \cdot 10^6$
Binary size FP [MB] (E + D)	176.4	10.52	2.4
Binary size int8 [MB] (E + D)	44.10	2.63	0.73

Table 4.1: Parameters introduced in Section (3.1.2) used for the network where ($E + D$) stands for *Encoder + Decoder*)

Training

As explained in section 1.1.2, one has to specify a *loss function* to train the network. As mentioned in Section 3.1.2.4 the Transformer can be seen as a classifier (each subword prediction is a possible prediction class). A popular loss function in classifying problems is the *Categorical cross-entropy*. It is expressed as:

$$\mathbf{L}_{cce} = - \sum_{i=1}^{\text{output size}} y_i \cdot \log(\hat{y}_i) \quad (4.3)$$

where y_i is the truth value at index i and $\hat{y}_i$ is the network prediction for this class. In this scheme the $\mathbf{y} \in \mathbb{R}^{output\ size}$ vector is one-hot encoded meaning its value is 1 for the true answer and 0 for others (e.g. $[0,0,1,0]$).

If one has many classes to discriminate like a whole dictionary of subwords, these ground truths can be huge [70]. This is where *Sparse Categorical Crossentropy* comes in. Instead of one hot encoding, this loss function expects the truth values to be indices. Meaning that instead of $[0,0,1,0]$, $[2]$ is encoded. The equation for *Sparse categorical crossentropy* thus becomes:

$$\mathbf{L}_{scce} = -\log(\hat{y}_y) \quad (4.4)$$

where $\hat{y}_y$ is the network prediction for the true label class.

4.1.4 Simplifications and consequences

The reduction of trainable weights and the reduced dataset lead to a network that cannot learn as much as it needs to. To be able to go on with the deployment, the following idea is introduced: compare the results between the full floating-point operations performed on the computer and the quantized version performed on the DE10. To do so, one could overfit the 20 000 sentences dataset on purpose to produce a trained network from which the weights can be used in the MHA implementation.

4.1.5 Result of the training

As shown in Figure 4.1 the network clearly overfits the data (the training loss monotonically decreases and the validation loss decreases to ± 200 epochs then rises up again).

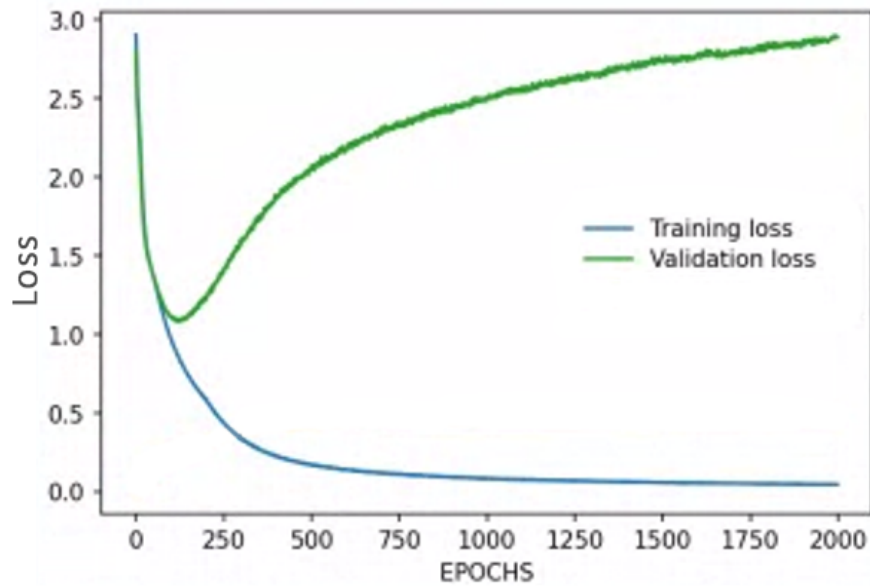


Figure 4.1: Training curve

4.2 Deploying the Transformer onto DE10-Nano

4.2.1 Transformer with TFLM: an impossible story

After being trained, the network has to be ported on the board. As discussed in Part I one can use TFLM to achieve this goal.

As mentioned above, TFLM is a fast evolving library and it is difficult to find up-to-date documentation about it. Some limitations one can encounter when using the library are presented in this section.

As a reminder of Section 1.3.4 once the model has been trained using Tensorflow it is converted to TFLM. This is done using a *converter* that breaks down the ML algorithm tree into *operations (ops)*.

4.2.1.1 Flex operations

As mentioned in Section 1.3.4 TFLite has its own operations. During the conversion process from Tensorflow to TFLite the converter attempts at generating an execution graph that contains these operations. However, it might happen that some regular

Tensorflow operations cannot be broken down to TFLite operations. In this case, TFLite can pull in regular Tensorflow ops at the cost of a much larger binary size. Those pulled in operations are called the *flex ops*. Flex operations are only possible in TFLite and not in TFLM. This means that if a model uses such operations and has to be deployed on a microcontroller, one should inspect the model and manually replace the heavy Tensorflow operations by simpler ones that can be broken down by the TFLite converter. This is not possible for all models and is thus a first limitation of the models that can be deployed using TFLM. To analyze how a model is broken down in ops, one can use the Tensorflow Model Analyzer (in experimental stage) [71]. The Transformer contains such operations but can be manually simplified to avoid *flex ops*.

4.2.1.2 Missing operations from TFLite

The second limitation comes from the separation between the lite and micro projects. It gives rise to a major downside: as explained in Section 1.3.4 TFLM uses the same *model converter* as TFLite to convert models into a *.tflite* file flatbuffer. Then a *file converter* converts the *.tflite* into a C++ array executable by the microcontroller. The problem is that not all operations of TFLite are available in the TFLM kernel. This means that even after a successful conversion to the *.tflite* version, a model could not be able to run onto an embedded platform, because its operations implementation still needs to be ported to micro.

This is what happens with the Transformer we try to implement on the DE10. The operations BATCH_MATMUL, REDUCE_PROD and SQUARED_DIFFERENCE are missing from the TFLM kernels. This is up to the TFLM community to port these operations from lite to micro [72]. Between the development of our code and writing of this text (March and July 2022), the SQUARED_DIFFERENCE operator has been ported, stressing again the fast-evolving nature of the library [73].

4.2.1.3 The end of TFLM

The lack of the C++/Make knowledge required to port these ops ourselves **forces the drop out of the Transformer deployment onto the DE10-Nano with the help of TFLM**. This is the reason why starting from Chapter 4.2.2 the workflow is directed towards the Transformer implementation in pure C. The transformer being a large architecture, only the MHA is implemented in this work.

4.2.2 MHA with CMSIS-NN: a possible story

To implement the MHA we use what is described in section 3.1.5.2 and the CMSIS library. This section aims at first presenting this library then give the results we obtain for the MHA implementation.

Common Microcontroller Software Interface Standard (CMSIS) is a library designed by ARM [74][75], providing different functions for Cortex-M and Cortex-A microprocessors. Although multiple parts exist in this library, this work mainly focuses on the use of the Neural Network (NN) and DSP ones. These parts are targeted for efficient execution of NN and DSP on Cortex-M processors by making use of a target configuration (with/without SIMD support, with/without DSP coprocessor). However, the board used in this work is equipped with a Cortex-A, as stated in section 1.2. This means that only some functions can be used from these libraries, from which we end up using only the fully connected ones. This means the functions are not particularly optimized for our current processor but would be for a deployment onto another platform using Cortex-M processors.

No other library could be found in time to implement the matrix multiplication functions, they are thus implemented by hand. However, this is done after investigating the CMSIS-DSP function structures, making it very easy to change from the custom to the library implementation. Indeed, one only has to change the function names with the ones coming from the library, as the data structures are similar between our implementation and what is done in CMSIS-DSP.

4.3 Results

As mentioned in sections and 3.1.2.2 and 3.1.5.2 one needs to implement the Softmax function to deploy the Attention layer. Therefore, MSE accuracy and timing results on the different Softmax implementations presented in Section 1.1.1.4 are first presented. Then the results of the execution time obtained for the software mapping of the MHA are depicted.

4.3.1 Quantization on Softmax

This section gives timing and MSE accuracy results of the polynomial approximation proposed in [15], the exponential implementation of CMSIS-NN, and compare them to the floating point implementation of Scipy in Python, considered as golden reference. Just as Tensorflow, Scipy is a Python library proposing ML helper functions. The LUT approach proposed by CMSIS-NN is not applicable in our case since only *tanh* and *sigmoid* activation functions are available (not the Softmax).

MSE metric is used to compare the Scipy version with the two other ones. The results are in favor of the CMSIS implementation which produces a MSE of 142 compared to the 714 of the polynomial approximation. This difference can be caused by the difficulty to choose the parameter α of the polynomial approximation that greatly influences the results. Results for the different implementations are visually depicted in Figure 4.2. It can be observed that the CMSIS-NN implementation behaves more closely to the Scipy one, particularly for a low-range input where our custom implementation does not give the expected results. This is confirmed by the MSE presented above.

Concerning the inference times, Also on this point, the CMSIS-NN implementation performs better than the other one.

	CMSIS-NN	Custom
MSE with Scipy [/]	142	714
Inference time [μs]	590	790

Table 4.2: Results for the two presented techniques to approximate Softmax function

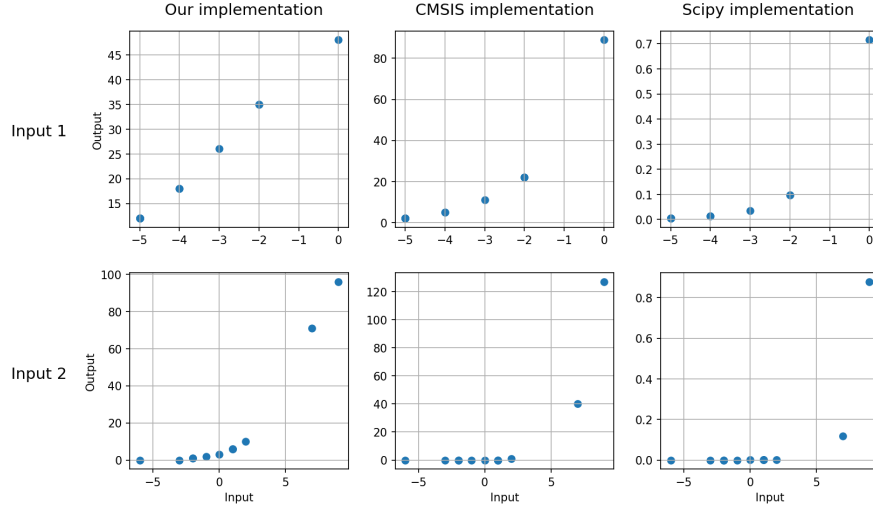


Figure 4.2: Behavior of the softmax function depending on the implementation for two different inputs.

4.3.2 Timing of MHA

We present here the results we obtain by mapping the algorithm presented in Section 3.1.5.2 on the DE10-Nano board. The algorithm is first mapped using full FP operations and without using any specialized library like CMSIS-NN (see Section 4.2.2). Then the 8-bits quantized version with CMSIS-NN is compared to this FP implementation.

Compared to what is presented in Section 3.1.5.2, weights must be transposed and reshaped beforehand in order to allow a contiguous memory access in the C code as well as to have the correctly shaped matrices. To ensure that the work is correct, output is generated using the full MHA layer from Keras (in Python) then emulated with a Python loop, and finally transcribed in C code.

Focusing on our goal of targeting small embedded platforms we use only one of the two Cortex-A9 present on the board. All the execution times are measured for one inference with a sequence of size $S = 12$. For both implementations, FP co-processors of the DE10-Nano are disabled (see Sections 1.2 and 2.1). The HPS clock frequency is set to 800 MHz.

The results for the FP configuration are presented in Figure 4.3. The total time needed to perform one inference is 121.7 ms from which 60.7 (49.9%) are dedicated to initial Q , K , V projections. As one could expected all projections take approximately the same amount of time. The second part taking most of the execution time is the matrix multiplications (34.5%) and more particularly the QK^T (Softmax-matmul) computation. This is in all probability due to the Softmax computation that must evaluate sums of exponentials. This operation can be computationally-intensive. Eventually the final dense layer takes 18.96 ms (15.58%) to project the attention output in the MHA output space. By using the DSP co-processors introduced in section 1.2 this total execution time can be lowered to 94.37 ms. To establish a fair comparison with the int8 implementation, numbers in Figure 4.3 will be used in the next discussions.

Regarding the int8 implementation, results are presented in Figure 4.4. The total execution takes a total of 24.66 ms which is a reduction of 79.73% compared to the FP mapping. The part taken by the Softmax-matmul computation is reduced, now only taking only 18.7% of the total inference time (compared to 29% for the FP). Initial projections are still taking most of the execution time (12.93 ms / 52.4%). These results confirm the advantage of using int8 operations instead of full FP.

Now that a baseline has been established it is interesting to think how the process could be accelerated. For one wishing to use multiple cores, this can be realized by parallelizing the projections. This is possible as their inputs do not depend on one another. This is not the case for the following matrix multiplications and dense layer, where previous steps have to be computed before being allowed to begin. These considerations are illustrated in Figure 4.5, the top three rows being for a platform containing three cores, then two cores and the last one for one core only.

For the first case, one could save $\approx 8.55\text{ms}$ (34.67%) of execution time.

On a more constrained device with only 2 cores, one could parallelize the computation of Keys and Queries projections, then Softmax-matmul and Values projection, as per Figure 3.15 these operations can be simultaneous (Values projection can be done later

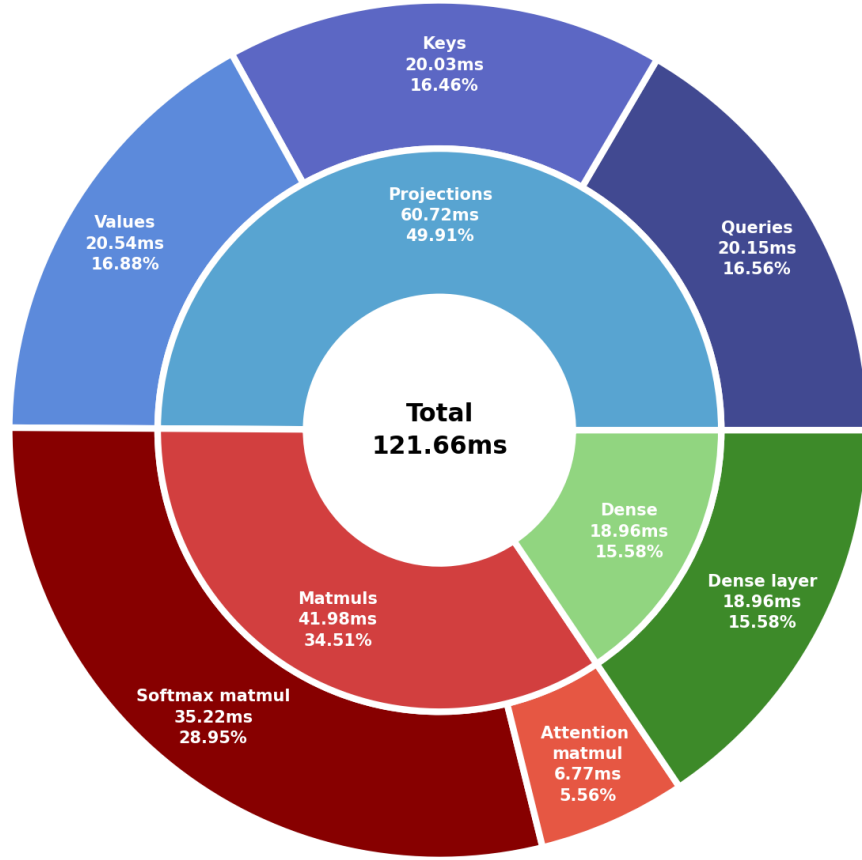


Figure 4.3: Profiling of the execution with the time taken by the different steps for a full FP implementation (no DSP co-processors) of the MHA.

than queries and keys). With this method, a total of ≈ 8.55 ms ($4.16 + 4.39$) could be saved. This is the same amount as the previous case. Note that these parallelization is efficient if the Queries and Keys can be projected in the same amount of time. Also, the Values projections has to complete before the end of the attention matmul. If these conditions are not met the computation will be slowed down because one of the core will have to wait for one of the computations to end. Results in Figure 4.5 illustrate these constraints are fulfilled for the presented algorithm. Depending on how the cores exchange data (in a shared or separated memories), computation time might differ.

