POLITECNICO DI TORINO

Master of Science's Degree in Electronic Engineering
Embedded Systems

Master of Science's Degree Thesis

In-depth study of state-of-the-art Transformer implementations for FPGAs

Supervisors

Prof. Mario CASU

Prof. Luciano LAVAGNO

Prof. Mihail LAZARESCU

Candidate

Carlos Esteban VERGARA PUCCINI

October 2024

Summary

Transformers, a groundbreaking innovation in artificial intelligence, have significantly impacted both natural language processing (NLP) and computer vision, with models such as ViT, GPT, and BERT leading the way. These models have demonstrated exceptional performance across a variety of tasks, including machine translation, text summarization, and sentiment analysis. Their success is largely due to the use of self-attention layers, which enable them to process large sequences of data efficiently while supporting parallelized training. However, deploying Transformer-based architectures on smaller platforms poses substantial challenges, particularly due to their high computational and resource requirements, especially in real-time and resource-constrained environments.

To address these challenges, Field-Programmable Gate Arrays (FPGAs) have emerged as a promising solution. FPGAs are versatile, reconfigurable integrated circuits that offer a unique combination of flexibility, parallelism, and energy efficiency, making them ideal for specialized applications such as deep learning inference. Nevertheless, implementing Transformers on FPGAs is a complex task that requires a diverse range of expertise. It involves several steps, from optimizing the neural network for quantized inference to designing an accelerator architecture that maximizes the efficient use of FPGA resources.

This work provides a comprehensive review of the latest developments in transformer accelerators using FPGAs. It examines the most recent advancements in FPGA-based architectures tailored to accelerate transformer models, delving into various implementation strategies and optimization techniques.

Acknowledgements

Before proceeding with the Thesis work, I would like to thanks to all the people that help me get through my master degree program.

I would like to thank my family in Colombia and in Italy that gave me all their support specially to aunt Maria Pia that took me in as another grandson, my cousin Alessandro that was always there to lend me a hand whenever I needed it, my uncles Marco and Juancho that came to visit me and filled me with trust, to my friend Youssef that joined me on holidays when everyone else was away, to my parents that called to check up on me every week, and to my girlfriend Cristina, if it were not for her I would have not take the risk to jump into this adventure.

I would like to give special thanks to Teodoro, Usman, and Roberto for their help during this thesis, since they were always kind, diligent, and supportive whenever I had a doubt.

Thanks to all of you without you I wouldn't have reached this far.

Sincerely
Carlos Vergara

Table of Contents

List of Tables vii

List of Figures viii

Acronyms xi

1 Introduction 1
- 1.1 Machine learning overview 1
  - 1.1.1 Determining Type of Training Experience 3
  - 1.1.2 Determine Target Function 3
  - 1.1.3 Determine Representation of Learned Function 3
  - 1.1.4 Determine Learning Algorithm 3
- 1.2 Neural Networks overview 3
  - 1.2.1 Sigmoid Neuron 3
  - 1.2.2 Training 5
- 1.3 Basic DNN models 6
  - 1.3.1 Convolutional Neural Network 6
  - 1.3.2 Recurrent Neural Network 7
  - 1.3.3 Common Layers in DNNs 8

2 Transformer architecture 10
- 2.1 Architecture Overview 10
  - 2.1.1 Self-Attention 12
  - 2.1.2 Multi-head attention layer 12
  - 2.1.3 Positional embedding 13
  - 2.1.4 Feed Forward layer 15
- 2.2 Transformer based models 15
  - 2.2.1 BERT 15
  - 2.2.2 GPT 16
  - 2.2.3 ViT 16
  - 2.2.4 Transformer model size discussion 17

3 Model Optimization 19
3.1 Quantization overview 19
3.1.1 Quantization Classification 20
3.2 Model Pruning overview 21
3.2.1 Unstructured Pruning 21
3.2.2 Structured Pruning 22
3.2.3 Semi-Structured Pruning 22
3.3 Knowledge Distillation 23

4 FPGAs and Deep Learning Inference 24
4.1 FPGA overview 24
4.1.1 FPGA based SOCs and SOMs 25
4.2 FPGA Design Flow 25

5 FPGA accelerators for Transformer model inferencing 28
5.1 Fitting Transformer into an accelerator 28
5.1.1 Quantized Transformer Accelerators 28
5.1.2 Pruned Transformer Accelerators 29
5.2 Modelling Transformer operations in FPGA 30
5.2.1 Matrix Multiplication 30
5.2.2 Softmax 35
5.2.3 LayerNorm 43
5.3 Discussion 45
5.4 Conclusions 46

Bibliography 47

List of Tables

2.1 Comparison of Model Sizes: ViT, BERT, GPT, and Original Transformer . . . 17

5.1 Comparison between Quantized or integer arithmetic accelerator . . 46
5.2 Comparison between accelerators that exploit sparsity patterns . . . 46

List of Figures

1.1 Summary of design choices from checkers example [1] 2
1.2 Single perceptron 4
1.3 Feed-forward architecture 5
1.4 Comparison between a Shallow Artificial Neural Network and a Deep Neural Network 6
1.5 A visual representation of a convolutional layer. The centre element of the kernel is placed over the input vector, of which is then calculated and replaced with a weighted sum of itself and any nearby pixels [7]. 7
1.6 An example of a simple recurrent network [8]. 8

2.1 Transformer - model architecture [12]. 11
2.2 Scaled dot-product Attention [12]. 13
2.3 MultiHead Attention layer block diagram [12]. 14
2.4 BERT architecture [14]. 16
2.5 ViT model overview [16]. 18

3.1 Knowledge Distillation process [27]. 23

4.1 FPGA Block Diagram [30]. 25
4.2 Versal Soc Module components [31] 26

5.1 [45] Matrix Multiplication architecture 31
5.2 8-bit multiplier architecture [51], where A and B are the operands multiplied by C. RC_A and RC_B are added to correct the result of the initial multiplication, RC_A is 1 when C is signed and RC_B is $-B_{MSB} \cdot 2^8 \cdot C$. 31

5.3 4-bit multiplier architecture [51], where A, B, C, D are the operands multiplied by E. the shifters at the beginning truncate the inputs to 4-bit representation and at the output are restored to their original size. Much like with the B · C correction for the 8-bit multiplier RC_B, RC_C, and RC_D for the 4-bit are calculated obtained in the same manner. As for RC_A is $(-A_{\text{MSB}} + A_{\text{MSB-1}}) \cdot 2^3 \cdot E$. . . . 32
5.4 Processing element for matrix multiplication [42]. When multiplying 8-bit by 1-bit values the PE operates 8 multiplications concurrently accumulating them at the end. Instead, when operating 8-bit × 8-bit mode, the PE can only performs one single multiplication at a time. 33
5.5 Binary transformer Matrix multiplication unit of the proposed accelerator [44] 34
5.6 Matrix multiplication process for a binary Transformer example [44]. 35
5.7 Processing Element capable of handling 8-bit by 4-bit and 8-bit by 8-bit [38]. 36
5.8 Piecewise Softmax architecture [55] 37
5.9 Architecture of the softmax operator. from [56], from Algorithm 1 "$q_1 = b/S_{pe}$, $q_2 = c/aS_{pe}^2$, $q_3 = \ln 2/S_{ec}$, and $q_4 = -1/q_3c$, where S_e is the scaling factor of the exponential computation input (relative to q_e), S_{pe} is the scaling factor of the polynomial approximation input (relative to q_{pe}) and a, b, c are the coefficient of the second-order polynomial that approximates the function, i.e., $a(x + b)2 + c$. Therefore, $q_{1,2,3,4}$ can be computed at design time and provided as constant values". 38
5.10 Architecture of the scalable softmax unit [46]. 40
5.11 Softmax implementation with each stage datapath highlighted [33]. 41
5.12 Base two exponential function architecture [58] 41
5.13 Base two exponential natural logarithm architecture [58] 42
5.14 Binary softmax architecture [44]. 43
5.15 LayerNorm architecture with integer square root[56] 44
5.16 LayerNorm architecture and schedule [33]. 45