As we want to stick to one core, we propose in Chapter 5 a hardware accelerator

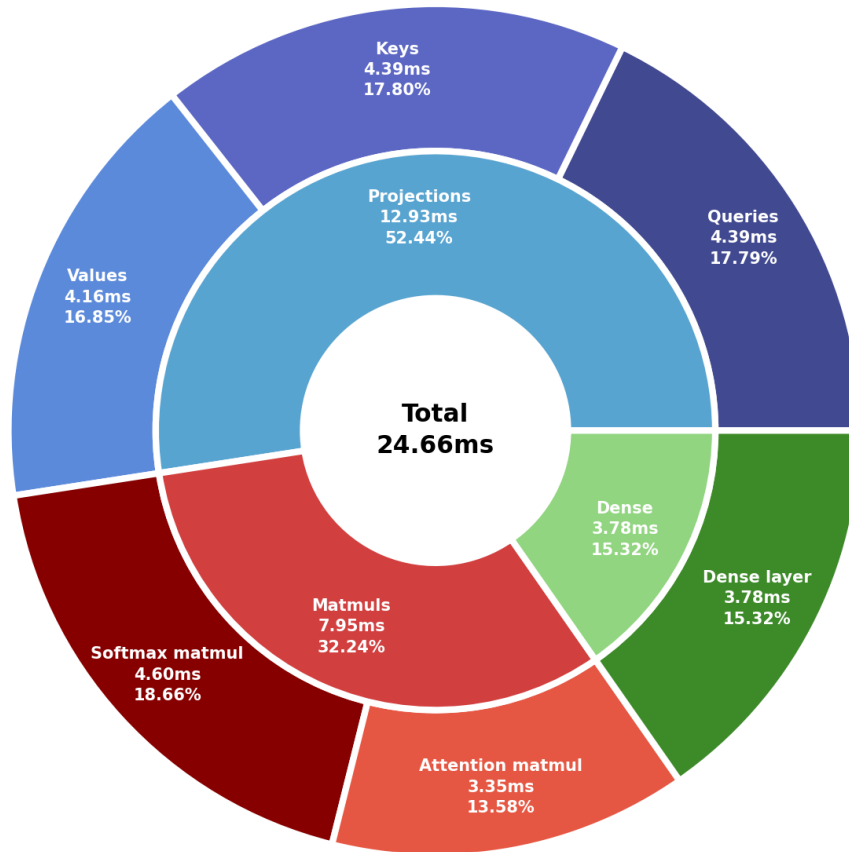


Figure 4.4: Broken down execution time for the int8 implementation of the MHA.

following these approaches, adapted to a HW-SW co-design.

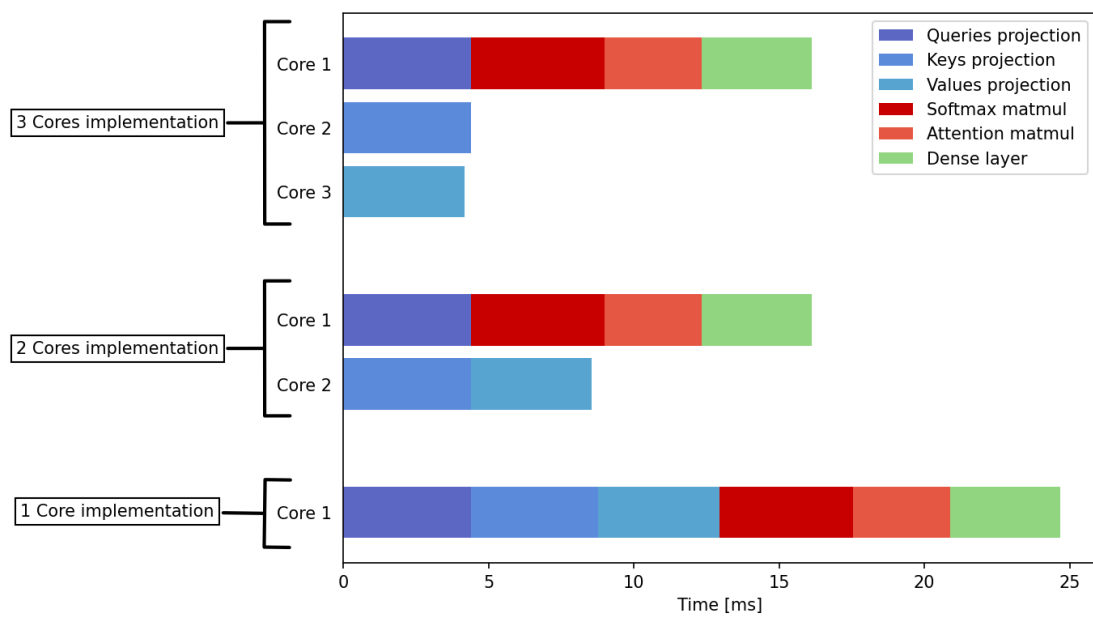


Figure 4.5: Different paradigms to map the software execution onto a multi-core platform

Chapter 5

MHA hardware acceleration

This chapter aims at presenting our hardware accelerator of the MHA. Section 4.3.2 shows that its main bottleneck is the input projections computation (i.e. $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$). To accelerate the process we use SW/HW co-design. For a reminder about the board architecture and its communications means as well as the different algorithms that can be used, see Sections 1.2 and 3.1.5.

The chapter is organized as follows:

First the co-design SW/HW is described. It consists in the tasks splitting between the Hardware and Software parts and in the full communication system. The designed acceleration algorithm is described afterwards and followed by a full description of the digital architecture implementing the presented algorithm. Focusing on the presented goal, DSP blocks (see Section 1.2) are disabled at synthesis time for our system to be as close as possible to a small embedded platform.

In the following sections, the keywords *power consumption*, *latency* and *area* are used in the Power - Performance - Area (PPA) sense. These are the three main measures commonly used to compare digital systems and are presented in Table 5.1.

The end of the chapter is dedicated to the final results of this work. It presents the area and the latency outcomes of the full accelerated system. A comparison is done

between the software implementations and the hardware-accelerated one.

Abbreviation	Word	Meaning
P	Power	The power consumption of a digital circuit
P	Performance	The performance factor is application-dependent. For instance if one is trying to compress images the performance could be expressed in pix/s the application is compressing. In the current context performance will be with regards to the time needed to complete one inference. The word latency is also used as the invert of performance. The lower the latency the higher the performance.
A	Area	Area is the metric expressing the die area needed to print a digital circuit. In the context of this work Area will be expressed in terms of Adaptive Logic Module (ALM) usage on the FPGA.

Table 5.1: Key metrics to evaluate digital designs

5.1 Hardware-Software system architecture

Software-Hardware (SW-HW) co-design means that part of our algorithm will be executed on a Central Processing Unit (CPU) and another part will be executed on a digital circuit specifically designed to implement this part. As the DE10-Nano board is used for the MHA deployment, these parts correspond to the Hard Processor System (HPS) and the Field Programmable Gate Array (FPGA) respectively (see Section 1.2).

5.1.1 Tasks splitting

When designing a HW-SW system, one has to choose how to allocate the resources at his disposal and where different parts of the code will be executed. As it can be observed in Figure 3.15 the system proposed in this work is composed of multiple parts:

1. Three linear projections (bottleneck of the MHA)

2. Matrix multiplication + softmax (Softmax matmul)
3. Matrix multiplication (Attention matmul)
4. Linear layer

The parallelism of the first projections ($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$) allows for a variety of possibilities that are summarized in table 5.2, with their advantages and drawbacks. From the possibilities offered by the board in terms of area and the time allowed to this research, **version (2.a) is chosen for this work.** In this approach projections are accelerated in parallel on the FPGA and the other operations are kept on HPS. For one wanting to develop an ASIC with constrained area and energy consumption, version (2.d) should be the most suited as it only needs one accelerating block that can be reused sequentially. Approach (3) also could be divided similarly as what is done for (1) and (2), i.e. parallelizing the projections or keeping them sequential. This approach has to be preferred for someone having a great area and development time, as it leads to a more intricate datapath than the other approaches, as well as added control logic. Moreover, a well-designed accelerator could handle the next operations (Softmax-matmul, matmul and linear layer) with the same accelerating block, and compute everything sequentially.

	Description	Advantages	Drawbacks
Approach (1)	a Compute Q , K and softmax-matmul on the FPGA while the HPS takes care of V , Q and K computation is sequential	Better use of time compared to other approaches because of maximum pipelining Saves logic utilization	More logic utilization than (2) Accelerator has to be fast enough to compute Q , K and softmax-matmul while HPS compute V . Otherwise HPS has to wait for Q and K leading to waste of time
	b Compute Q , K and softmax-matmul on the FPGA while the HPS takes care of V , Q and K computation is parallelized	No constrain on accelerator speed compared to (1.a)	More logic utilization than (1.a) and (2)
Approach (2)	a Compute Q , K and V on FPGA then let the HPS do the rest: Q , K and V in parallel	Maximum acceleration between (2.a),(2.b) and (2.c)	Greater area than (2.b) and (2.c)
	b Compute Q , K and V on FPGA then let the HPS do the rest: Q , K and V in sequential	Lower area than (2.a)	Slower than (2.a)
	c Compute Q , K and V on FPGA then let the HPS do the rest: Q , K are parallel and V is computed while HPS computes Softmax-matmul	Same area than (2.b) but greater acceleration since V is done in parallel with softmax-matmul	Slower than (1)
	d Compute Q , K and V on FPGA then let the HPS do the rest: Q , K are sequential and V is computed while HPS computes Softmax-matmul	Lower area than (2.a) and (2.c) Faster than (2.b)	Slower than (2.a) and (2.c)
Approach (3)	Compute everything on FPGA	Faster than (1) and (2)	Greater area than (1) and (2)

Table 5.2: Different possible system configurations with their respective advantages and drawbacks

5.1.2 Global overview of the system

The global overview of our Hardware-Software co-design as well as the different program steps are shown in Figure 5.1. More details about the communication between the HPS and FPGA are given in section 1.2.3.

First the HPS writes the input embedded vectors $\mathbf{E}$ in a shared memory called *Input Sequence Memory*. Once it is done the HPS sends a signal to a block called *Control Unit* through a Parallel Input/Output (PIO). The FPGA now knows it can start the projections in the block *Linear Layer*. The results of those matrix products are then written in shared memories called *K Memory*, *Q Memory* and *V Memory*. Once the FPGA has finished all the computations and the writing of the results, it sends a signal through a PIO to the HPS signalling the matrices $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ are ready. The HPS will read the *K Memory*, *Q Memory* and the *V Memory* to use the data they contain for the rest of the Attention mechanism.

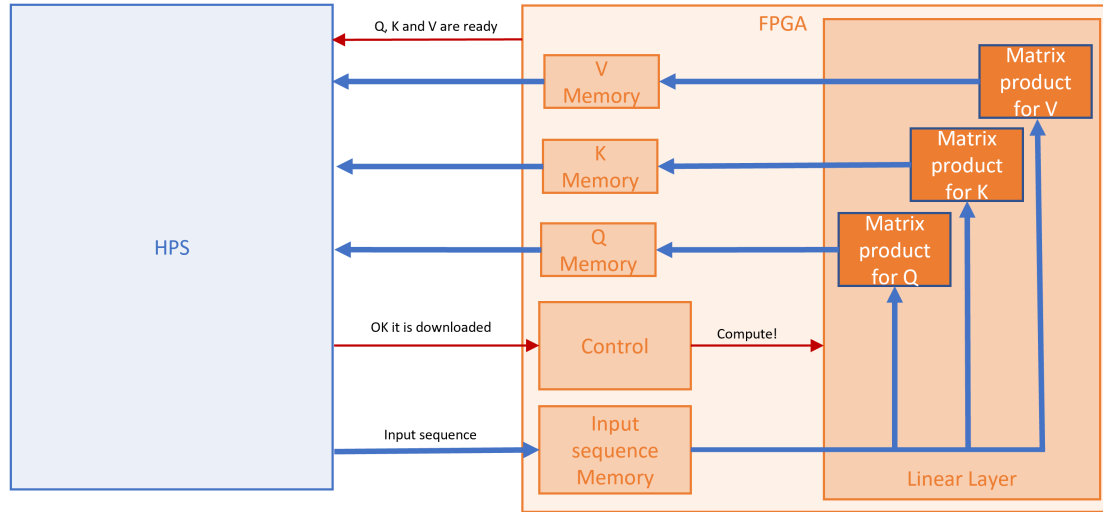


Figure 5.1: Global overview of the hardware design

Figure 5.2 illustrates the memories arrangement and communications. This data transfer is implemented *via* the Avalon-MM bridge presented in Section 1.2.3. The HPS masters the Avalon-MM AXI bridge (AXI being the used protocol [76]). Based on the address the HPS provided, the MM bridge then selects the adequate slave and reads/writes on it depending on the requested operation.

Our system contains four different memories: one to provide the embedded vectors, and three others to output the results of the projections. These memories have to be interfaced through a controller, which are themselves slaves of the MM bridge. The two Parallel Input/Output (PIO) are used for control reasons and in the same way as the memories: HPS reads and writes to what appears as addresses for it, and the Avalon bridge translates it into the correct behaviour.

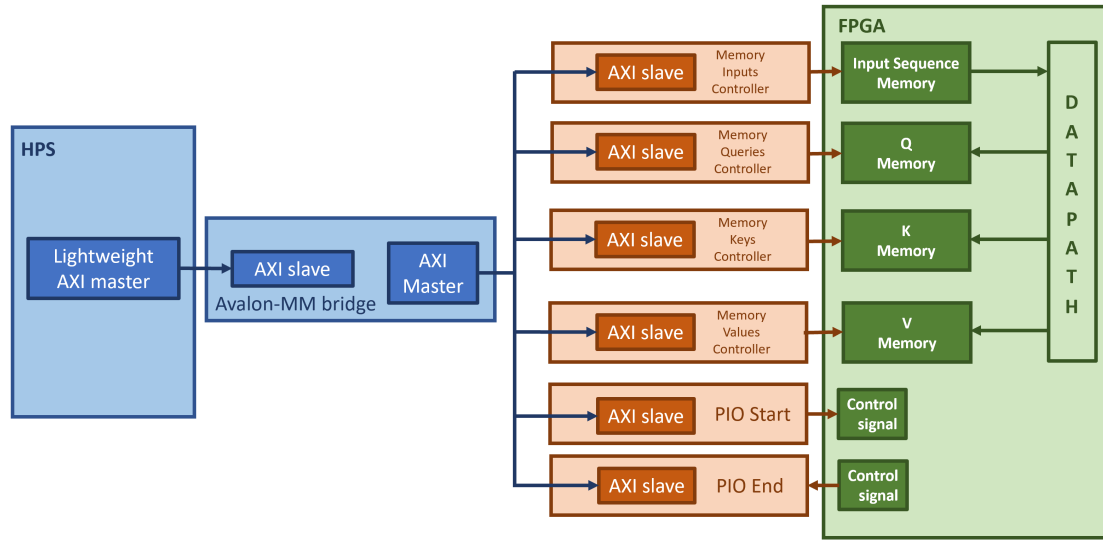


Figure 5.2: Overview of the complete system, the different parts and their interactions

5.2 Dot Product acceleration

As explained in section 5.1.1, the FPGA part of our system accelerates the matrix products $Q = E W_q$, $K = E W_k$ and $V = E W_v$ in parallel. A reminder of the parameters of our Transformer design is done in Table 5.3 and Table 5.4. The numerical values are chosen in Section 4.1 and the dimensions are precised in Section 3.1.2. All operations are performed in 8 bits.

Global parameters of the Transformer	
Name	Dimension
S	12
H	8
P	8
d_{model}	64
$\mathbf{E}$	$(S \times d_{model})$
$\mathbf{Q}$	$(S \times d_{model})$
$\mathbf{K}$	$(S \times d_{model})$
$\mathbf{V}$	$(S \times d_{model})$
$\mathbf{W}_q$	$(d_{model} \times d_{model})$
$\mathbf{W}_k$	$(d_{model} \times d_{model})$
$\mathbf{W}_v$	$(d_{model} \times d_{model})$

Table 5.3: Global parameters of the Transformer

Parameters of the Transformer for a head i	
Name	Dimension
$\mathbf{Q}^i$	$(S \times P)$
$\mathbf{K}^i$	$(S \times P)$
$\mathbf{V}^i$	$(S \times P)$
$\mathbf{W}_q^i$	$(d_{model} \times P)$
$\mathbf{W}_k^i$	$(d_{model} \times P)$
$\mathbf{W}_v^i$	$(d_{model} \times P)$

Table 5.4: Parameters of the Transformer for a head i

As explained in section 3.1.2.2, the Multi-head Attention parallelizes the computation of the scaled dot-product attention in H heads. For each head i , the matrix multiplications to compute become:

$$\mathbf{Q}^i = \mathbf{E} \mathbf{W}_q^i \quad (5.1)$$

$$\mathbf{K}^i = \mathbf{E} \mathbf{W}_k^i \quad (5.2)$$

$$\mathbf{V}^i = \mathbf{E} \mathbf{W}_v^i \quad (5.3)$$

For the following discussions, we choose as generic names:

- $\mathbf{W}^i$ to denote any of the trained weight matrices $\mathbf{W}_q^i$, $\mathbf{W}_k^i$ and $\mathbf{W}_v^i$, $\mathbf{W}^i \in \mathbb{R}^{(d_{model} \times P)}$
- Superscript i designates head i
- $\mathbf{C}^i = \mathbf{E} \cdot \mathbf{W}^i \in \mathbb{R}^{(S \times P)}$ to represent any of $\mathbf{Q}^i$, $\mathbf{K}^i$, $\mathbf{V}^i$, the projected matrices.