Acronyms

AI
: artificial intelligence

ML
: Machine Learning

MLP
: Multi Layered Perceptron

DNN
: Deep Neural Network

CNN
: Convolutional Neural Network

RNN
: Recurrent Neural Network

NLP
: Natural Language Processing

MHA
: Multi Head Attention

FFN
: Feed Forward Network

QAT
: Quantization Aware Training

PTQ
: Post-Training Quantization

FPGA
: Field Programable Gate Array

LUT
: Look Up Table

DSP
: Digital Signal Processor

FF
: Flip Flop

CLB
: Complex Logic Block

RAM
: Random Access Memory

FIFO
: First in First out Memory Buffer

SOC
: M System On Chip/Module

RTL
: Register Transfer Level

Chapter 1

Introduction

This chapter is designed to offer a comprehensive overview of essential machine learning concepts, serving as a foundational introduction for the following chapters. It begins by outlining the core principles of machine learning, providing context for its various applications and significance in modern technology. The focus then shifts to deep neural networks, a critical area within machine learning, where the chapter delves into their architecture, functionality. Finally, the chapter concludes with a detailed summary of the most basic and commonly used layers in deep learning models, explaining their roles and how they contribute to the overall performance of these models.

1.1 Machine learning overview

To start explaining what is Machine Learning (ML) it is first necessary to have a sufficiently clear idea of the word learning. Many authors have provided different definitions regarding this concept, and even though definitions can be related each author develops a completely unique philosophical framework to conceptualize machine learning.

Fort this work the definition and framework used in [1]; learning is defined as:

- *A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E*

Given this definition the problem of defining learning is reduced to three terms, Experience **E**, Task **T**, and Performance **P**. These Terms can be derived from a concrete problem or context, to illustrate this [1] proposes the following example:

- *A computer program that learns to play checkers might improve its performance*

as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself [1].

Given the previous formulation, **T**, **P**, and **E**, can be established as follows:

- Task **T**: playing checkers;
- Performance measure **P**: percent of games won against opponents;
- Training experience **E**: playing practice games against itself.

This definition of learning is sufficiently broad and clear enough that it is possible to formulate many problems as learning problems, where the solution to these is a computer program that satisfies the definition of learning itself. To design the aforementioned computer programs, [1] proposes the strategy shown in Figure 1.1 contextualized for checkers example.

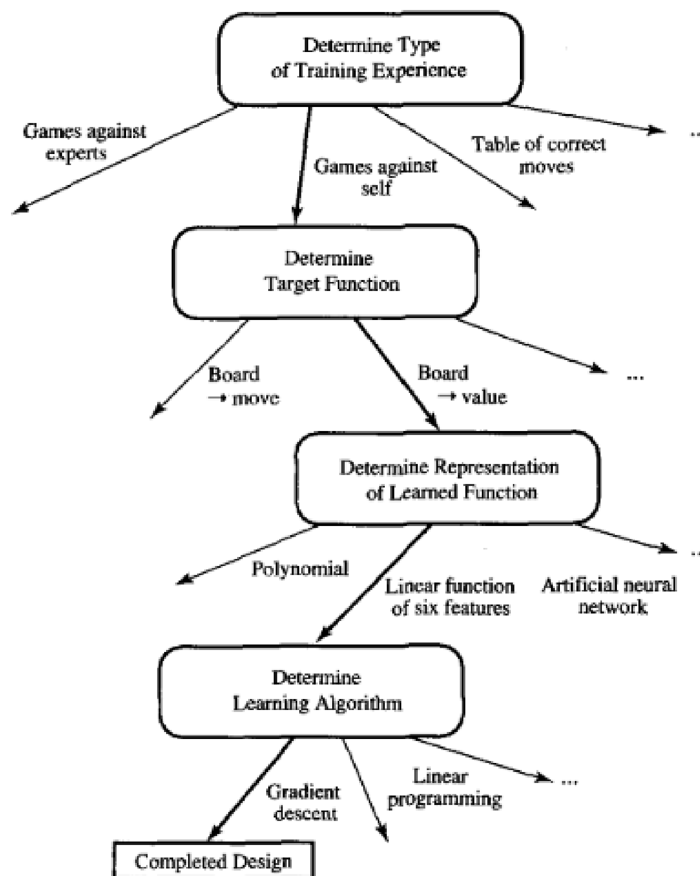


Figure 1.1: Summary of design choices from checkers example [1]

1.1.1 Determining Type of Training Experience

The first step, consist of modeling how data for training is acquired to train the machine learning program: whether it is generated by the designer or taken from a dataset, and if data is going to be labeled or not. This step conditions the overall result of the program and its success depends on having training data that represents correctly the distribution of the overall data population.

1.1.2 Determine Target Function

The second step, is simply defining the output of the program, for example continuing with the checkers example the output of the model could be the next movement to perform or scoring the board state. This step is important since it heavily conditions the implementation of the prediction algorithm.

1.1.3 Determine Representation of Learned Function

This step consist of choosing the mathematical model or algorithm that is going to be trained, for example it could be a simple regression, a neural network, a decision tree, etc. It is important to mention that for the sake of brevity learned function is often referred simply as model.

1.1.4 Determine Learning Algorithm

Finally once all the steps have been defined, a strategy to adjust the trainable parameters of the model must be defined, this strategy can be for example a mean square error adjustment, or a gradient descent algorithm to name a few.

1.2 Neural Networks overview

The remaining part of this introduction focuses on one of the most wide-spread representation models, the Neural Networks and it aims to explain Deep Neural Networks **DNN** by first presenting the most simple neural network i.e. a single Sigmoid Neuron to then explaining how they are trained using back-propagation, and finally present some of the most common Neural Networks architectures and layers.

1.2.1 Sigmoid Neuron

The Sigmoid Neuron is inspired by its predecessor the Perceptron that was developed in the 1950s and 1960s by the scientist Frank Rosenblatt. It consist of a

mathematical model that follows the formula shown in equation 1.1 [2] and are graphically represented as shown in Figure 1.2.

$$output = \frac{1}{1 + e^{(\sum_j w_j x_j)+b}} \tag{1.1}$$

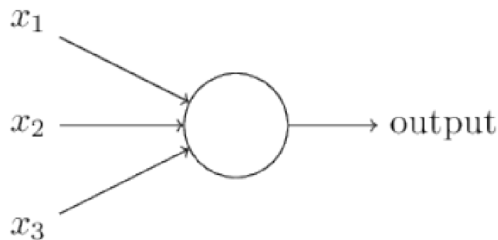


Figure 1.2: Single perceptron

Equation 1.1 can be divided into three parts: the first being the input of the model denoted in the equation 1.1 as x; the second part being the weighted sum, where w and b are the parameters that the model has to learn; and the third part is the activation function that in this particular case is just a logistic function also known as sigmoid function, thus the naming of the neuron. It is worth noting that in this example [2] all numeric values inside the model ranges from 0 to 1.

The important aspect about Neurons is that they can be connected in many ways, the most simple way to connect neurons is in a feed-forward way meaning that neurons are grouped in layers that serve as input to subsequent layers, without backward loops. This way each subsequent layer performs a more complex computation, giving the overall network the capacity to represent any kind of mathematical function, as shown in Figure 1.3. This feature proves useful in a great number of tasks where mathematical models are required to make predictions based on data, or classify said data. Examples of these tasks could be Image classification and Text translation, to name a few.

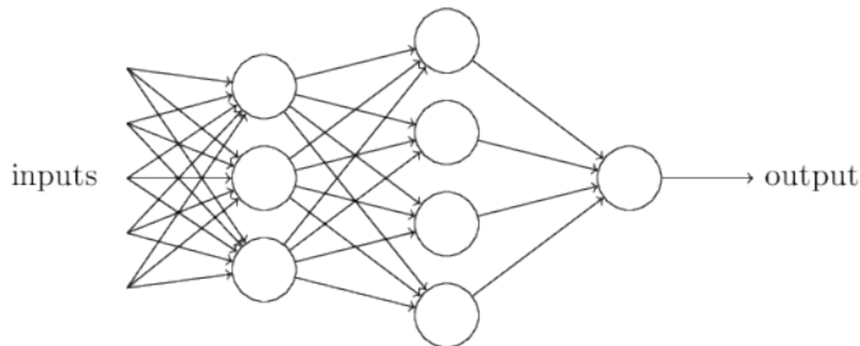


Figure 1.3: Feed-forward architecture

Once the number of inputs, layers, and outputs is defined, the next step to finish the model is defining the weights and biases, this is done by training the network adjusting the values until the model is sufficiently accurate.

1.2.2 Training

When referring to training a neural network, it is intended as executing the learning algorithm that adjusts the neural networks parameters. This algorithm basically consist of running inferences i.e. producing outputs with suitable data and then, scoring the accuracy of the output in some capacity to later, based on these results modify the values of the parameters accordingly. This is summarized in the following enumeration [3]:

1. Iterate over a dataset of inputs.
2. Process input through the network.
3. Compute the loss (how far is the output from being correct).
4. Propagate gradients back into the network's parameters.
5. Update the weights of the network, typically using a simple update rule shown in equation 1.2.

$$weight = weight \text{ - } learning_rate \text{ * } gradient \tag{1.2}$$

It is worth mentioning that, due to neural network computational complexity, the majority of software tools use back propagation and gradient decent methods to adjust the network parameters.

1.3 Basic DNN models

Having introduced the most basic Neural Network the feed forward layer, is time to expose some of the most common neural networks and also some of the functions used along with them.

To start off, the main difference between a simple neural network and a deep neural networks can be reduced to the higher complexity of the latter with respect to the former. This complexity is measured in the form of deepness of a given network where deepness generally means the amount of neuron layers between the input and output of the model [4]. These layers are always regarded as hidden layers whereas the input and output are represented as layers themselves, when representing the model graphically. Figure 1.4 shows a comparison between two neural networks that differ on depth, the one on the left is considered a shallow neural network, and the one in the right can be regarded as a deep neural network.

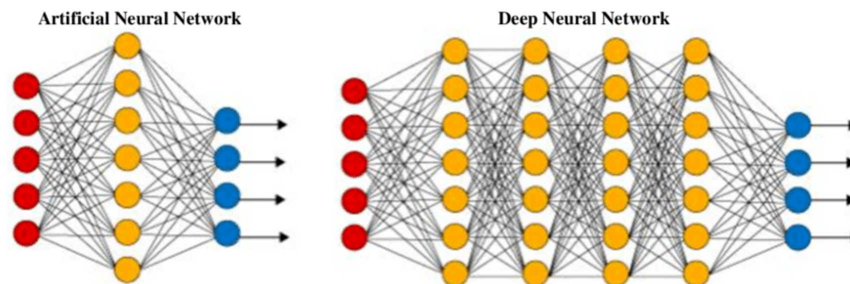


Figure 1.4: Comparison between a Shallow Artificial Neural Network and a Deep Neural Network

The reason why this terminology became prevalent is due to the increase over the years of computational power that allowed the one to three layer models to grow substantially to models of even more than 50 layers of depth [5].

Having stated what deepness means in the context of neural networks the following sections show other types of neural networks used together with Feed Forward networks.

1.3.1 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a specialized type of neural network designed primarily for processing structured grid data, such as images. CNNs are highly effective in tasks like image recognition, object detection, and image segmentation due to their ability to automatically and adaptively learn spatial hierarchies of features from input images. Its key components are:

- Convolutional Layers: These layers apply kernels i.e. performs a 2D or 3D convolution to the input data, helping detect patterns in the data, Figure 1.5 Shows a graphic example of how a convolution layer performs 2D convolution;

- Pooling Layers: Pooling, often MaxPooling, is used to reduce the spatial dimensions of the data, which reduces computation and helps to make the detected features more invariant to scale and orientation, in case of MaxPooling it down-samples data by picking the maximum value over a window [6].

- Fully Connected Layers or Feed Forward Layers: After several convolutional and pooling layers, the data is typically flattened and passed through one or more fully connected layers, which make the final classification or prediction.

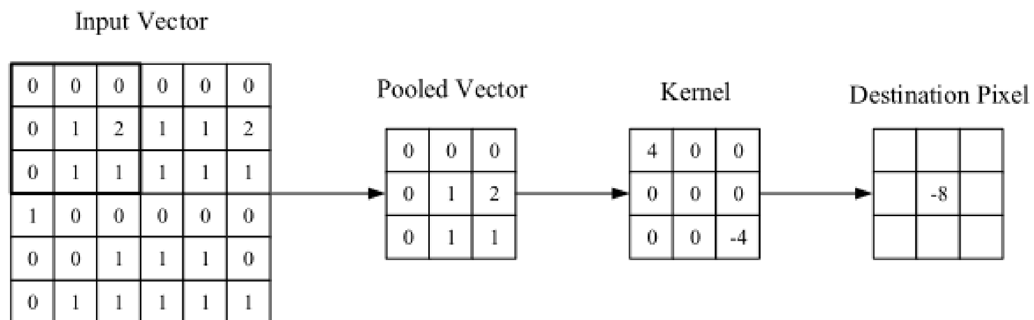


Figure 1.5: A visual representation of a convolutional layer. The centre element of the kernel is placed over the input vector, of which is then calculated and replaced with a weighted sum of itself and any nearby pixels [7].

CNNs excel in capturing spatial features and are the backbone of most modern computer vision systems.

1.3.2 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a class of neural networks well-suited for sequential data, such as time series, language data, or any data where the order of inputs is important. Unlike feed forward neural networks, RNNs have connections that form directed cycles as Figure 1.6 shows. This mechanism allows RNNs maintain a 'memory' of previous inputs, which is crucial for tasks like language modeling, speech recognition, and machine translation.

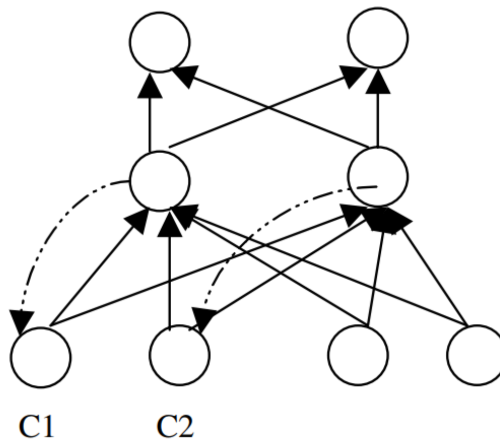


Figure 1.6: An example of a simple recurrent network [8].