These matrix products will be processed by a custom SA in the hardware (see Section 3.1.5.3). Four different designs of the systolic array are possible. They allow to play with the trade off between processing time and resource allocation. These different

cases are developed hereunder and the results are summarized at the end of the section in Table 5.5.

Case 1

As a recall of section 3.1.5.3, the SA must have the shape of the matrix product result. It means that to compute $\mathbf{C}^i$, one needs $S \cdot P = 96$ PEs. Also, the number of needed cycles to finish the matrix product is thus $\max(d_{model} + S - 1, d_{model} + P - 1)$ cycles (75 in our case). The problem with a design having an allocated area which depends on the sequence length S is the board could not be able to create all the needed PEs, due to a lack of available resources. Indeed, S could dramatically increase. Also, the sequence length could vary from one inference to another and our system could not work because the dimensions could not fit with our design anymore. This is depicted in Figure 5.3a. This first case is thus not implemented.

Case 2

To overcome this, our design will process each embedded vector $\mathbf{e}^s \in \mathbb{R}^{d_{model}}$ one by one. $\mathbf{e}^s$ is thus the row of $\mathbf{E}$ at index s ($0 \leq s < S$). To do so, *pipelining* is used. Pipelining allows the system to start a new computation without waiting the end of the previous one. Using this technique reduces the number of PEs required to compute $\mathbf{C}^i$ to 8 ($= 1 \cdot P$). Indeed, the matrix multiplication will have the following shape: $(1 \times d_{model})(d_{model} \times P) = (1 \times P)$. The drawback of this area reduction is the increase of the needed cycles. Indeed, this technique uses $S \cdot (d_{model} + P - 1)$ ones. In our case this gives 852 cycles. This case is presented in Figure 5.3b. We denote $\mathbf{c}^{s,i} = \mathbf{e}^s \cdot \mathbf{W}^i \in \mathbb{R}^{(1 \cdot P)}$, a row of a projected matrix. Therefore this case is not implemented either.

Case 3

In section 3.1.5.3, we have also seen that zeros are inserted in the input vectors to get the right inputs at the right time in each PE. This insertion leads to a unwanted delay in the process. It is possible to remove this delay simply by considering the trained

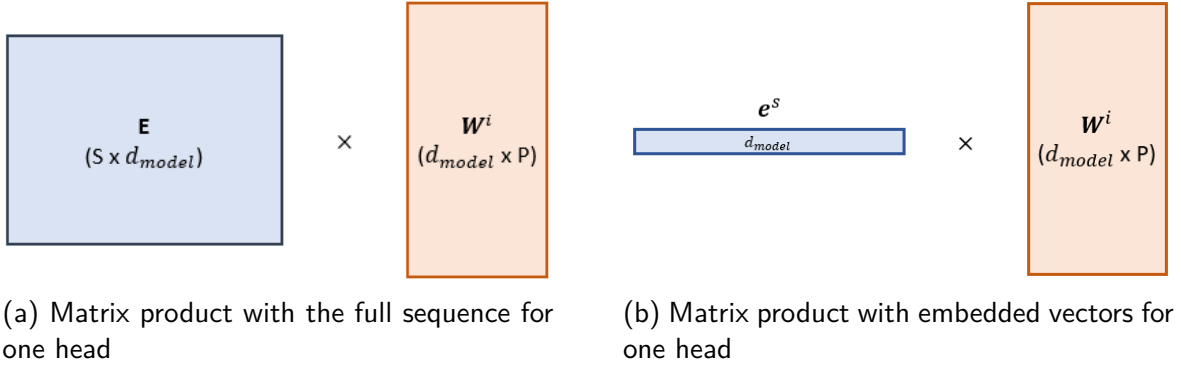


Figure 5.3: Input sequence

weight matrix as P independent columns as shown equation 5.4:

$$\mathbf{c}^{s,i} = \mathbf{e}^s \begin{bmatrix} \mathbf{w}_0^i & \mathbf{w}_1^i & \dots & \mathbf{w}_{P-1}^i \end{bmatrix} \quad (5.4)$$

$$c_p^{s,i} = \mathbf{e}^s \mathbf{w}_p^i \quad (5.5)$$

where $\mathbf{w}_p^i \in \mathbb{R}^{(d_{model} \times 1)}$ and $0 \leq p < P$.

By doing this we replace one SA of shape $(1 \times P)$ into P parallel SA of shape (1×1) because the matrix products will have the shape $(1 \times d_{model}) \cdot (d_{model} \times 1) = (1 \times 1)$. This is represented in Figure 5.4a. This accelerates the matrix product, taking in this case d_{model} cycles to get the result $\mathbf{c}^{s,i}$, which is the concatenation of the P results $c_p^{s,i}$ of the PE_p . The matrix $\mathbf{c}^i$ is thus obtained in 768 cycles ($S \cdot d_{model}$).

This improves the speed by 11% without adding any additional resources and it is represented in Figure 5.4a.

Case 4

It is possible to further accelerate the process with a technique inspired of the *Column Balanced Block Pruning* acceleration in subsection 3.1.5.2. Indeed we divided the input embedded vector $\mathbf{e}^s$ into N embedding blocks which gives $\mathbf{e}_n^s \in \mathbb{R}^{(1 \times \frac{d_{model}}{N})}$ as well as the independent weights columns into N weights blocks giving $\mathbf{w}_{n,p}^i \in \mathbb{R}^{(\frac{d_{model}}{N} \times 1)}$. With $0 \leq n < N$, we have now:

$$\mathbf{c}^{s,i} = \begin{bmatrix} \mathbf{e}_0^s & \mathbf{e}_1^s \dots & \mathbf{e}_{N-1}^s \end{bmatrix} \begin{bmatrix} \mathbf{w}_{0,0}^i & \mathbf{w}_{0,1}^i & \dots & \mathbf{w}_{0,P-1}^i \\ \mathbf{w}_{1,0}^i & \mathbf{w}_{1,1}^i & \dots & \mathbf{w}_{1,P-1}^i \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{w}_{N-1,0}^i & \mathbf{w}_{N-1,1}^i & \dots & \mathbf{w}_{N-1,P-1}^i \end{bmatrix} \quad (5.6)$$

$$c_p^{s,i} = \mathbf{e}_0^s \mathbf{w}_{0,p}^i + \mathbf{e}_1^s \mathbf{w}_{1,p}^i + \dots + \mathbf{e}_{N-1}^s \mathbf{w}_{N-1,p}^i = \sum_{n=0}^{N-1} \mathbf{e}_n^s \mathbf{w}_{n,p}^i = \sum_{n=0}^{N-1} c_{n,p}^{s,i} \quad (5.7)$$

This structure is shown in Figure 5.4b. The advantage of this is that all the $c_{n,p}^{s,i}$ are computed in parallel at the same time and requires less clock cycles than the $c_p^{s,i}$. The disadvantage is that we must process one extra addition to get the result $c_p^{s,i}$ as it can be seen in equation 5.7. To accelerate the addition, we add the $c_{n,p}^{s,i}$ two by two in parallel, then the results of each addition two by two, etc (see Figure 5.5). One thing to note is that N must divide e^s and $\mathbf{w}_p^i$ in equal sized blocks. In our design, $d_{model} = 64$. It means that N can have the following values : 1, 2, 4, 8, 16, 32, 64. One can note all of these numbers are powers of 2 and $\log_2 N$ clock cycles are needed to compute this addition.

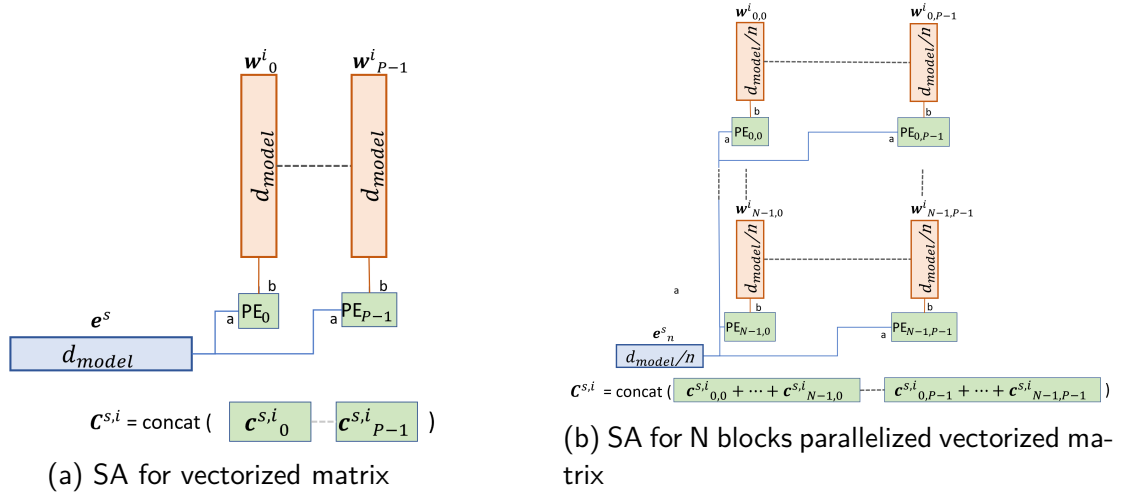


Figure 5.4: SA for matrix multiplication acceleration

Choice of N

Parameter N has an influence on the number of PEs in the hardware architecture and

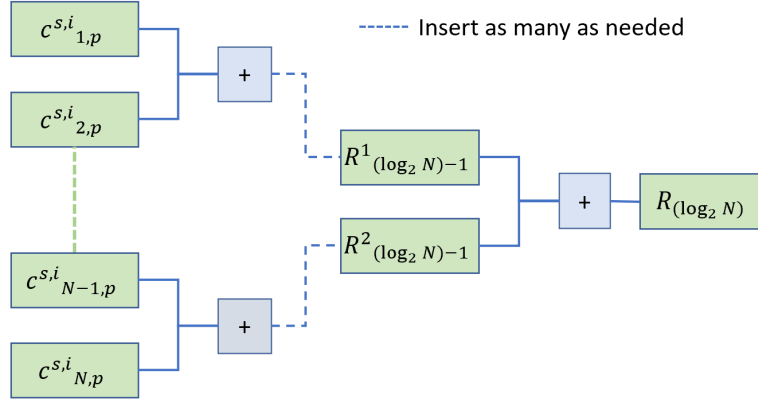


Figure 5.5: Accelerated addition

on the number of needed cycles to complete the projections.

As show in Figure 5.6a, the number of PE will linearly increase depending on the value of N . Indeed:

$$\text{Number of needed PE's to compute } \mathbf{c}^{s,i} = H \cdot N \cdot P \cdot 3 \quad (5.8)$$

the factor 3 comes because the Multi-Head Attention is composed of 3 projections in parallel: $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$.

Figure 5.6b shows the needed number of cycles depending on N . we have:

$$\text{Number of needed cycles to compute } \mathbf{c}^{s,i} = \frac{d_{model}}{N} + \log_2 N \quad (5.9)$$

$$\text{Number of needed cycles to compute } \mathbf{C}^i = \left(\frac{d_{model}}{N} + \log_2 N \right) \cdot S \quad (5.10)$$

We can observe that the number of needed cycles dramatically decreases when N is increasing. Also, setting N to 32 or 64 gives the same number of needed cycles, but setting N to 64 implies twice as many PEs than setting a N to 32. One can conclude that it isn't interesting to set N higher than 32. Finally, the choice of N depends on the available resources on the boards. The limit of the DE-10 Nano with our design was setting a N to 4. As we can see on the graph, it is still a very interesting acceleration

since it only requires 18 cycles to compute all the matrix multiplications between an embedded vector and a weight matrix.

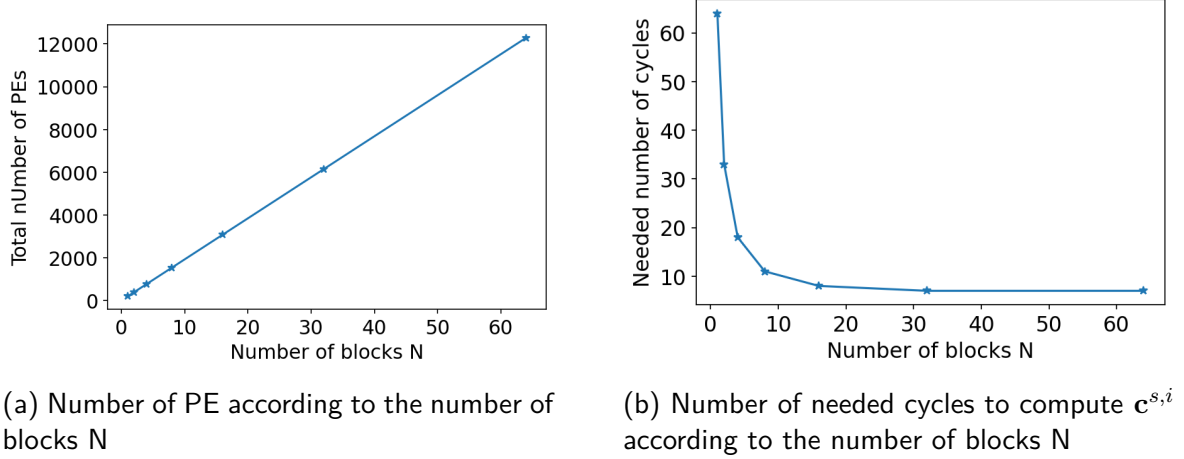


Figure 5.6: Influence of N on the system

Table 5.5 summarizes and compares the number of needed cycles to compute $c^{s,i}$ and the number created PEs of all the different cases. One thing to note is that d_{model} is often greater than S . A trade off between the allocated resources and the latency must be done. On the one hand, *Case 1* offers the fastest processing time but needs way more resources than all the other cases. On the other hand, *Case 2* and *Case 3* offer the most economic at the price of an increased latency. The best proposed solution is the *Case 4* since it offers a reduction of the needed resources *Case 1* needs while decreasing the latency of *Case 2* and *Case 3*. This the reason why **Case 4** is chosen for this work.

	Number of needed cycles to compute Q, K and V	Number of needed PE's
Case 1	$\max(d_{model} + S - 1, d_{model} + P - 1)$	$S \cdot P \cdot H \cdot 3$
Case 2	$(d_{model} + P - 1) \cdot S$	$1 \cdot P \cdot H \cdot 3$
Case 3	$d_{model} \cdot S$	$1 \cdot P \cdot H \cdot 3$
Case 4	$(\frac{d_{model}}{N} + \log_2 N) \cdot S$	$N \cdot P \cdot H \cdot 3$

Table 5.5: Summary of the different SA designs

5.3 Digital circuit

This section aims at describing the digital circuit we design in its whole, flowing through all the hierarchy levels. In this section, the notation $A[y:x]$ comes from the *Verilog* language. It points the bits going from x to y (x being a less significant bit than y) of a register or multiple-bit wire named A . A variable is declared as $A[(l-1):0]$ where l is the bit width of the variable.