1.3.3 Common Layers in DNNs

Having exposed the most common neural network architectures, It is important to mention that DNNs models can be designed a mixture of the mentioned above by composing them with layers that perform different operations, so a model can be composed of layers that perform convolution, layers that are recurrent, and others that are just feed forward. Aside from these layers there are other kinds of layers that help model the non linearity of the model such as:

Activation Layers

Activation layers apply a function to the output of a layer, introducing non-linearity into the model, which is crucial for the network's ability to learn complex patterns. Without activation functions, the network would only be able to learn linear relationships, severely limiting its power.

Some common activation functions are:

- ReLU (Rectified Linear Unit): ReLU is the most widely used activation function in deep learning, defined as (1.3). It introduces non-linearity while being computationally efficient and facilitates training of models.

$$f(x) = max(0, x) \tag{1.3}$$

- Sigmoid: The sigmoid function maps inputs to a range between 0 and 1, making it useful for binary classification problems. However, it is less efficient for training than ReLu gradients.

- Tanh (Hyperbolic Tangent): Tanh is similar to the sigmoid function but maps inputs to a range between -1 and 1. It is also not as efficient for training as ReLu but is preferred over sigmoid in some cases because of its centered output.

SoftMax Layer

The SoftMax layer is typically used as the final layer in a neural network designed for multi-class classification problems. It converts the raw output scores (logits) from the previous layer into probabilities, which represent the likelihood of each class.

The softmax function is only one of infinitely many functions that map non-normalized scores onto probability vectors [9]. It applies the equation (1.4) returning the aforementioned probability vector.

$$Softmask(X) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{1.4}$$

Normalization Layers

Normalization layers are used to stabilize and accelerate the training process of deep neural networks by standardizing the inputs to each layer. They help prevent issues like internal covariate shift, where the distribution of inputs to a layer changes during training.

Common Normalization Techniques:

- Layer Normalization: normalizes using (1.5) the inputs across the features for each data point independently [10], where $E[x]$ is the mean across the input dimension, and $\mathrm{Var}[x]$ is the variance making it more suitable for recurrent networks.

$$y = \frac{x - E[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta \tag{1.5}$$

- Batch Normalization: Batch normalization normalizes the output of a layer by applying (1.5) with the difference that the mean and the variance are parameters set in training and calculated across each mini batch of data, instead of the input dimension [11].

Chapter 2

Transformer architecture

After introducing the concepts of Machine Learning and Neural Networks, This current chapter will shift its focus to the Transformer architecture. Introduced in the groundbreaking paper "Attention Is All You Need" by Vaswani et al [12]. in 2017, represents a significant leap forward in the field of deep learning, particularly in natural language processing (NLP). Unlike traditional models like Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs), which rely on sequential data processing or convolutions, the Transformer architecture is based entirely on a mechanism called self-attention. This innovation allows Transformers to handle long-range dependencies in data more effectively and efficiently, making them highly powerful for tasks involving sequential information, such as language translation, text generation, and even beyond NLP, in areas like image processing and time-series analysis.

2.1 Architecture Overview

Figure 2.1 shows the original Transformer [12], it displays a block diagram with each of the transformers layers, notice that there are two blocks encapsulating Feed Forward, Multi-Head Attention, and Add & Norm layers, these are named Encoder and Decoder, where the Encoder feeds it's output to the Decoder and the Decoder gives it's output to the final linear layer that is a simple feed forward network as explained in the previous chapter. The purpose of this design is that it was meant to operate on sequential data, since the Transformer was mainly developed for text translation task. The way it operates is that the positional encoders embed the sequential information of the input, and then the encoder extracts what part of the sequence are the most important and derive a context from it, then sends this context to the decoder That was fed with the actual output sequence and produces the next token. It is important to state that even though the originally

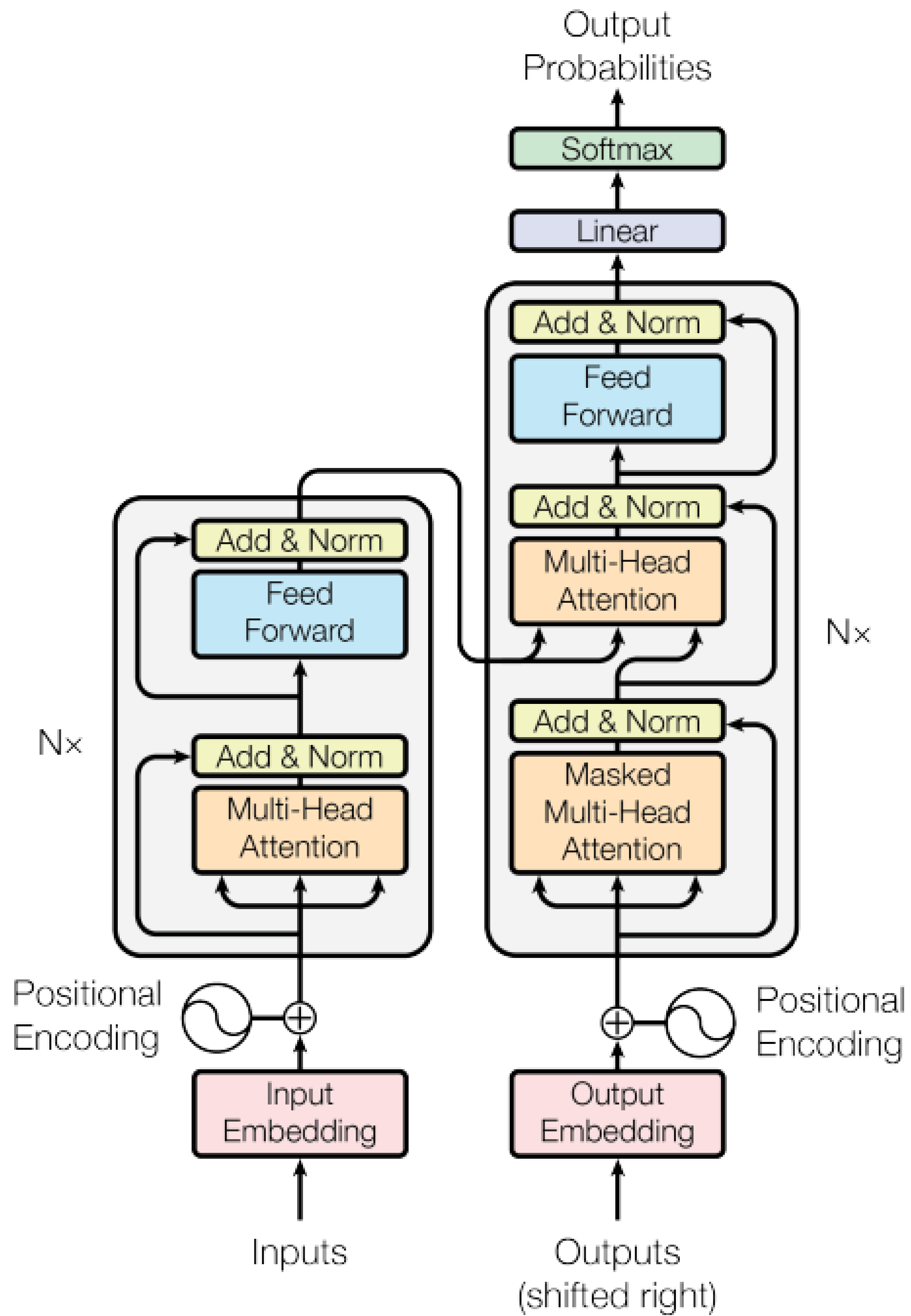


Figure 2.1: Transformer - model architecture [12].

the Transformer architecture was designed in this manner many transformer model only use Encoders or Decoders given that, by themselves alone, have been proven

to be effective for many other task such as sentiment analysis and text generation.