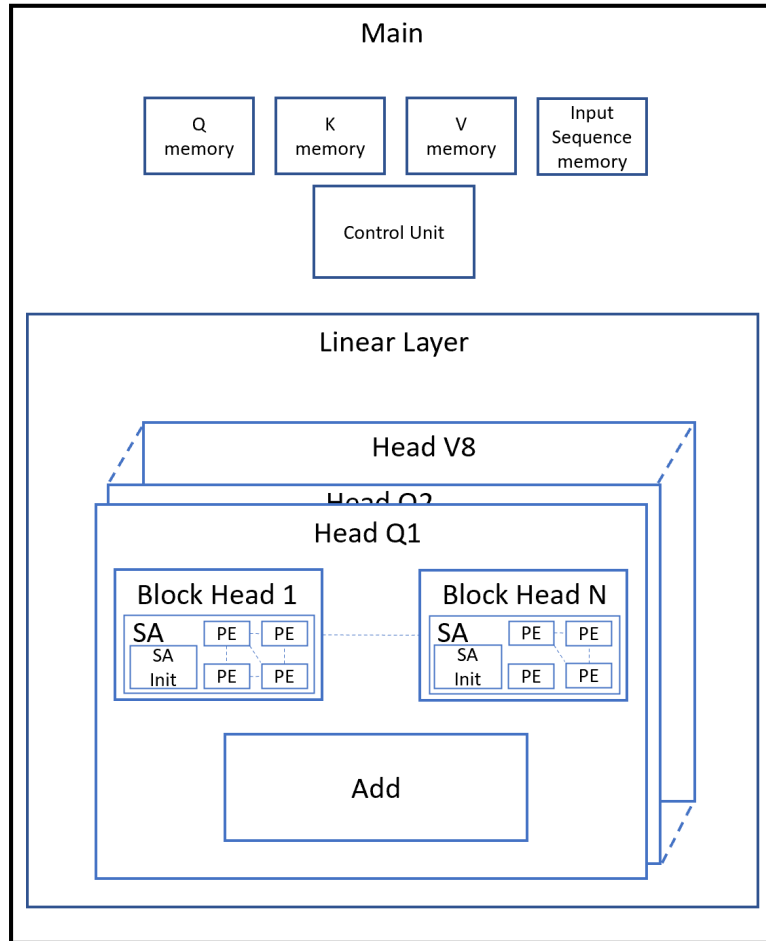


Figure 5.7: Hierarchy of the digital circuit

Figure 5.7 shows the hierarchy of the hardware architecture. On the same level, one can find the four on-chip memories, a *Control Unit* and the *Linear Layer*. The on-chip memories *Q memory*, *K memory*, *V memory* and *input sequence memory* store the input sequence and the results of the projections **Q**, **K** and **V** respectively. The

Control Unit coordinates the HPS and the FPGA. Indeed as seen in Figure 5.1 both parts have to communicate with each other to be able to synchronize the beginning and the end of the computation. The *Linear Layer* performs the computations of equation 5.7 of each head for each projection in parallel. It is built in sub-blocks that will be explained in this section from the highest level (*Linear layer* block) to the lowest level (*PE* block).

5.3.1 Memories

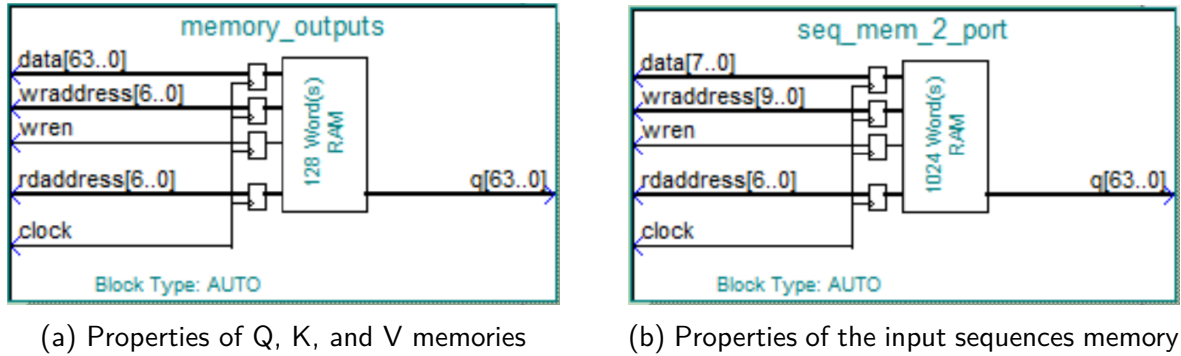


Figure 5.8: Memories properties

***Q* memory, *K* memory and *V* memory**

Those memories are 2 Port RAM on chip memories with one read port and one write port, as shown in Figure 5.8a. Their role is to be filled by the results of the computation of the FPGA part. To properly choose the capacity of the memory, one has to refer to Table 5.3. **Q**, **K** and **V** have a dimension $(S \times d_{model})$. Each data has a Data Size (D_S) of 8 bits meaning $S \cdot D_S \cdot d_{model} = 6144$ bits (768 bytes) are needed to store the result of one projection.

We chose and justify in Section 5.3.4.1 a word size of 64 for the memories. It means that a depth of at least 96 ($6144/64 = 96$) is needed. The smallest available depth higher than 96 we have at disposition is 128. This leads to a total capacity of 1024 bytes. One thing to note is that the size of the memory depends on the size of the sequence length which constrains the scalability of our system if we want to translate longer sequences. To circumvent this limitation, the HPS could read in memory

each time an embedded vector has been processed. It requires more complexity and may increase the processing time since memory access is much slower than projection computation, as highlighted in section 5.4. A solution would be to create deeper memories to cover longer sequences.

Input sequences memory

Figure 5.8b shows the properties of the *Input Sequences Memory*. This is a 2 Port RAM on chip memory with one read port and one write port. This memory is filled by the HPS. Just as the **Q**, **K** and **V** memories, the *Input Sequences Memory* must have a capacity of $S \cdot d_{model} \cdot D_S = 6144$ bits because the embedding matrix **E** has a shape of $(S \times d_{model})$. The difference here is that the HPS fills the memory one byte at the time, for reasons particular to the Altera functions which provide a read/write access from to HPS to the FPGA. This is why the width for the written data must be 8. It means that we needed a depth of 768 ($6144/8 = 768$ words). The smallest available depth higher than 768 was 1024.

5.3.2 Control unit

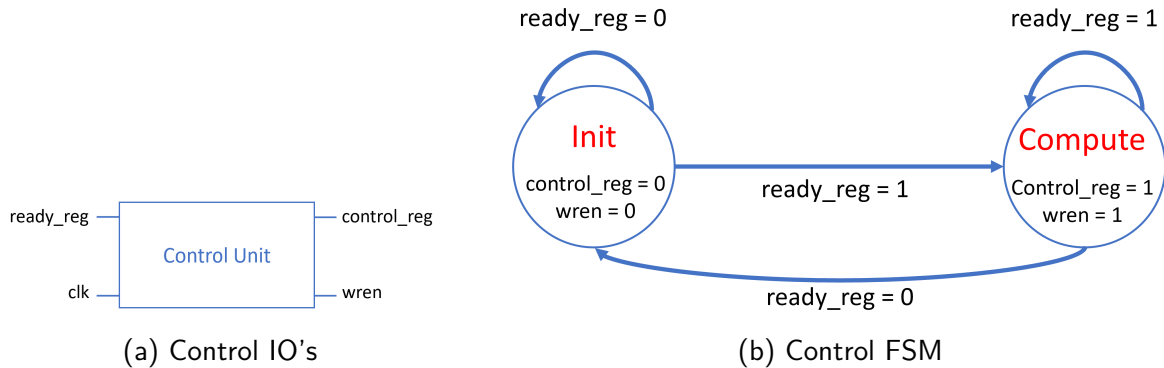


Figure 5.9: Control Unit

The control unit is a very simple finite state machine composed of two states: *Init* and *Compute*. The control unit has two inputs: *clk*, *ready_reg* and two outputs: *wren*, *control_reg* (see Figure 5.9). As shown in Figure 5.1 the FPGA has to wait until the HPS has written all the embedded vectors in the *Input sequence memory*. The mean

the HPS lets the FPGA know that this writing process is finished is through a Parallel Input/Output (PIO), mastered by the HPS. In the FPGA, the register *ready_reg* stores the value of the PIO and its value is updated at each clock cycle of the FPGA *clk*.

The output *wren* stands for *write enable*. It is used to enable the write process of the results in the Q, K and V memories. The second output *control_reg* is used to enable the start of the computation.

During the wait time, the register *ready_reg* is at the low level since the HPS has set the PIO to 0. The FPGA stands in the *Init* state. In this state, no result must be written in memories as they are not ready. The signal *wren* is thus set to 0.

Once the HPS has finished to write the input sequence, it sets the value of the PIO to 1 which changes the value of *ready_reg* to 1. This changes the state of the FSM to the *Compute* one. In this state the FPGA starts the computation of the matrix products. Since the FPGA has to write its results in memories, the control module sets *wren* to 1. When the HPS receives the complete projections and do not need the results anymore it sets *ready_reg* to 0. This allows the FPGA to return to state *Init* and wait until a new projection is required.

5.3.3 Linear Layer

Figure 5.10 shows the structure of the *Linear Layer* module. The linear layer is the body of the matrix products. It takes an embedded vector $\mathbf{e}^s$ named *seq_row* as input which has a $D_S \cdot d_{model} = 512$ bits width. The *Linear Layer* module outputs the three products $\mathbf{c}_{q,k,v}^s = \mathbf{e}^s \cdot \mathbf{W}_{q,k,v}$ between an embedded vector and the three weight matrices $\mathbf{W}_q$, $\mathbf{W}_k$ and $\mathbf{W}_v$. The outputs *add_Q*, *add_K*, *add_V* are the concatenation of the $\mathbf{c}^{s,i} = \mathbf{e}^s \cdot \mathbf{W}^i$ computed in parallel for each head i of the Queries, Keys and Values.

In order to gain in design simplicity and processing speed, we decide to store the weights matrices locally in the module by using 512 bits registers. Each register stores one column of the lasts. Since $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{(d_{model} \times d_{model})}$, and we need to store

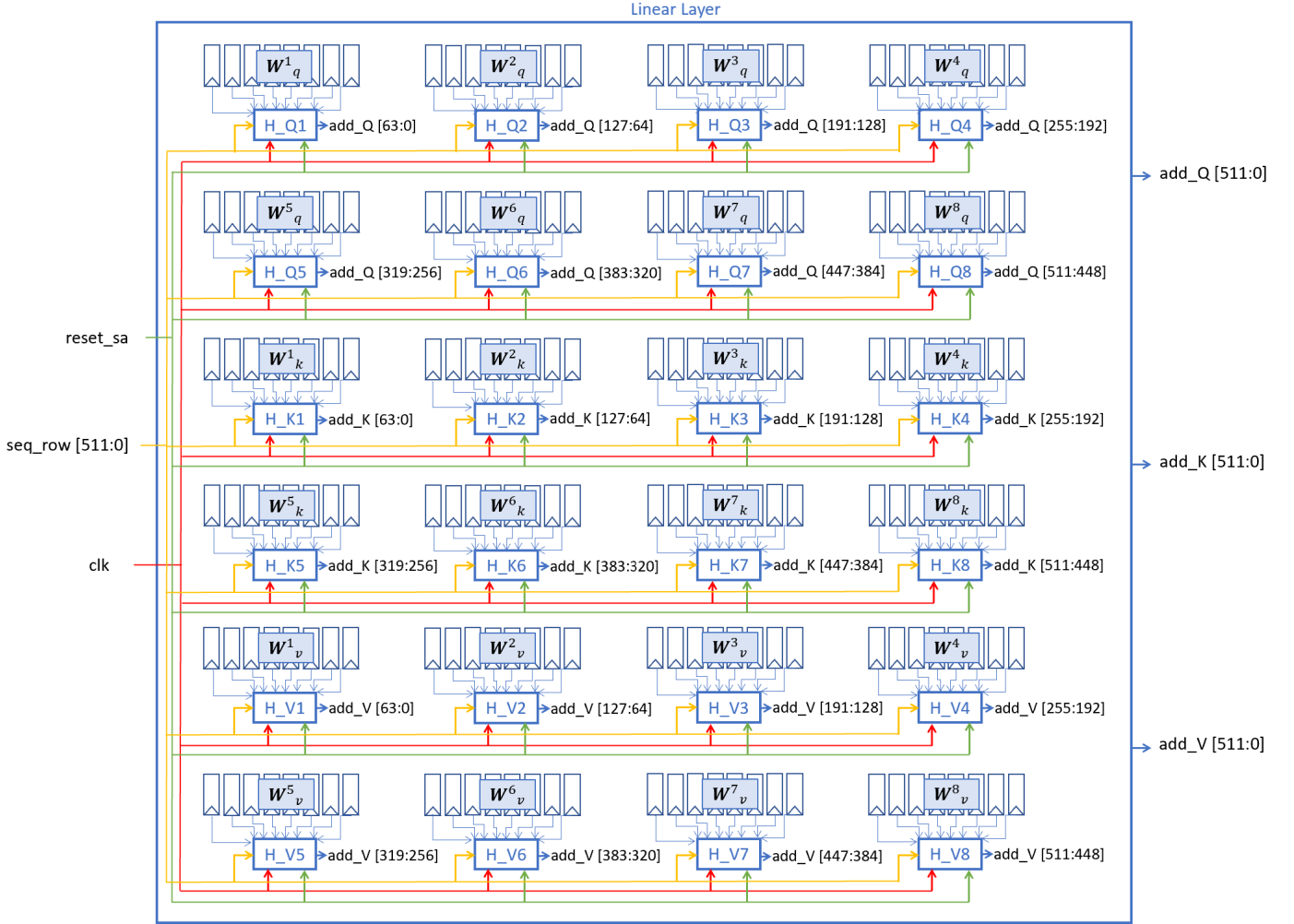


Figure 5.10: *Linear Layer* module

three of them, we need

$$\frac{3 \cdot d_{model}^2 \cdot D_S}{512} = 192$$

registers of size 512.

In order to compute all the $\mathbf{c}^{s,i} = \mathbf{e}^s \cdot \mathbf{W}^i \in \mathbb{R}^{(1 \times P)}$ explained in section 5.2, 8 instances of the *head* module ($H_{Q1}, \dots, H_{Q8}, H_{K1}, \dots, H_{K8}, H_{V1}, \dots, H_{V8}$) are created for each matrix product. Each of these blocks will be responsible for filling part of the output vector. The following sections will now introduce the implementation of the Linera Layer. Each section is a building block of the preceding one. Figure 5.7

contains all the presented blocks as well as their place in the hierarchy.

5.3.3.1 Head module

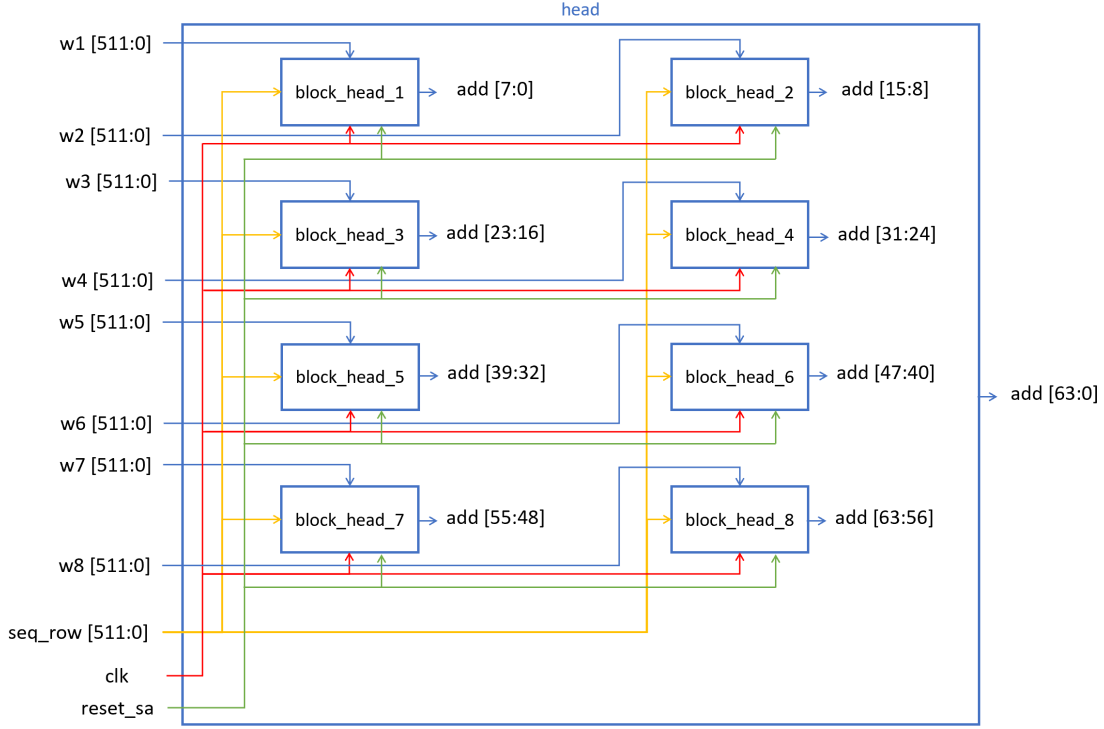


Figure 5.11: *head* module

Figure 5.11 shows the structure of the *head* module. This module parallelizes the matrix product of one head into P vector products (equation 5.7). Each head instance takes $P = 8$ weight vectors $\in \mathbb{R}^{(d_{model} \times 1)}$. As each data is represented in 8 bits, each weight vector is 512 bits wide. The concatenation of those 8 weight vectors forms what we previously called $\mathbf{W}^i$.

To be able to parallelize the computation of each head, we create P instances of the module *block_head* which will compute $c_p^{s,i}$ of equation 5.7. The output of each *block_head* module will fill part of the global *add* output. The output will thus be the concatenation of the P $c_p^{s,i} \in \mathbb{R}^{(1 \times 1)}$, i.e. $add = c^{s,i} \in \mathbb{R}^{(1 \times P)}$. Each data having an 8-bits representation this leads to a total width of 64 bits.

5.3.3.2 Block_head module

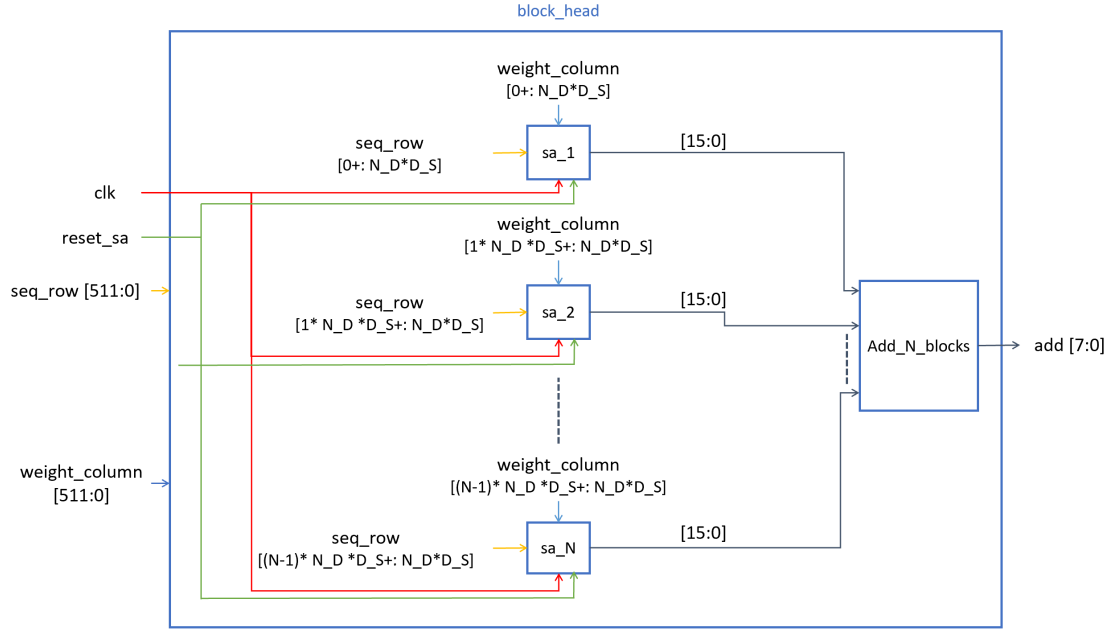


Figure 5.12: *Block_head* module

This module further parallelizes the computations by splitting the vector multiplication in N blocks sa_n that compute the $c_{n,p}^{s,i}$ of equations 5.7. Again, each $c_{n,p}^{s,i}$ has an 8 bits representation. The N results are then added in a block Add_N_blocks to get $c_p^{s,i}$ using the accelerated addition depicted in Figure 5.5. Each sa_n block takes as inputs a N^{th} fraction of the embedded vector (seq_row) and a N^{th} part of the weight vector. This is shown in Figure 5.12 with the input vectors being split between the different sa_n blocks. N_D denotes the number of data in each block vector ($\frac{d_{model}}{N}$).

The output of the sa_n blocks is 16 bits due to the inner multiplication. The block Add_N_blocks outputs the 8 most significant bits of the result to respect the quantization scheme described in Section 1.1.1.4.

5.3.3.3 Systolic_array module

This section aims at explaining the sa_n blocks of Section 5.3.3.2. The module presented here is a Systolic Array (SA). Before explaining the architecture of this

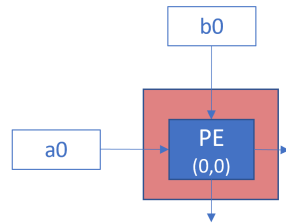
module, some discussions about different possible scenario concerning the shape of the systolic array must be done.

Boundary conditions of a systolic array of shape $(I \times J)$

As it can be observed in Figure 3.17 and in Figure 3.18a, all the PEs are interlinked together. They propagate their inputs in_a and in_b to their neighbors through their outputs out_a and out_b . The challenge is to determine where the input should come from and where it has to be propagated. For example, as it can be observed in Figure 3.17, the PEs on the left border should take their inputs from the first matrix and not from their neighbors. These considerations lead to boundary conditions that are developed hereafter.

Case $I = 1, J = 1$

As it can be observed in Figure 5.13 and in Table 5.6, only one boundary case appears. The inputs of the PE are linked to the input vectors $\mathbf{a} \in \mathbb{R}^{(1 \times 1)}$, $\mathbf{b} \in \mathbb{R}^{(1 \times 1)}$ and none of its outputs are linked.



Color	In_a	In_b	Out_a	Out_b
	a_0	b_0	/	/

Table 5.6: linked I/O of each PE according to its position for the case $I = 1, J = 1$

Figure 5.13: Boundary conditions of a systolic array for the case $I = 1, J = 1$

case $I = 1, J > 1$

With this shape, three boundary cases appear as it can be seen in Figure 5.14 where $0 \leq i < I$ and $0 \leq j < J$. Table 5.14 gives the linked inputs and outputs of the PE's belonging to those boundary cases.

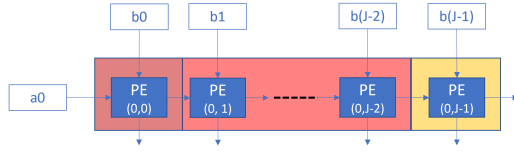


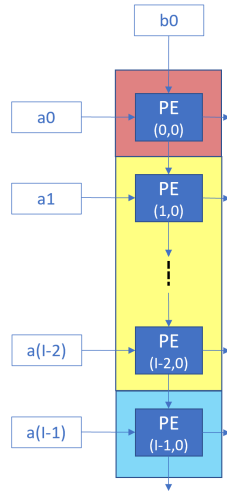
Figure 5.14: Boundary conditions of a systolic array for the case $I = 1$, $J > 1$

Color	In_a	In_b	Out_a	Out_b
	a_0	b_0	$PE_{(i,j+1)}$	/
	$PE_{(i,j-1)}$	b_j	$PE_{(i,j+1)}$	/
	$PE_{(i,j-1)}$	b_{J-1}	/	/

Table 5.7: linked I/O of each PE according to its position for the case $I = 1$, $J > 1$

case $I > 1$, $J = 1$

With this shape, three boundary cases appear as it can be seen in Figure 5.15. Table 5.15 gives the linked inputs and outputs of the PE's belonging to those boundary cases.



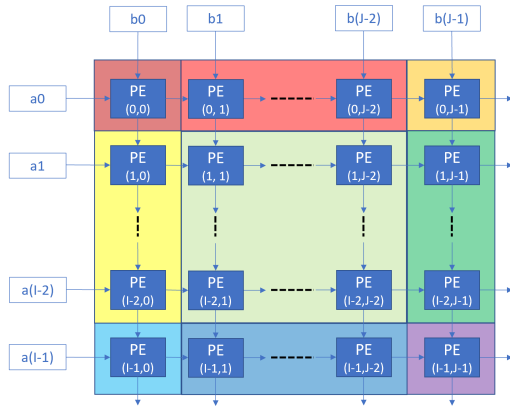
Color	In_a	In_b	Out_a	Out_b
	a_0	b_0	/	$PE_{(i+1,j)}$
	a_j	$PE_{(i-1,j)}$	/	$PE_{(i+1,j)}$
	a_{I-1}	$PE_{(i-1,j)}$	/	/

Figure 5.15: Boundary conditions of a systolic array for the case $I > 1$, $J = 1$

Table 5.8: I/O of each PE according to its position the case $I > 1$, $J = 1$

case $I > 1$, $J > 1$

With this shape, nine boundary cases appear as it can be seen in Figure 5.16. Table 5.16 gives the linked inputs and outputs of the PE's belonging to those boundary cases. Note that this is the most general case.



Color	In _a	In _b	Out _a	Out _b
	a ₀	b ₀	PE _(i,j+1)	PE _(i+1,j)
	PE _(i,j-1)	b _j	PE _(i,j+1)	PE _(i+1,j)
	PE _(i,j-1)	b _{J-1}	/	PE _(i+1,j)
	a _j	PE _(i-1,j)	PE _(i,j+1)	PE _(i+1,j)
	PE _(i,j-1)	PE _(i-1,j)	PE _(i,j+1)	PE _(i+1,j)
	PE _(i,j-1)	PE _(i-1,j)	/	PE _(i+1,j)
	a _{I-1}	PE _(i-1,j)	PE _(i,j+1)	/
	PE _(i,j-1)	PE _(i-1,j)	PE _(i,j+1)	/
	PE _(i,j-1)	PE _(i-1,j)	/	/

Figure 5.16: Boundary conditions of a systolic array for the case $I > 1, J > 1$

Table 5.9: linked I/O of each PE according to its position for the case $I > 1, J > 1$

Systolic array in the hardware

The SA block is now introduced. To understand it one needs to appreciate how a matrix is stored in hardware registers.

Let's define two generic input matrices as $\mathbf{MA} \in \mathbb{R}^{r_{MA} \times c_{MA}}$ and its hardware representation $MA[D_S * r_{MA} * c_{MA} - 1 : 0]$, as well as $\mathbf{MB} \in \mathbb{R}^{r_{MB} \times c_{MB}}$ ($MB[D_S * r_{MB} * c_{MB} - 1 : 0]$).

From an hardware point of view, each input matrix $\mathbf{M} \in \mathbb{R}^{I \times J}$ is flattened in a register as shown in Figure 5.17. Each $\mathbf{M}_{i,j}$ has a size D_S bits. LSBs and MSBs denote the the Least Significant Bits and Most Significant Bits respectively.



Figure 5.17: Matrix flattened in a register

We want to design a hardware as scalable as possible. This is the reason why we design a module (see Figure 5.18) capable of adapting itself to any $(I \times J)$ shape. It is composed of two main parts: a block `s_a_init` that updates the input vectors **a** and **b** at each clock cycle and the systolic array in itself containing $I \times J$ PE blocks.

To link the PEs, two intermediate wires a_int $[D_S \cdot r_{MA} \cdot c_{MB} - 1 : 0]$ and b_int $[D_S \cdot c_{MB} \cdot r_{MA} - 1 : 0]$ are used to make the connections between the inputs/outputs of the PEs. Their size allows each PE to use a D_S bit subset of them. a_int (resp. b_int) is used to connect out_a 's (resp. out_b 's) with in_a 's (resp. in_bs) of the PEs.

All the PEs are instantiated respecting the linked I/Os of Table 5.9. Each PE accumulates its result in a register out_c which is properly wired to the output of the module c $[(O_S * r_{MA} * c_{MB}) - 1 : 0]$, where O_S is the output data size of the multiplication in the PE block. O_S differs from D_S because since two 8-bits inputs are multiplied, the result is a 16-bits integer. It will then be scaled down to 8-bits by the Add_N_blocks . Each output of the PE blocks is wired to a portion of the output c depending on its position in the SA. The connection is done to get something similar to what is presented in figure 5.17.

In our final design, the module only generates one PE block because we chose to work with arrays of shape (1×1) to increase the speed of the process but our system could work with systolic arrays of any $(I \times J)$ shape. One can note we use the rules of Table 5.9 and not of Table 5.6. The only difference is that the out_a and out_b are linked to the wires a_int and b_int . In our design these lasts will never be used but allows the architecture to be very scalable.

5.3.3.4 S_a_init module

Figure 5.19 depicts the structure of the module. The challenge of this block is to send the right inputs $\mathbf{a} \in \mathbb{R}^I$ and $\mathbf{b} \in \mathbb{R}^J$ to the PEs, at the right time. We design an algorithm capable of sequentially updating $\mathbf{a}$ and $\mathbf{b}$, following the rules depicted in Figure 3.17 and Table 3.5. The structures of $a[D_S * r_{MA} - 1 : 0]$ and $b[D_S * c_{MB} - 1 : 0]$ are the same as for the flattened matrix (Figure 5.17), a_0 being the D_S LSBs and a_{I-1} being the D_S MSBs of $a[D_S * r_{MA} - 1 : 0]$. To create this pattern, Algorithms 1 and 2 are used, given for generic matrices $MA[D_S * r_{MA} * c_{MA} - 1 : 0]$ and $MB[D_S * r_{MB} * c_{MB} - 1 : 0]$. They depend on a counter that increases with the clock.

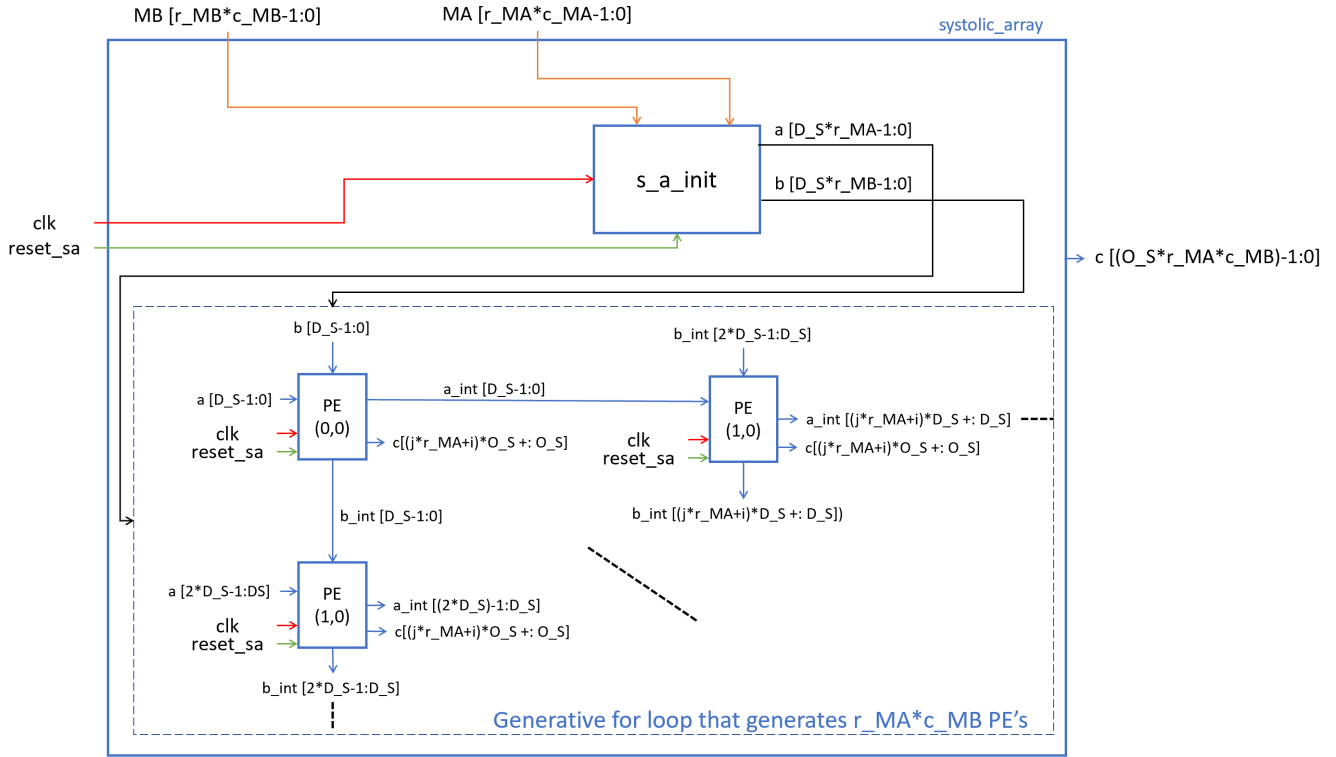


Figure 5.18: Structure of the systolic array module

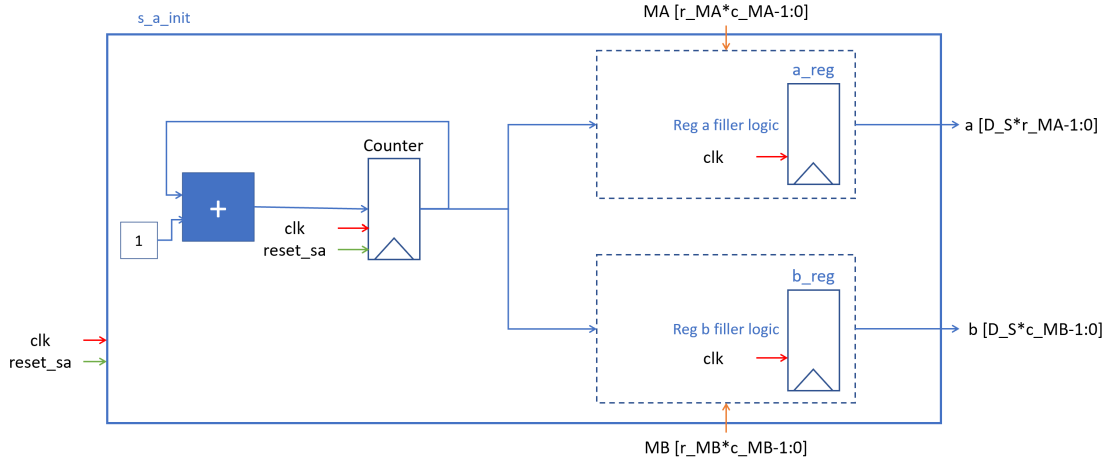


Figure 5.19: Structure of the s_a_init module

The wire $reset_sa$ is used to reset the counter to 0. This allows to start a new matrix product after resetting the counter.