2.1.1 Self-Attention

Self Attention could be considered the central part of any transformer model since is in charge of finding relationships on data. In the case of the transformer it can be illustrated as Figure 2.2 shows. It operates by first computing the matrix multiplication between the query **Q** and the keys **K**, which represents the similarity between the elements. The resulting scores are then usually scaled down by the square root of the number of columns the **K** input, this is due to the softmax layer that comes after since it could lead extremely small gradients, which can destabilize training. After scaling, the scores are passed through a SoftMax function to obtain the attention matrix i.e. how much the keys and the values actually match, which then is embedded into the value input **V** using another matrix multiplication, this can be all summarized in the formula (2.1) or represented visually with Figure 2.2. The resulting output, captures the relevant information from the entire sequence, allowing the model to focus on the most important parts of the input when making predictions.

$$\text{Attention}(Q, K, V) = \text{Softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

Where d_k is the dimension of each token that belongs to the sequence and Q, K, V are matrices of equal dimensions.

2.1.2 Multi-head attention layer

Multi-Head Attention is an extension of the self-attention mechanism presented in the previous section. Used in the Transformer architecture, instead of relying on a single attention operation, Multi-Head Attention splits the inputs data into multiple smaller subsets that are then processed in parallel applying the Scale Dot-Product function this can be illustrated as Figure 2.3 where each block of scaled dot product attention is referred as "head". It is important to signal that the scale factor is now divided by the square root of the number of heads since the input to each individual head is now split equally across heads. One explanation to these choice rather than doing a single head that process all data together is that the model is then able to capture different types of relationships and patterns within the data simultaneously, letting the Transformer process data in parallel rendering Multi-Head Attention a crucial component in the Transformer's ability to handle complex tasks like language understanding and sequence modeling. The behaviour he this layer can be synthesized in the Formulas (2.2) and (2.3).

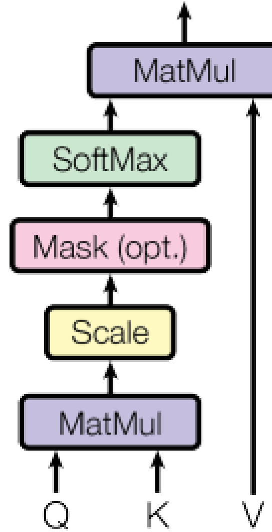


Figure 2.2: Scaled dot-product Attention [12].

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_i, ..., \text{head}_n)W^O \tag{2.2}$$

$$\text{head} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.3}$$

It is important to point out that all operations shown correspond to matrix operations and that splitting are done, in principle, through matrix multiplications where Q, K, V are matrices of $\Re^{s \times d}$ and W_i^K, W_i^Q, W_i^V are the weights of the initial linear layers in Figure 2.3 that each have dimensions of $\Re^{d \times \frac{d}{h}}$ thus the input to each attention head shall be of $\Re^{s \times \frac{d}{h}}$.

2.1.3 Positional embedding

Positional embedding is also one of the most innovative components of the Transformer architecture, They address the understanding of sequence order. Since Transformers process input data in parallel rather than sequentially as seen by the multi head attention layer, there must be a layer that takes into account the position of each token in a sequence. Positional embedding is added to the input token embedding, in the original Transformer they are implemented as shown in

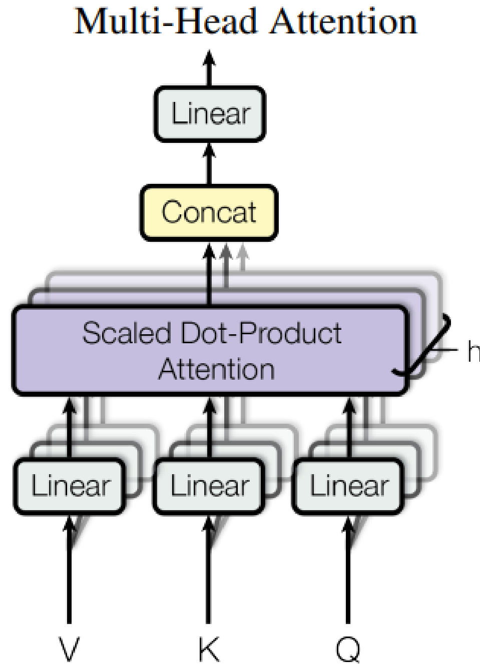


Figure 2.3: MultiHead Attention layer block diagram [12].

Formula (2.4), this way the position in a sequence of a particular Token is embedded into the data itself, hence the name, This gives the chance to the subsequent layers to take into account the order of a sequence. The understanding of sequence is especially crucial in tasks regarding natural language, where the order of words plays a significant role in the context and interpretation, but it has also been proven to be as important for image classification tasks since images can be subdivided and therefore it can be also modelled as a sequence of data.

$$\begin{aligned} PE_{pos,2i} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{pos,2i+1} &= sin(pos/10000^{2i/d_{model}}) \end{aligned} \tag{2.4}$$

Where pos is the position of a token given a sequence, i is a token element index, d_{model} is the length of the token, The output PE of this sequence then is added to the input token respective element.

It is important to point out that even if the original Transformer presented uses sinusoidal encoding for this embedding, the authors also state that nearly identical results can be achieved using learned encoding, where the encoder is actually a series of parameters that are adjusted during training.

2.1.4 Feed Forward layer

Contrary to what was previously mentioned int this work, Feed Forward layer in a transformer does not correspond to a single fully-conected layer, instead [12] intends as a Feed Forward layer the network described by the Formula (2.5).

$$\text{FFN}(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{2.5}$$

Where x is the input matrix coming from the attention layer, W_1, b_1, W_2, and b_2 are the parameters of simple feed forward networks or linear layers. Thus the Feed forward network is just two linear layers that have in between them an activation layer.

2.2 Transformer based models

Since the original Transformer architecture appearance in 2017 there have been many other deep learning models that have inherited from it, these models models have advanced the field of deep learning, particularly in natural language processing (NLP). Leveraging the power of self-attention mechanisms and parallel processing, that were a huge standout for the original Transformer at the time, transformer based models are capable to handle large-scale data and complex tasks with unprecedented efficiency and accuracy. Models such as BERT and GPT, have pushed the boundaries of what can be achieved with artificial intelligence in general. Ths achievement makes Transformer-based models the go-to architecture for many cutting-edge AI applications. In this section we are going to present the three most relevant architectures currently BERT, GPT, and VIT.

2.2.1 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a groundbreaking Transformer-based model introduced by Google in 2018 [13], which has significantly advanced the state of the art in natural language processing (NLP). Unlike previous models that typically processed sequences in a unidirectional manner (left-to-right or right-to-left), BERT is designed to understand context in a truly bidirectional way, meaning it considers the entire sequence simultaneously when making a prediction. This bidirectional approach allows BERT to capture more nuanced relationships between tokens, leading to better performance on a wide range of tasks such as sentiment analysis [14]. Figure 2.4 displays BERT architecture, it can be appreciated that BERT can be reduced to just an stack of Transformer encoders connected sequentially. The important thing is that each token in a sequence has access to all tokes i.e. it does not mask the token that

come afterwards, allowing the model to relate all tokens with each other within a sequence.