Algorithm 1: update reg_a depending on the counter value

```
cond_a ← (counter < r_MA + c_MA - 1)
for m ← 0, r_MA - 1 do
  if cond_a and ~ reset then
    if (m > counter) or (counter > (m + c_MA - 1)) then
      | a_reg[m * D_S + : D_S] ≤ 0
    else
      | a_reg[m * D_S + : D_S] ≤
        MA[((m * c_MA) + (counter - m)) * D_S + : D_S]
    end
  else
    | a_reg ≤ 0
  end
end
end
```

Algorithm 2: update reg_b depending on the counter value

```
cond_b ← (counter < r_MB + c_MB - 1)
for n ← 0, c_MB - 1 do
  if cond_b and ~ reset then
    if (n > counter) or (counter > (n + r_MB - 1)) then
      | b_reg[n * D_S + : D_S] ≤ 0
    else
      | b_reg[n * D_S + : D_S] ≤ MB[((counter - n) * c_MB + n) * D_S + : D_S]
    end
  else
    | b_reg ≤ 0
  end
end
end
```

5.3.3.5 PE module

Now that the SA has been described, let's introduce its main building block: the Processing Element (PE). The designed architecture is similar to Figure 3.18b, and presented in more details in Figure 5.20. At each clock cycle, the value of the register *c_out_reg* stores:

$$c_out_reg \leq in_a \cdot in_b + c_out_reg \quad (5.11)$$

At each clock cycle, the inputs $in_a[D_S-1:0]$ and $in_b[D_S-1:0]$ are stored in registers $in_a_reg[D_S-1:0]$ and $in_b_reg[D_S-1:0]$. The wire $reset_sa$ is used to set all the output registers to 0 when a reset is called. This way, all registers of the systolic array are reset to 0 and allows the SAs to start a new matrix product.

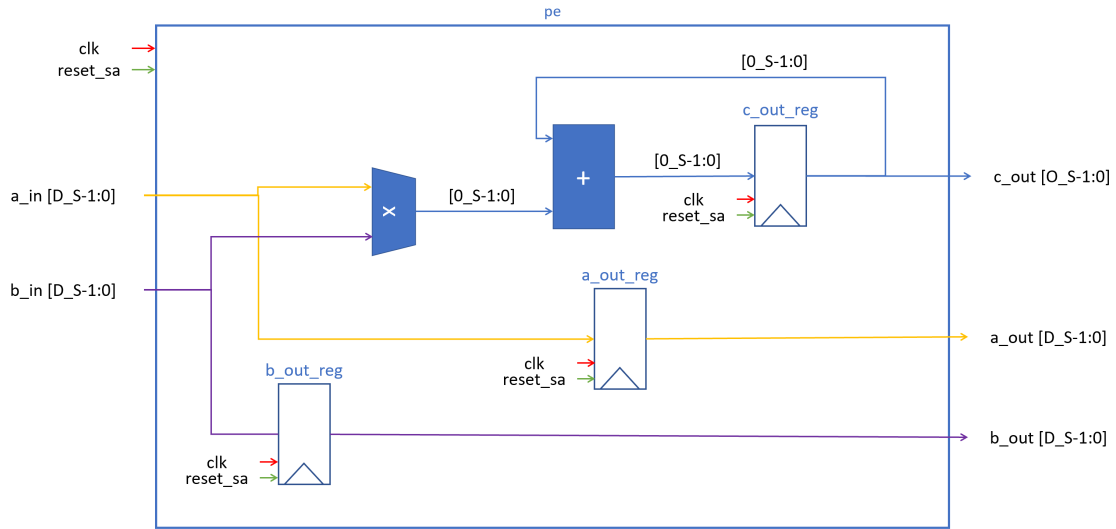


Figure 5.20: Architecture of the pe module

5.3.4 Pipelining

Our system happens in three steps: read inputs from memory, process matrix computations and write the results in output memories. To be more efficient, we decide to use pipelining. As stated in Section 5.2 pipelining allows the system to start a new computation without waiting the end of the previous one. In our case, this means reading the next inputs while processing the current ones and writing the result of the previous matrix product. This is represented in Figure 5.21, where the three main parts are represented in yellow, orange and green respectively. Those parts are separated by clocked registers, updated at the same time, allowing data to flow synchronously from one part to another.

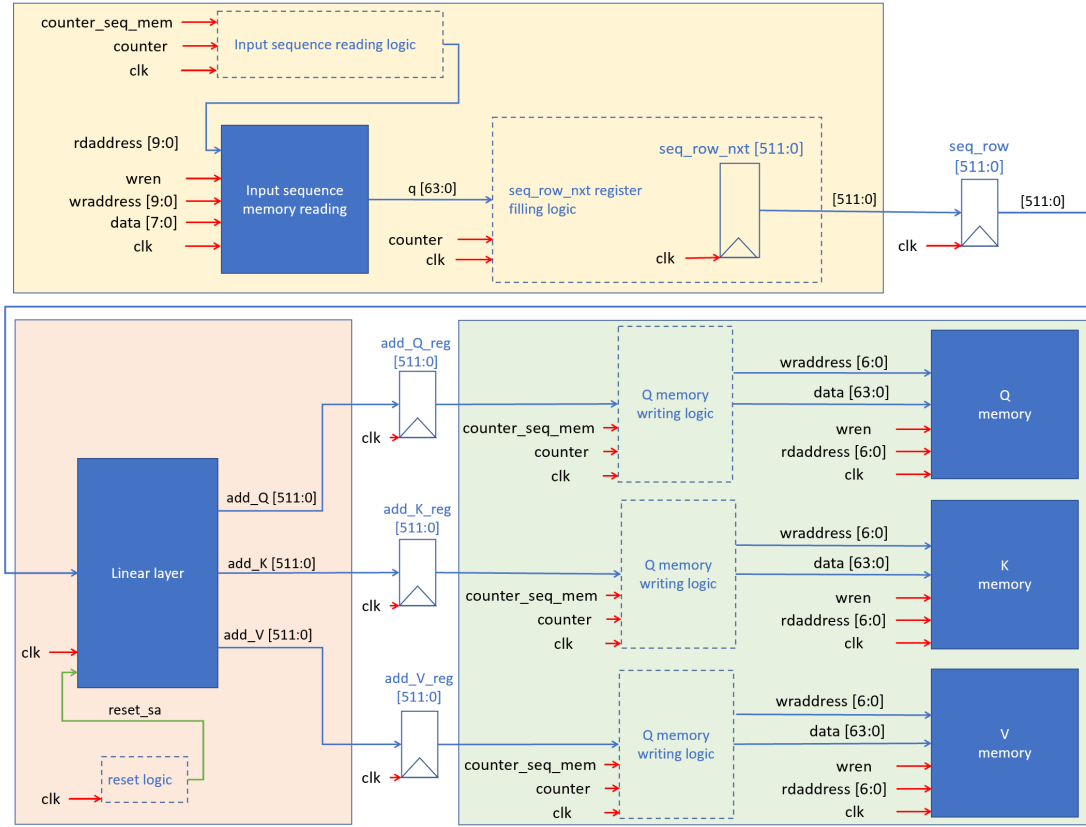


Figure 5.21: Pipelining in the *main* module

5.3.4.1 Pipelining timing

The main challenge with pipelining is to assess the time each part is taking to perform its operations. Indeed, a part cannot pass its data to the next one if this last did not finish its process. The following sections first assess the number of cycles taken by matrices computation and adapt other parts of the system to meet this timing. The number of cycles will be the minimum one the pipeline has to take before taking a new input embedded vector. This way we make use of the full accelerator gain.

Matrix computation

As shown in Figure 5.6, the number of blocks N influences the number of used resources and the number of cycles needed for the projections of one input embedded vector. The DE10-Nano has enough available resources to support $N \leq 4$. As shown

in Figure 5.6b, the fastest processing time needed to compute $e^s W_q$, $e^s W_k$ and $e^s W_v$ with $N = 4$ is 18 cycles. As explained in the next section, the SA has to be reset, adding one more cycle to the matrix processing in this pipelining context. The rest of the operations will try to be under these 19 cycles to use the full gain of the accelerator.

Input reading

From the used Transformer parameters, an embedded vector is 512 bits long ($d_{model} \cdot D_S$). The memories have a 2-cycles read latency. It means that at each cycle one can update the read address and the result will be available two cycles afterwards. From these considerations and the condition presented in the previous paragraph we need the input memory reading to be complete in $\frac{512}{mem_width} + 1 \leq 19$ cycles. To respect this condition the memory width must thus be at least 31. To have some margin, a 64-bits wide memory was used. With this size, 9 cycles are needed to read the entire embedded vector.

Output writing

When the computations $e^s W_q$, $e^s W_v$ and $e^s W_v$ are done, the data will flow from the accelerating part to the output writing one. A write can be done in one cycle. For the same reasons as for the input reading, a memory width of 64 is chosen.

5.3.4.2 Logic

This sections aims at describing the logic that rules the pipelining. Six different blocks control the dataflow between the parts. They are presented hereunder.

Input sequence reading logic

This block updates the read input address at each clock cycle in order to get a part of the embedded vector outputted 2 cycles afterwards by the *Input Sequence Memory*. The output will be processed in the *seq_row_nxt register filling logic*.

seq_row_nxt register filling logic

As it can be observed in Figure 5.21, the embedded vector fed to the matrix computing

part is provided by the `seq_row` register. This register has to stay stationary for 19 cycles, i.e. the time it takes to process the data. However, pipelining should enable the filling of the next input while processing the current one. To do so, the `seq_row_nxt` buffer is filled. When the processing of the projections is done, the `seq_row_nxt` content is put in the `seq_row` register (in one cycle), and the loop can reiterate.

Reset logic

Every time the processing of an embedded vector is finished, the SA in the Linear Layer as well as its inputs are cleared to zero before taking the new inputs.

Q, K, V memory writing logic

This block works on the same principle as the one reading the inputs from memory. A buffer register (*add_Q, K, V_reg*) is put between the linear layer block and the memories. While the inputs of the next time step are processed in the linear layer, the output of the current one are written to memory, by blocks of 64 bits. The address is updated according to the *Input-Reuse Linear* pattern of section 3.1.5.2.

5.4 Final results

This section demonstrates the efficiency of our designed accelerator. For all experiments, the measured time is expressed for one inference. First, the timing and area measurements of the hardware-only execution are presented. Hardware-only means the time between the start of the computation of the projections and the end of the outputs writing in memory. Then, the final results of our co-design HW/SW are shown where HPS and FPGA work together.

Hardware-only execution

Figure 5.22 shows the measured area, the theoretical and measured latency of the projections in HW as a function of the variable N (see Section 5.2).

As a reminder we expect the number of PEs to linearly increase with N , as shown in Figure 5.6a. In Figure 5.22 one can confirm that the measured number of Adaptive

Logic Module (ALM) increases linearly with N . This is what we expect since the number of used ALMs is strongly linked to the number of PEs.

Figure 5.22 also presents the theoretical and measured latency of the hardware acceleration. The measured latency is defined as the time difference between the moment the HPS has finished to write the input sequence in memory and the moment the FPGA has finished the writing of the projections in memories. The theoretical one is computed using Equation 5.10. The behavior of the measured latency almost matches with the theoretical one. It also decreases following a $\frac{1}{N}$ curve when N linearly increases. The difference between the theory and the measurements can mainly be explained by an extra pipelining step required to write the last computed outputs in memory. Furthermore each pipelining step is one cycle longer since reset cycle is needed to reset all the registers of the SAs before the computations of the next pipelining step.

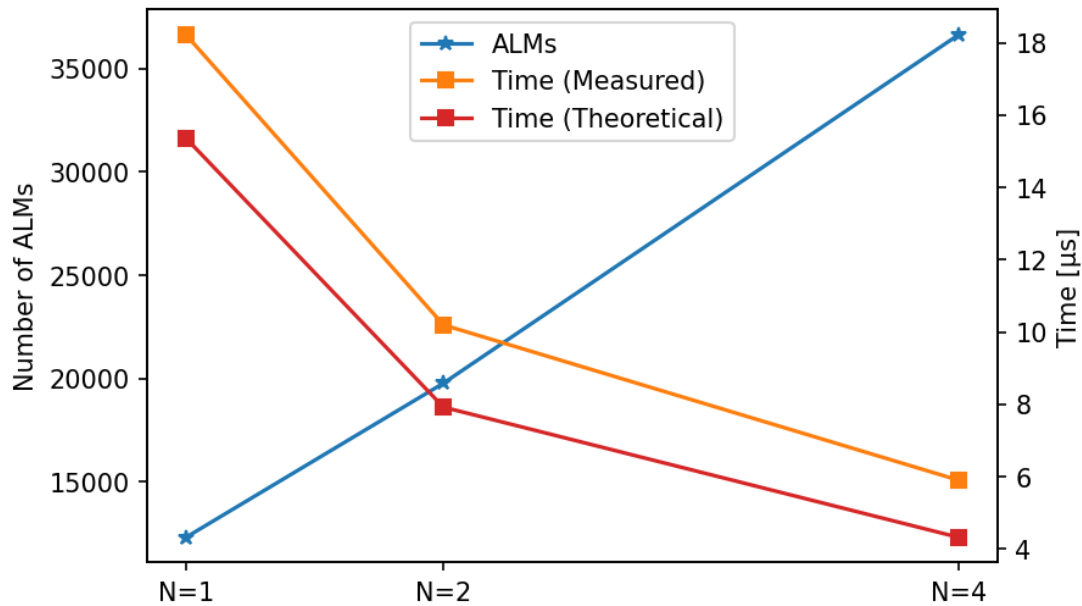


Figure 5.22: Area (Measured) and Latency (Measured and theoretical) depending on N . The theoretical curve comes from Section 5.2.

Hardware-Software execution

As a reminder (see Section 4.3.2 and Figure 4.4) the int8 MHA implementation has a 24.66 ms execution time without any hardware acceleration system. In this case, only the HPS processes the MHA with a clock frequency of 800 MHz.

Figure 5.23 shows the execution time of the MHA when using our hardware accelerator with $N = 4$. As a reminder, the HPS reads the outputs fed by the FPGA in the on-chip memories, stores them in a C array and uses those to compute the matrix products (*Softmax matmul/ Attention matmul*). In this configuration, the HPS still works with a 800 MHz clock frequency while the FPGA uses a 50 MHz one. The processing time drops to 13.29 ms, i.e. 54% of the previous result. **This implementation leads to a reduction of 46.1% of the total MHA inference time compared to the int8 implementation on the HPS.**

This result deserves further discussions. Indeed our hardware system accelerates the projections part of the MHA. Figure 5.23 points out that the accelerated projections use 13.28% (1.76ms) of the global processing time. In Figure 4.4, we can observe that the int8 projections implementation on HPS initially take 12.93 ms. **Our design is thus accelerating the projections computations by 86.39%.**

Furthermore, this result includes the inputs writing and the outputs reading. Without taking those steps into account, the accelerated projections only take 0.04% (5.9 μs) of the processing time. Considering only this computation part of the hardware projections, the inference time of the int8 software projection (12.93 ms) is accelerated by a factor 2191. We can conclude that the bottleneck of our accelerator is the read/write in memory. Indeed, the computational part of the accelerated projections only represents 0.33% of the total accelerated projections (computation + read/write) inference time.