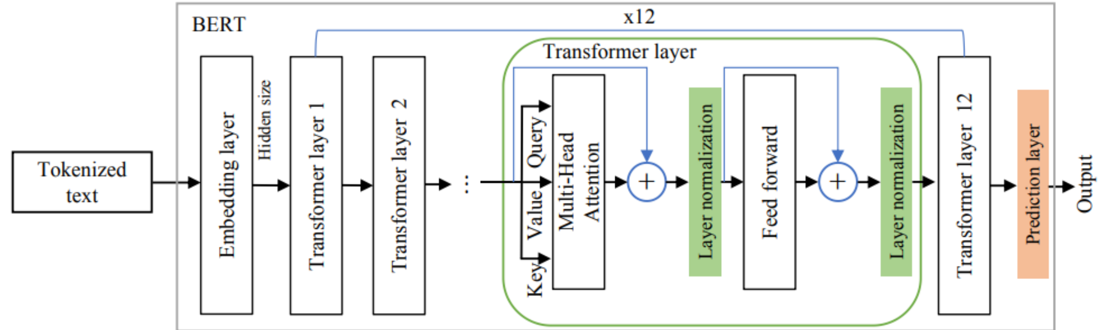


Figure 2.4: BERT architecture [14].

2.2.2 GPT

GPT (Generative Pre-trained Transformer) is a series of powerful Transformer-based language models developed by OpenAI, known for their ability to generate human-like text [15]. Different from BERT GPT operates as a unidirectional (left-to-right) language model. It predicts the next word in a sentence based on the preceding context, making it particularly well-suited for generative tasks where text needs to be produced sequentially, such as creative writing, dialogue generation, and code completion. Another difference with BERT is also that, GPT only uses of the Decoder block from the original Transformer architecture unlike BERT that only uses the Encoder block.

2.2.3 ViT

All the models that have been presented until now have mainly focused on solving NLP task, in the case ViT, which stands for Vision Transformer, it uses the transformer architecture to solve Computer Vision tasks such as image classification [16].

In the same manner as BERT, ViT leverages the Encoder block to perform inference with the slight difference that ViT uses a modified version of the Encoder, the modification being the position of the layer normalization blocks is shifted downwards and the use of GeLu activation function instead of the usual ReLu function. Figure 2.5 shows the vision transformer architecture in the form of a block diagram. To pass images to the encoder, Vision Transformer divides the image into smaller ones referred as patches that are then tokenized. Patch tokenization

consist of flattening the sub-image's dimensions and channels into a 1-D vector that is grouped with the other flattened patches, making them a sequence. Once the sequence is built Each patch is linearly projected into a fixed-dimensional embedding space, i.e. passed through an embedding layer that consist of a single linear layer this layer also helps embedding position into each patch transforming them into token, thus being able to serve as an input to the encoder.

2.2.4 Transformer model size discussion

Having presented the Transformer architecture and some of it's most know variations, it is necessary to reference each model size in number of parameters. Table 2.1 displays the size in parameters of each, notice that the sizes vary from millions of parameters to billions, this means that executing models with sizes like these represent a huge demand on the computer in regards to memory, to store, or access parameters; computation, since in order to have acceptable execution times the model demands a large amount of operations per seconds; and lastly energy since the execution of these models also spends a considerable amount of energy. This demands affect the overall usage of this models since they represent costs that make them unsuitable to be used outside of data-centers dedicated to training and inferencing of neural networks, to reduce this demands there are some techniques to reduce the overall complexity of a particular model without an appreciable degrade in performance, one particular technique to reduce the model size is called quantization which is going to be the subject of the next chapter.

Model	**Parameters (approx.)**
Original Transformer (Base)	65 million
BERT Base	110 million
BERT Large	340 million
GPT (GPT-1)	117 million
GPT-2 Small	117 million
GPT-2 Medium	345 million
GPT-2 Large	762 million
GPT-2 XL	1.5 billion
GPT-3 (175B)	175 billion
ViT Base	86 million
ViT Large	307 million
ViT Huge	632 million

Table 2.1: Comparison of Model Sizes: ViT, BERT, GPT, and Original Transformer

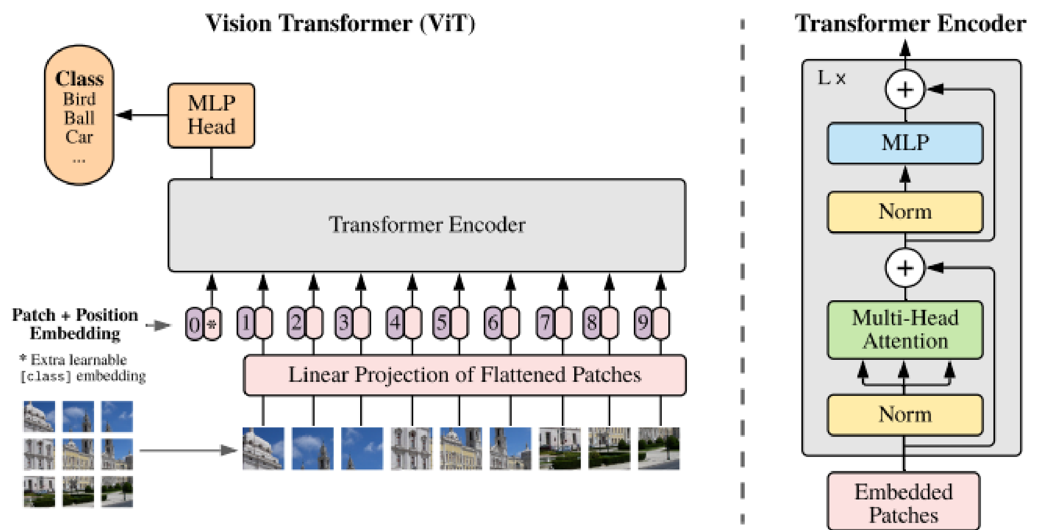


Figure 2.5: ViT model overview [16].

Chapter 3

Model Optimization

Given the growing size of Transformers based models and it's impact on scalability, time of inference and overall cost, it becomes apparent that in order to have models with an increasing number of parameters of parameters, something must be used to reduce the size of the model and the complexity. fortunately given the flexible nature of neural networks, i.e. they can be retrained and adjust the parameters to better fit the task at hand, some optimizations on the way the model behaves can be performed. The optimized model can be achieved using one or many techniques such as knowledge distillation [17], pruning [18], and or model quantization [19].

3.1 Quantization overview

Neural network quantization is a technique used to reduce the size and computational complexity of deep learning models by mapping the parameters (weights and biases) with lower-precision representations, typically from floating-point (e.g., FP32) to integers (e.g., INT8). The goal is to make neural networks more efficient, especially in environments where computational resources are limited, such as edge devices, mobile platforms, and embedded systems. The benefits from performing quantization from FP32 to INT8 are noticed in the decreases of memory overhead, since storing parameters and intermediate results decreases by a factor of 4 while the computational cost for matrix multiplication reduces quadratically by a factor of 16 [20].

Mathematically Quantization is mapping values from a continuous space to a discrete space, This mapping is performed using a constant piece-wise function, such as(3.1). This mapping is what then is going to be used to compress the values of a floating point number, to an integer one [21].

$$Q(r) = q_i \\ r \in [r_i, r_{i+1}) \tag{3.1}$$

It is possible to distinguish two forms in which intervals are assigned to values, uniform and non uniform. when doing the conversion from a high precision value to an integer it is preferred to do an uniform mapping, since it allows to also perform computations in the quantized domain [22].