One last thing to note is that the more portions of the Transformer architecture we accelerate with the FPGA, the less the read/write operations will have an impact on the acceleration timing. Indeed, the computational time would be closer (or higher) than the read/write time and the acceleration would be more relevant.

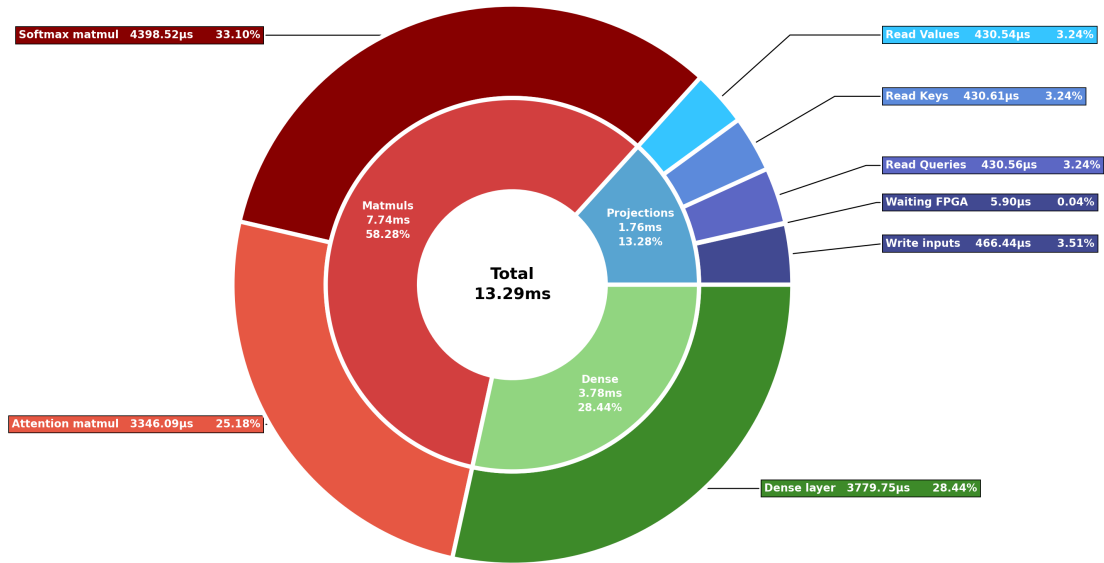


Figure 5.23: Breakdown of execution time when reading memory from FPGA before computing. *Waiting FPGA* denotes the time during which the FPGA computes and writes outputs to memory.

To overcome the reading/writing bottleneck, we try to avoid the outputs reading steps. The HPS could directly read the outputs in the on-chip memories fed by the FPGA when the matrix products are processed without storing them beforehand. The results of this experiment are shown in Figure 5.24. Unexpectedly this technique is less efficient than the original one. We can indeed observe that the whole process takes 19.60 ms which is 47.48% more than the implementation of Figure 5.23. It still reduces the inference time by 20.52% compared to the int8 implementation on the HPS. One plausible explanation for this result would be that for this case we have to drop the Altera read function. This last is specialized to interface the Avalon MM bridge and could thus be more efficient than a "classic" C memory read.

Finally, Figure 5.25 makes a comparison between all the implemented MHA versions that are presented throughout this work. It highlights the efficiency of using quantization as well as the gain performed by our hardware accelerator. As one can observe the final implementation (rightmost bar) reduces the total execution time by 89% compared to the initial FP one. One last thing to note is that the FPGA accelerator

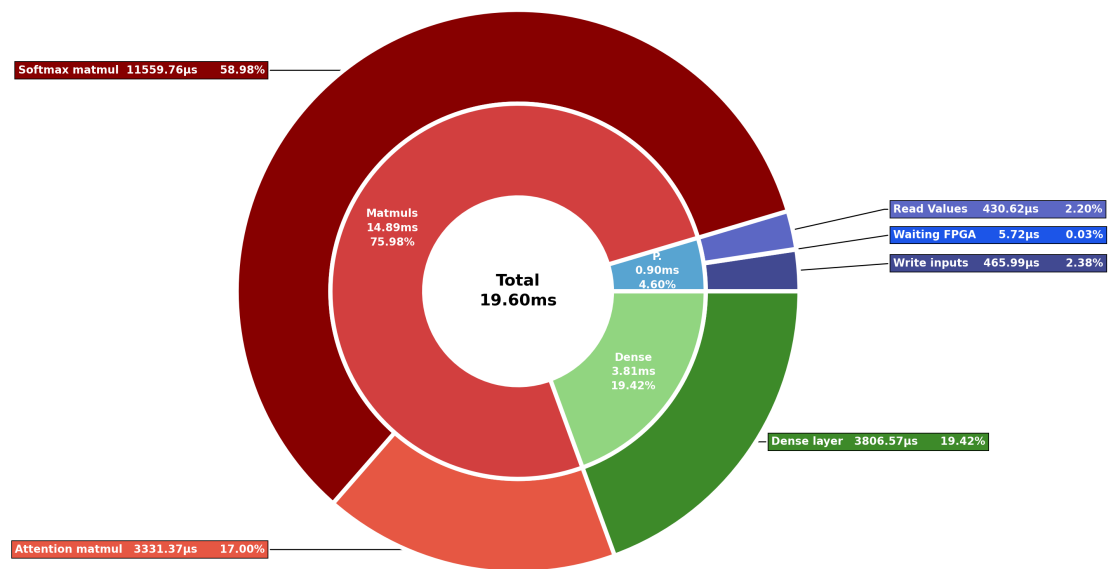


Figure 5.24: Breakdown of execution time when live - reading memory from FPGA during Softmax-matmul operation 'P.' stands for 'Projections

achieves satisfying performances even if its clock is 16 times slower than the HPS one.

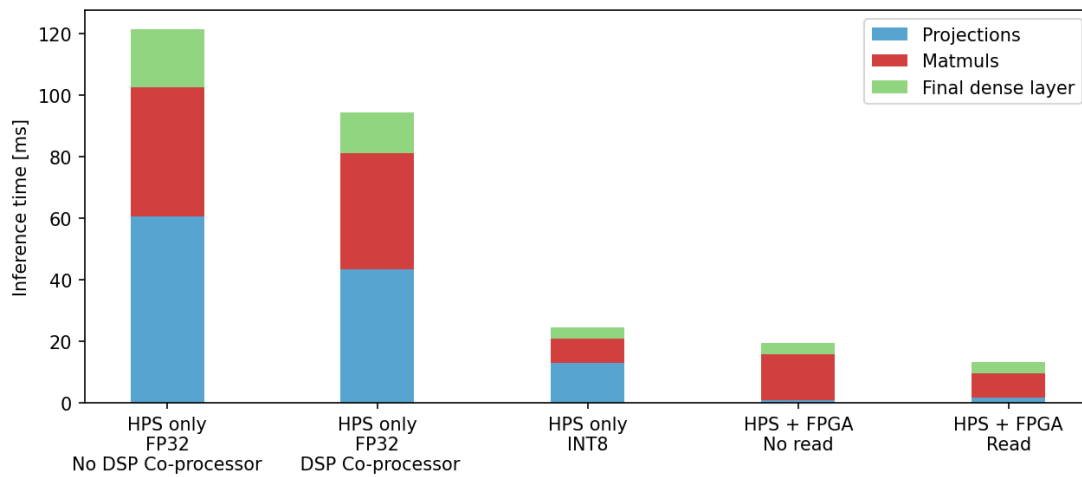


Figure 5.25: Comparison of the results for the different implementations. *Read* stands for the implementation where the HPS reads the projections computed by the FPGA before processing the matrix multiplications. *No Read* designates when memories are live-read during matrix computations on the HPS.

Conclusion

Embedded Machine Learning is becoming a major trend in IT research. This paradigm proposes to deploy ML models onto small devices to tackle tasks such as smart sensing or monitoring. In this context, an unified network capable of handling multiple tasks at once would be useful. This would indeed reduce the number of different architectures that have to be researched and optimized. Transformer seems as a good lead to such an architecture. As a consequence, this work proposes to deploy, optimize then accelerate a Transformer network onto a DE10-Nano. The major goal is to study the efficiency of an hardware accelerator on such architecture.

As a starting point we propose to study TFLM, a library designed to ease the deployment of ML architecure onto embedded devices. To first assess the usability of the library with the DE10-Nano, we propose the implementation of a CNN. This network is trained to reach an accuracy of 98.02% on the MNIST dataset. A quantized version of the network is compared to this baseline, achieving 98.01% of accuracy. Then the two versions are timed and compared. These timing results are surprising compared to what is expected between a FP and int8 implementation. The quantized version indeed never outperforms the FP one.

With the toy example working, Part II of the thesis addresses the mapping of a Transformer onto the DE10-Nano. An attempt is made to train the network but must be dropped for resources constrains reasons.

We then try to port the architecture using TFLM. However, missing operations implementations prevent this port from being successful. For this reason we go on

with the implementation of the MHA using C language and CMSIS-NN. The software port is successful and yields expected results. From a FP implementation taking 121 ms for one inference, we speed up the process by 79.73% to reach 24.7 ms using integer-only arithmetic. This result shows the advantage of using int8 operations and weights/activations as well as optimized libraries with the MHA.

To further accelerate the process, a SW/HW system is realized using the FPGA embedded on the DE10-Nano. Multiple acceleration schemes are presented, from which one is chosen to be implemented on the board. The strategy implies projecting $\mathbf{Q}, \mathbf{K}$ and $\mathbf{V}$ on the FPGA while processing the rest of the computations on the HPS. This implementation leads to a reduction of 46.1% of the inference time compared to the int8 implementation on the HPS. Compared to the initial FP32 computations the total inference time is now reduced by 89%. One thing to note is that the designed accelerator uses a lot of Area resources and the acceleration gain would be lower with a reduction of used ALMs.

The presented results show the interest of designing hardware accelerators for embedded Machine Learning. The Transformer is still a young architecture and deserves further researches, be it in embedded deployment, multi-task training, etc.

As when performing an inference, time is a scarce resource. Here are some research tracks we could not achieve in the year dedicated to this research:

- Continue de development of an efficient hardware pipelining, allowing the acceleration of more parts of the Transformer. One strategy could consist of designing one big accelerated SA used to project $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ as well as computing $\mathbf{Q}\mathbf{K}^T$. Combinations of the systems presented in Table 5.2 can be the starting point of such developments.
- Create a memory interface for the accelerator to access other weights than with which it is initialized. This would allow to load different models into the accelerator.
- Print the circuit onto an ASIC and study power consumption.

- Port the missing TFLM operations to deploy the complete Transformer on the board.
- A slack analysis of the hardware accelerator to increase the clock frequency of the FPGA.

Bibliography

- [1] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *arXiv:1706.03762 [cs]*, Dec. 5, 2017. arXiv: 1706.03762. [Online]. Available: <http://arxiv.org/abs/1706.03762> (visited on 12/08/2021).
- [2] "Chapter 16. systolic systems." (), [Online]. Available: <https://gyires.inf.unideb.hu/KMITT/a52/ch04.html> (visited on 08/04/2022).
- [3] "Building a recommendation system using CNN - v2." (), [Online]. Available: <https://kaggle.com/code/marlesson/building-a-recommendation-system-using-cnn-v2> (visited on 07/31/2022).
- [4] A. Dosovitskiy, L. Beyer, A. Kolesnikov, *et al.*, *An image is worth 16x16 words: Transformers for image recognition at scale*, Jun. 3, 2021. DOI: 10.48550/arXiv.2010.11929. arXiv: 2010.11929[cs]. [Online]. Available: <http://arxiv.org/abs/2010.11929> (visited on 07/31/2022).
- [5] R. Hu and A. Singh, "UniT: Multimodal multitask learning with a unified transformer," *arXiv:2102.10772 [cs]*, Aug. 18, 2021. arXiv: 2102.10772. [Online]. Available: <http://arxiv.org/abs/2102.10772> (visited on 02/14/2022).
- [6] S. Zhang, S. Roller, N. Goyal, *et al.*, *OPT: Open pre-trained transformer language models*, Jun. 21, 2022. arXiv: 2205.01068[cs]. [Online]. Available: <http://arxiv.org/abs/2205.01068> (visited on 08/18/2022).
- [7] "Global electricity usage of ICT network operators." Last Modified: 2021-07-01T14:32:41+00:00. (Jul. 23, 2019), [Online]. Available: <https://www.ericsson.com/en/reports-and-papers/research-papers/global-electricity-usage-of-ict-network-operators---an-extensive-data-set> (visited on 08/18/2022).

- [8] "Terasic - all FPGA boards - cyclone v - DE10-nano kit." (), [Online]. Available: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=1046> (visited on 07/31/2022).
- [9] A. Gallez and A. Vanden Clooster, *General · argallez/memoire_git_2*. [Online]. Available: https://github.com/argallez/memoire_git_2 (visited on 08/19/2022).
- [10] "Introduction to neural networks," Analytics Vidhya. (Jan. 31, 2022), [Online]. Available: <https://www.analyticsvidhya.com/blog/2022/01/introduction-to-neural-networks/> (visited on 08/14/2022).
- [11] J. Carson, N. K. Chakshu, I. Sazonov, and P. Nithiarasu, "Artificial intelligence approaches to predict coronary stenosis severity using non-invasive fractional flow reserve," *Proceedings of the Institution of Mechanical Engineers Part H Journal of Engineering in Medicine*, vol. 234, Aug. 3, 2020. DOI: 10.1177/0954411920946526.
- [12] P.-L. Pröve. "An introduction to different types of convolutions in deep learning," Medium. (Feb. 7, 2018), [Online]. Available: <https://towardsdatascience.com/types-of-convolutions-in-deep-learning-717013397f4d> (visited on 08/14/2022).
- [13] "CNN | introduction to pooling layer," GeeksforGeeks. Section: Machine Learning. (Aug. 5, 2019), [Online]. Available: <https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/> (visited on 08/14/2022).
- [14] M. Garifulla, J. Shin, C. Kim, *et al.*, "A case study of quantizing convolutional neural networks for fast disease diagnosis on portable medical devices," *Sensors*, vol. 22, no. 1, p. 219, Jan. 2022, Number: 1 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 1424-8220. DOI: 10.3390/s22010219. [Online]. Available: <https://www.mdpi.com/1424-8220/22/1/219> (visited on 08/15/2022).
- [15] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: Integer-only BERT quantization," arXiv, arXiv:2101.01321, Jun. 8, 2021, type: article. arXiv: 2101.01321[cs]. [Online]. Available: <http://arxiv.org/abs/2101.01321> (visited on 05/18/2022).

- [16] X. Dong, X. Zhu, and D. Ma, "Hardware implementation of softmax function based on piecewise LUT," in *2019 IEEE International Workshop on Future Computing (IWOFc)*, Dec. 2019, pp. 1–3. DOI: 10.1109/IWOFc48002.2019.9078446.
- [17] V. Bushaev. "How do we 'train' neural networks ?" Medium. (Oct. 22, 2018), [Online]. Available: <https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73> (visited on 07/29/2022).
- [18] "Setting the learning rate of your neural network.," Jeremy Jordan. (Mar. 2, 2018), [Online]. Available: <https://www.jeremyjordan.me/nn-learning-rate/> (visited on 07/29/2022).
- [19] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, Jan. 29, 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs]. [Online]. Available: <http://arxiv.org/abs/1412.6980> (visited on 08/13/2022).
- [20] 3Blue1Brown, *What is backpropagation really doing? | chapter 3, deep learning*, Nov. 3, 2017. [Online]. Available: <https://www.youtube.com/watch?v=Ilg3gGewQ5U> (visited on 07/29/2022).
- [21] B. J. McCaffrey and 08/18/2014. "Understanding neural network batch training: A tutorial -," Visual Studio Magazine. (), [Online]. Available: <https://visualstudiomagazine.com/articles/2014/08/01/batch-training.aspx> (visited on 07/29/2022).
- [22] "What is overfitting?" (Mar. 6, 2021), [Online]. Available: <https://www.ibm.com/cloud/learn/overfitting> (visited on 08/14/2022).
- [23] "Cyclone® v 5csea6 FPGA - product specifications," Intel. (), [Online]. Available: <https://www.intel.com/content/www/us/en/products/sku/210459/cyclone-v-5csea6-fpga/specifications.html> (visited on 07/31/2022).
- [24] "Cortex-a9." (), [Online]. Available: <https://developer.arm.com/Processors/Cortex-A9> (visited on 07/31/2022).
- [25] S. Jose, "Avalon memory-mapped interface specification," p. 88, 2007.