Uniform quantization means that the ranges are divided in equal intervals, thus in the case of integer numbers stored in bits this operation can be represented as formula (3.2) shows:

$$s = \frac{2^b - 1}{\alpha - \beta} \\ z = -\mathrm{round}(s\beta) - 2^{b-1} \tag{3.2}$$

where s is the scale factor and an z is the zero-point the integer value to which the real value zero is mapped. Once the mapping is performed, to get the particular representation of a number in continuous space to the quantized space the Formula (3.3) is employed.

$$x_q = \mathrm{quantize}(x, b, s, z) = \mathrm{clip}(\mathrm{round}(sx + z)), -2^{b-1}, 2^{b-1} - 1) \tag{3.3}$$

Where the round function rounds the decimal value to the nearest integer, and the clip function handles overflow by setting the quantized value to the maximum or min values allowed by the number of bits b that used to represent the integer. In contrast to go from the quantized space to the continuous spaces the Formula (3.4), notice that returning from the quantize space does not yield the original continuous value, that is the main reason why quantization carries out a loss of precision.

$$x = \mathrm{dequantize}(x_q, s, z) = \frac{1}{s}(x_q - z) \tag{3.4}$$

It is important to point out that, the dynamic range of the continuous space is assumed to be symmetric, i.e. that is centered at zero, cancelling the z term of Formulas (3.3) and (3.4).

Once we have the expressions to transit to or from the quantize space, it is possible to apply quantization to the inputs of an specific operation, and dequantization to the outputs. In this manner the operation uses integer arithmetic that is far more efficient than floating-point arithmetic.

3.1.1 Quantization Classification

Having explained quantization, the follow up question appears. How to set the scale factor and zero point, values of a given operation or in other words how to

determine the dynamic range of the input values and output values of a layer. To answer that question, there are many techniques employed to perform quantization, weather fully or partially to some layers of the networks. these techniques can be group in three classes Post Training Quantization PTQ, Quantization Aware Training QAT.

Post training quantization

This method quantizes the model after it has been trained. It is relatively easy to implement but may lead to accuracy degradation since the network was not trained to handle quantized values.

Quantization aware training

In QAT, the model is trained with quantization in mind. The training process simulates quantized values (e.g., using INT8), so the model learns to adjust to these lower-precision operations. This approach generally preserves more accuracy than PTQ.

3.2 Model Pruning overview

Model pruning consist of removing unnecessary or less important parameters (weights or connections) from a trained model to make it more efficient in terms of memory usage, speed, and power consumption. In the same manner as Quantization, Pruning is trade off between small degradation in model quality, and a reduction in model size.

Pruning can be classified in several ways as [23] suggest, one of the ways consist on how coarse the pruning method i.e. whether pruning consist of zeroing parameters or removing neurons, filters, or heads of a model. these two kinds approaches are called as unstructured, structured, and an in between approach simply named semi-structured.

3.2.1 Unstructured Pruning

Also known as weight-wise pruning, it is the finest-grained case. Mathematically unstructured pruning can be formulated as an optimization problem [24]:

Given a dataset $D = \{(xi, yi)\}_{i=1}^{n}$, where D is composed of x_i input and y_i output pairs; and a desired sparsity level k (i.e., the number of non-zero weights) the resulting weights are obtained using Formula (3.5).

$$\min_{W} L(W; D) = \min_{W} \frac{1}{N} \sum_{i=1}^{N} \ell(W; (x_i, y_i))$$
$$\text{s.t.}\, W \in \mathbb{R}^m,\ \|W\|_0 \leq k \tag{3.5}$$

Where W represents the weights of a neural network, $\ell(.)$ is a loss function, m the shape of the weights, and $\|.\|_0$ counts the number of non zero elements that vector contains.

It is important to point out that although unstructured pruning reduces the model size without actually impacting the accuracy, it does not reduce the computational complexity of the model. Moreover it adds to it given that the newly created W parameters must be stored in a way that signals out the sparsity, therefore a decoding algorithm or dedicated hardware is needed to process the weights effectively giving an overhead to the overall inference time.

3.2.2 Structured Pruning

Structured pruning is much more coarse than unstructured pruning since it mainly consist on removing network connections, such as neurons, channels, filters, or layers, rather than individual weights. contrary to unstructured pruning, this approach is more hardware-friendly, leading to models that are more efficient to run on common hardware (such as GPUs and CPUs) since they require less calculations, but it does have an impact on accuracy.

In the same manner as unstructured pruning, structured pruning can also be defined as simple optimization problem [23]:

Given a specific prune ratio and a neural network with $S = \{s_1, s_2, ..., s_L\}$, where s_i can be the set of channels, filters, neurons, or transformer attention heads in layer i. Structured pruning aims to search for $s' = \{s'_1, s'_2, ..., s'_L\}$ to minimize performance degeneration and maximize speed improvement under the given prune ratio, where $s'_i \subseteq s_i$, $i \in \{1, \ldots, L\}$.

3.2.3 Semi-Structured Pruning

Also called pattern based-pruning [25], where the primary focus is on designing and selecting optimal patterns. The goal is to achieve high accuracy and execution efficiency by considering theoretical implications, algorithm design, compiler optimizations, and hardware performance.

One example of pattern based pruning could be dividing an weight matrix into smaller sub-matrices that then each are unstructured prune a fixed amount of weights as [26] presents.

3.3 Knowledge Distillation

Unlike the previous, techniques where there is no requirement for training to be involved. Knowledge distillation consist of compression technique where the knowledge is transferred from a large pre-trained teacher model to a small student model, so it can replicate or mimic the teacher model's behavior. The resulting model is then more efficient than the original but has comparable accuracy to the original model [27]. Figure 3.1 shows a representation of how this technique works.

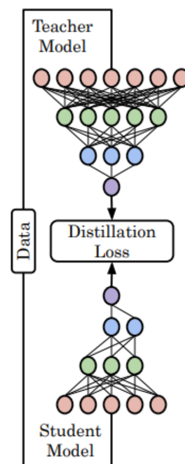


Figure 3.1: Knowledge Distillation process [27].

Notice that differently from the other methods where the optimization can take part after training, by solving an optimization problem. Distillation can take part only on the training of the student model, where the learning algorithm or loss function, takes as an input not only the dataset, but also the output of the teacher model.

Chapter 4

FPGAs and Deep Learning Inference

Field Programmable Gate Arrays (FPGAs) are programmable devices that offer a flexible platform for implementing custom hardware at a relatively low development cost. They also allow the user to add features, or fix bugs by simply loading a new configuration after deployment in-field, thus the name field-programmable [28]. These features make FPGAs ideal accelerators for deep learning inference, as they support fine-grained parallelism and associative operations like broadcast and collective response [29]. In order to achieve this goal, it is crucial to make proper usage of the internal resources the FPGA proivdes. This implies that the accelerator design consist of a trade-off between overall parallelism and reuse of resources. This whole chapter consist of introducing FPGA components, how do an FPGA interacts with memory and other peripherals, and how the accelerator design flow usually consist when designing an accelerator for a Transformer model.