- [26] *TensorFlow lite for microcontrollers*, original-date: 2021-04-08T21:40:50Z, Aug. 13, 2022. [Online]. Available: <https://github.com/tensorflow/tflite-micro> (visited on 08/13/2022).
- [27] "Quickstart for android | TensorFlow lite," TensorFlow. (), [Online]. Available: <https://www.tensorflow.org/lite/android/quickstart> (visited on 08/07/2022).
- [28] "iOS quickstart | TensorFlow lite," TensorFlow. (), [Online]. Available: <https://www.tensorflow.org/lite/guide/ios> (visited on 08/07/2022).
- [29] "FlatBuffers: FlatBuffers." (), [Online]. Available: <https://google.github.io/flatbuffers/> (visited on 08/07/2022).
- [30] Google Developers, *Lightning talk: FlatBuffers*, Mar. 12, 2015. [Online]. Available: <https://www.youtube.com/watch?v=olmL1fUnQAQ> (visited on 08/07/2022).
- [31] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*, 1st edition. Beijing Boston: O'Reilly Media, Jan. 21, 2020, 501 pp., ISBN: 978-1-4920-5204-3.
- [32] *TensorFlow lite for microcontrollers*, original-date: 2021-04-08T21:40:50Z, Aug. 13, 2022. [Online]. Available: https://github.com/tensorflow/tflite-micro/blob/73c0be723ceb8c726d2a86fe79a18dfd6953bef3/tensorflow/lite/micro/tools/project_generation/create_tflm_tree.py (visited on 08/13/2022).
- [33] "Netron." (), [Online]. Available: <https://netron.app/> (visited on 08/13/2022).
- [34] "Post-training integer quantization | TensorFlow lite," TensorFlow. (), [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_integer_quant (visited on 08/07/2022).
- [35] "MNIST handwritten digit database, yann LeCun, corinna cortes and chris burges." (), [Online]. Available: <http://yann.lecun.com/exdb/mnist/> (visited on 08/07/2022).
- [36] A. Ltd. "Neon – arm®," Arm | The Architecture for the Digital World. (), [Online]. Available: <https://www.arm.com/technologies/neon> (visited on 08/07/2022).

- [37] "Models - hugging face." (), [Online]. Available: <https://huggingface.co/models> (visited on 07/31/2022).
- [38] "Google translate." (), [Online]. Available: <https://translate.google.com/> (visited on 08/13/2022).
- [39] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," p. 24,
- [40] p. gudikandula purnasai. "Recurrent neural networks and LSTM explained," Medium. (Mar. 27, 2019), [Online]. Available: <https://purnasaigudikandula.medium.com/recurrent-neural-networks-and-lstm-explained-7f51c7f6bbb9> (visited on 07/03/2022).
- [41] K. Cho, B. van Merriënboer, C. Gulcehre, *et al.*, *Learning phrase representations using RNN encoder-decoder for statistical machine translation*, version: 3, Sep. 2, 2014. arXiv: 1406.1078[cs, stat]. [Online]. Available: <http://arxiv.org/abs/1406.1078> (visited on 08/13/2022).
- [42] "Understanding of LSTM networks - GeeksforGeeks." (), [Online]. Available: <https://www.geeksforgeeks.org/understanding-of-lstm-networks/> (visited on 07/31/2022).
- [43] E. Guerriero. "Answer to "why does the transformer do better than RNN and LSTM in long-range context dependencies?"," Artificial Intelligence Stack Exchange. (Apr. 7, 2020), [Online]. Available: <https://ai.stackexchange.com/a/20084> (visited on 07/31/2022).
- [44] M. Ibrahim. "Top 3 transformer models 2021," The Techlife. (Feb. 11, 2022), [Online]. Available: <https://medium.com/the-techlife/top-3-transformer-models-2021-cd85b4e1cc1b> (visited on 07/31/2022).
- [45] "Transformer: A novel neural network architecture for language understanding," Google AI Blog. (), [Online]. Available: <http://ai.googleblog.com/2017/08/transformer-novel-neural-network.html> (visited on 08/08/2022).
- [46] "Deep learning #4: Why you need to start using embedding layers | by rutger ruizendaal | towards data science." (), [Online]. Available: <https://towardsdatascience.com/deep-learning-4-embedding-layers-f9a02d55ac12> (visited on 08/03/2022).

- [47] "15.6. subword embedding — dive into deep learning 1.0.0-alpha0 documentation." (), [Online]. Available: https://d2l.ai/chapter_natural-language-processing-pretraining/subword-embedding.html (visited on 08/03/2022).
- [48] Math of Intelligence, *Visual guide to transformer neural networks - (episode 1) position embeddings*, 2020. [Online]. Available: <https://www.youtube.com/watch?v=dichIcUZf0w> (visited on 08/03/2022).
- [49] M. Saeed. "A gentle introduction to positional encoding in transformer models, part 1," Machine Learning Mastery. (Jan. 31, 2022), [Online]. Available: <https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/> (visited on 08/03/2022).
- [50] D. Bahdanau, K. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, May 19, 2016. arXiv: 1409.0473[cs, stat]. [Online]. Available: <http://arxiv.org/abs/1409.0473> (visited on 08/03/2022).
- [51] N. Moritz, G. Wichern, T. Hori, and J. L. Roux, "All-in-one transformer: Unifying speech recognition, audio tagging, and event detection," in *Interspeech 2020*, ISCA, Oct. 25, 2020, pp. 3112–3116. DOI: 10.21437/Interspeech.2020-2757. [Online]. Available: https://www.isca-speech.org/archive/interspeech_2020/moritz20_interspeech.html (visited on 08/13/2022).
- [52] N. Vijayrania. "Different normalization layers in deep learning," Medium. (Mar. 14, 2021), [Online]. Available: <https://towardsdatascience.com/different-normalization-layers-in-deep-learning-1a7214ff71d6> (visited on 08/13/2022).
- [53] L. Mao. "Layer normalization explained," Lei Mao's Log Book. (May 31, 2019), [Online]. Available: <https://leimao.github.io/blog/Layer-Normalization/> (visited on 08/13/2022).
- [54] "Encoder models - hugging face course." (), [Online]. Available: <https://huggingface.co/course/chapter1/5> (visited on 07/28/2022).
- [55] "Decoder models - hugging face course." (), [Online]. Available: <https://huggingface.co/course/chapter1/6> (visited on 07/28/2022).

- [56] "Sequence-to-sequence models - hugging face course." (), [Online]. Available: <https://huggingface.co/course/chapter1/7> (visited on 07/28/2022).
- [57] Newton. "Computational complexity of self-attention in the transformer model," Stack Overflow. (Jan. 13, 2021), [Online]. Available: <https://stackoverflow.com/q/65703260/6904014> (visited on 08/03/2022).
- [58] T. J. Ham, S. J. Jung, S. Kim, *et al.*, "A³: Accelerating attention mechanisms in neural networks with approximation," *arXiv:2002.10941 [cs]*, Feb. 21, 2020. arXiv: 2002.10941. [Online]. Available: <http://arxiv.org/abs/2002.10941> (visited on 12/05/2021).
- [59] "STM8 8-bit microcontrollers (MCU) - STMicroelectronics." (), [Online]. Available: <https://www.st.com/en/microcontrollers-microprocessors/stm8-8-bit-mcus.html> (visited on 08/07/2022).
- [60] Y. Liu, M. Ott, N. Goyal, *et al.*, *RoBERTa: A robustly optimized BERT pre-training approach*, Jul. 26, 2019. DOI: 10.48550/arXiv.1907.11692. arXiv: 1907.11692[cs]. [Online]. Available: <http://arxiv.org/abs/1907.11692> (visited on 08/07/2022).
- [61] "GLUE benchmark." (), [Online]. Available: <https://gluebenchmark.com/> (visited on 08/07/2022).
- [62] H. Peng, S. Huang, T. Geng, *et al.*, "Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, ISSN: 1948-3287, Apr. 2021, pp. 142–148. DOI: 10.1109/ISQED51717.2021.9424344.
- [63] A. Burrello, M. Scherer, M. Zanghieri, F. Conti, and L. Benini, "A microcontroller is all you need: Enabling transformer execution on low-power IoT endnodes," in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, Aug. 2021, pp. 1–6. DOI: 10.1109/COINS51742.2021.9524173.
- [64] H. Wang, Z. Zhang, and S. Han, *SpAtten: Efficient sparse attention architecture with cascade token and head pruning*, Jan. 3, 2021. DOI: 10.48550/arXiv.2012.09852. arXiv: 2012.09852[cs]. [Online]. Available: <http://arxiv.org/abs/2012.09852> (visited on 08/07/2022).

- [65] "CSC/ECE 506 spring 2012/1c dm - PG_wiki." (), [Online]. Available: https://expertiza.csc.ncsu.edu/index.php/CSC/ECE_506_Spring_2012/1c_dm (visited on 08/04/2022).
- [66] "Google colaboratory." (), [Online]. Available: https://colab.research.google.com/?utm_source=scs-index (visited on 08/15/2022).
- [67] "A transformer chatbot tutorial with TensorFlow 2.0." (), [Online]. Available: <https://blog.tensorflow.org/2019/05/transformer-chatbot-tutorial-with-tensorflow-2.html> (visited on 08/04/2022).
- [68] "Europarl parallel corpus." (), [Online]. Available: <https://www.statmt.org/europarl/> (visited on 08/04/2022).
- [69] Y. Chai. "Counting the number of parameters in deep learning," Yekun's notepad. (Oct. 11, 2019), [Online]. Available: <https://cyk1337.github.io/notes/2019/10/11/NN/Counting-the-number-of-parameters-in-deep-learning/index.html> (visited on 08/04/2022).
- [70] K. E. Koech. "Cross-entropy loss function," Medium. (Jul. 16, 2022), [Online]. Available: <https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e> (visited on 08/07/2022).
- [71] "TensorFlow lite model analyzer," TensorFlow. (), [Online]. Available: https://www.tensorflow.org/lite/guide/model_analyzer (visited on 08/04/2022).
- [72] *TensorFlow lite for microcontrollers*, original-date: 2021-04-08T21:40:50Z, Aug. 13, 2022. [Online]. Available: https://github.com/tensorflow/tflite-micro/blob/73c0be723ceb8c726d2a86fe79a18dfd6953bef3/tensorflow/lite/micro/docs/porting_reference_ops.md (visited on 08/13/2022).
- [73] "Port reference SquaredDifference op from lite to micro · issue #1097 · tensorflow/tflite-micro." (), [Online]. Available: <https://github.com/tensorflow/tflite-micro/issues/1097> (visited on 08/13/2022).
- [74] "CMSIS NN software library." (), [Online]. Available: <https://www.keil.com/pack/doc/CMSIS/NN/html/index.html> (visited on 08/07/2022).
- [75] A. Ltd. "CMSIS," Arm Developer. (), [Online]. Available: <https://developer.arm.com/tools-and-software/embedded/cmsis> (visited on 08/07/2022).

- [76] "6.1.6. AXI bridge intel® FPGA IP," Intel. (), [Online]. Available: <https://www.intel.com/content/www/us/en/docs/programmable/683609/21-3/axi-bridge.html> (visited on 07/28/2022).

Appendix A

Tensorflow & Keras

As explained in chapter 1.1.2, a neural network has to be trained before it can be used. The basic workflow is the following: data is fed to the network that outputs a prediction, then an error is computed regarding to this output and the expected one, and the weights are updated according to a *backpropagation algorithm*. This approach is called *supervised learning* and is the one we used to train our own model. Many training frameworks are available in Python, the most famous ones being PyTorch and Tensorflow. Since our original idea was to use a derivative of Tensorflow for the software mapping, we chose to use this latter. This section goes over the fundamental ideas and concepts one has to know for correctly using this framework. Indeed, it allows the user to hide a lot of the complexity of machine learning training, but can also be very powerful for someone aware of the possibilities it can offer.

Firstly being a Python framework, Tensorflow has many derivatives in different languages (Javascript, C++, Java, Swift, Objective-C). The main workflow is the following: one has to train the model it wants to deploy in Python then convert it in a format that its objective target will understand. This can bring several challenges, as many tools that are used in Python are not available on the target platform, and sometimes difficult to convert from the computer to a specific platform. For example, operations that are implemented in the main Python library could not yet be available

on the platform one is trying to deploy on, or even not support some input types such as float numbers. This is explained in section 4.2.1.

Tensorflow embeds the *Keras* library, which is a collection of several well-known ML layers and functions. With this library, you can easily create ML models without having to code everything from scratch every time. It includes layers such as *Fully connected*, *Convolutional*, *Recurrent*, etc. This greatly simplifies the creation and training of models, especially for the simple ones as explained in section A.1.1.

A.1 Execution graph vs Eager execution

As explained in the previous section, one can train and infer ML models using only Python. But an ML algorithm can also be seen as a Directed Acyclic Graph (DAG) containing *Tensors* linked by *Operations (Ops)*. This view brings several advantages as the decomposition of the algorithm in several atomic operations, and an execution speedup compared to a *eager execution*, where the inference is read line by line by a Python interpreter. By default, Tensorflow 2.0 executes all of its operations in eager mode, but creates graphs when it saves a model. This allows other versions of Tensorflow (Javascript, C++, Java, etc.) to run the model, without resorting to a Python interpreter. ¹

In Tensorflow, the graph is automatically created by the autograph class, based on the user Python code using *tf.function* decorators. In a first step called *Tracing*, all functions are converted to graphs, which get executed every time this particular function is called again. A graph having fixed input/output (I/O) tensor shapes and dtypes, a function has to create a new graph every time it is called with different input/output characteristics than it has used to trace the graph.

A.1.1 Different model types

Tensorflow has the philosophy of making development gradually more complex as the developer wants to create more intricate architectures. Three model types are avail-

¹https://www.tensorflow.org/guide/intro_to_graphs

able, and it is important to understand what is possible with each of them to rightly choose the one suited for an application. The different possibilities are summarized on figure A.1 and are explained in the following sections. All of these possibilities can be used together, e.g. a sequential model can be a subpart of a functional one, or a layer defined using the subclassing model can be integrated in a sequential model, etc.

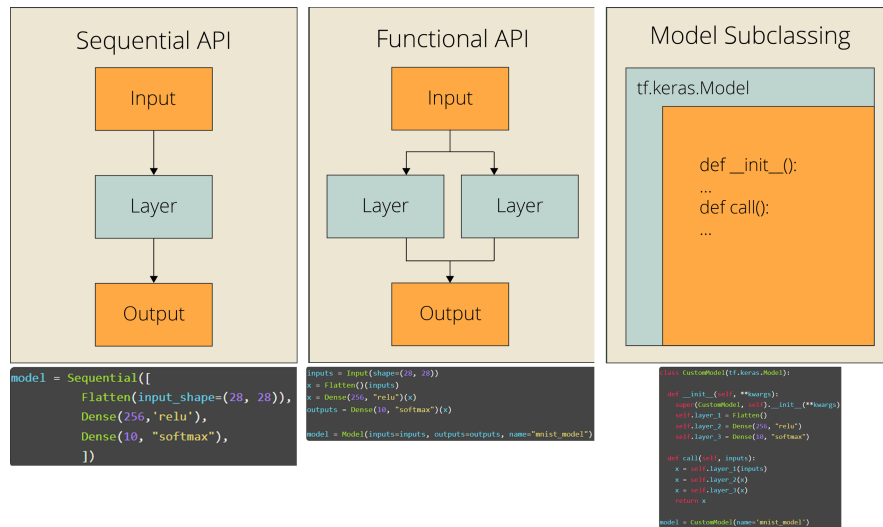


Figure A.1: Different Keras API with the amount of code needed for each

Sequential model Sequential models are the most simple ones. They can take only one input and produce one output for each layer. This kind of model is simple but can already fit a vast number of applications such as CNNs, multilayer perceptrons and autoencoders. It has the advantage of a really fast development cycle, as the code is really simple to implement, as well as having a clear interface to hyperparameters.

Functional model The functional model treats models and layers as functions (hence the name) and is more flexible than the sequential API. It can create models and layers with multiple I/Os, at the expense of a bit more complexity than the sequential one. It appeared later than the other APIs and documentation on the web is thus more scarce. Yet it is a powerful compromise between sequential and

subclassed models. As shown on figure A.1, one has to specify the relations between the layers, inputs and outputs.

Subclasses model Model subclassing is the most intricate way of designing Tensorflow models. It asks to manually specify the layers in a Python class that extends the *Keras.Model* one, and specify the call flow in a function. With this technique, it is also possible to specify different behaviors during training or inference time.

After designing the model, it has to be *compiled* with a loss (MSE, crossentropy, etc.), an optimizer (SGD, Adam, etc.) and optional metrics (to evaluate performances through training). Once all these steps are complete, one can train its model. The followed process for this work is presented in section 4.1.