4.1 FPGA overview

FPGA chips have come a long way since their first appearance over 30 ago. Nowadays, modern FPGAs are becoming more and more complex than just simple arrays of programmable logic and IO blocks. They are primarily composed of programmable logic cells, known as configurable logic blocks (CLBs), a programmable interconnect network, and programmable input/output cells surrounding the device. Additionally, FPGAs include a variety of embedded components such as digital signal processing (DSP) blocks, used for tasks like multiply-and-accumulate operations, block RAMs (BRAMs), look-up tables (LUTs), flip-flops (FFs), clock management units, and high-speed I/O links, all interconnected through a hierarchy of programmable switches [28] as presented in Figure 4.1.

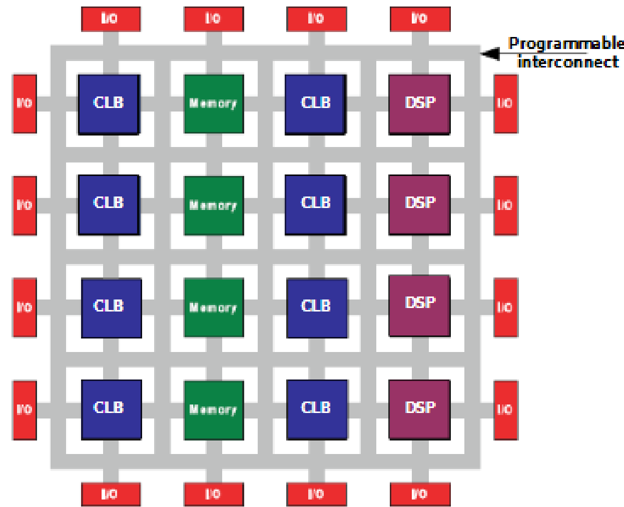


Figure 4.1: FPGA Block Diagram [30].

4.1.1 FPGA based SOCs and SOMs

Aside from the general evolution of FPGAs and continuous addition of more and more complex modules inside the die, FPGA boards normally called System on Modules have also evolved to the point where they can function not only as an accelerator but also execute the complete applications, generally this boards consist of an FPGA chip, with multiple interfaces that connect the chip to memory IO peripherals, and also PCI ports that serve as a communication High speed link between board and the host computer. Moreover many vendors offer FPGA chips that not only hold the actual FPGA but integrate it with ARM aplication processors, and more specialized computation DSP called vector processors. One major example of these systems on chips (SOC) can be the AMD Xilinx Versal AI Edge series shown in Figure 4.2.

4.2 FPGA Design Flow

Having Defined the algorithm, i.e. the optimized Transformer model, and the FPGA device that is in charge of accelerating inference, The remaining step is implemneting accelerator, in order to do so there is general work flow that consist of the following steps:

1. Architecture Design: Based on the algorithm's needs, choose a hardware architecture that best supports parallelism (e.g., dataflow, pipelining, loop unroling, etc).

Figure 4.2: Versal Soc Module components [31]

2. Partition the design: Divide the design into functional blocks or modules, such as processing units, memory controllers, communication interfaces, and control logic.
3. Data flow analysis: Determine how data will be passed between different blocks and the memory.
4. Model the design: Write the accelerator architecture and behaviour using High level synthesis or RTL code.
5. Debug: Check that the implemented accelerator model produces the same results as the optimized AI algorithm. This step is performed by means of a testbench capable of evaluating the correctness of the modeled accelerator behaviour.
6. Synthesize the accelerator: This step consist of mapping the FPGA resources, scheduling the operations, and interconnecting them resulting in what is called a net-list. Synthesis procedure is highly automated as of right now but the designer can add some configurations that can guide the CAD tools used.
7. Verification: Check that the synthesis result makes a sound use of the available resources and that it's performance hit the expected mark. This step also

performs logic or RTL level simulation to check that the net-list matches the behaviour of the HDL or High level synthesis code.

Chapter 5

FPGA accelerators for Transformer model inferencing

Having presented the stages to accelerate a Transformer using FPGA, we can discuss the concrete challenges and solutions when implementing the accelerator.

5.1 Fitting Transformer into an accelerator

Referencing Chapter 2 of this work, Transformers vast number of parameters and inputs along with the fact that AI models operate withe them in the domain of real numbers, thus being map to FP32 in computer systems, the memory footprint and overall computational complexity is such, that it becomes not possible to execute inference on smaller devices such as FPGA. To solve this issue, designers resort to the model compression techniques similar to the ones presented in chapter 3 of this work.

5.1.1 Quantized Transformer Accelerators

Compressing the model using quantization not only reduces the size of parameters but also allows for the implementation of integer arithmetic instead of floating point, which highly reduces the usage of DSPs, and also increases the performance since integer computation is lighter. However, quantization methods vary from implementation to implementation. For the rest of this section we are going to reference works that performed or used quantization and a brief mention of what was done:

[32] implements integer quantization for the attention mechanism in a non-uniform manner, mapping parameters and activations to INT16, while inputs are mapped to INT8. In contrast, [33], [34], and [35] apply INT8 quantization to all layers and parameters except for the softmax layer, where floating-point arithmetic is used. [36] also uses INT8 quantized models, but performs 1-bit quantization on the Q and K matrices. [37] performs INT8 quantization-aware training (QAT) for BERT-based models. [38] takes a more aggressive approach, assigning 4-bit quantization to the weights and 8-bit quantization to the intermediate layers in BERT. [39] uses INT8 quantization for linear operations, INT16 for activation layers, and 32-bit precision for exponential and square-root. Another approach by [40] uses a mixed quantization scheme, with some layers using uniform fixed-point quantization and others employing non-uniform quantization. [41] designs an accelerator capable of handling models with different levels of quantizations, it was tested with resolutions of 8 bits, 6 bits, and 4 bits. [42] proposes different technique to quantize the network, using Hessian analysis, it mainly consist on performing mixed quantization row-wise, i.e. quantization differs from one row of parameters to the another. [43] uses INT4 for quantization for parameters and downscale activation from the usual FP32 to FP16. The most extreme form of quantization reviewed was the one used by [44] which uses binary BERT models as base for the accelerator.

5.1.2 Pruned Transformer Accelerators

Another broadly used approach for compressing a Transformer to fit into an FPGA is the pruning of parameters and operations in a transformer, in the same fashion as quantization we are going to reference works that used it and briefly explain how it was implemented:

[45] introduces block-wise balanced pruning, a semi-structured technique where a parameter matrix is divided into blocks, and a specified number of parameters are pruned from each block. The pruned weight matrix is then compacted, and a bitmap of ones and zeros is used to indicate the positions of the remaining weights in the original matrix.

In regards to semi-structured pruning, [46] introduces $N : M$ sparsity method, where the weight matrix is divided into N groups of columns, each containing M non-zero elements. They also propose an algorithm to optimize both the sparse matrix and the N configuration. Similarly, [47] applies a structured pruning method known as column-based pruning, combining block-wise pruning with the $N : M$ sparsity approach.

[48] develops an accelerator capable of supporting both block-wise and block-wise balanced pruning. In contrast, [49] is the only work reviewed that implements an accelerator for unstructured pruning. Finally, [50] stands out as the only

study dynamic token pruning, i.e. there is dedicated hardware to prune tokens in the attention matrix, and static pruning to get sparse weight matrices, where the attention weights are pruned using semi-structural block pruning and the FeedForward layers are pruned using structural row/column pruning.

5.2 Modelling Transformer operations in FPGA

Following the design flow proposed in Chapter 4, one of the most crucial steps in designing an accelerator is partitioning the design into modules. For Transformers, accelerators are typically divided into modules that perform the core mathematical operations within them, concretely, matrix multiplication, softmax, and layer normalization. In this work, we will discuss the rationale behind how each some works design the architecture of each individual model.

5.2.1 Matrix Multiplication

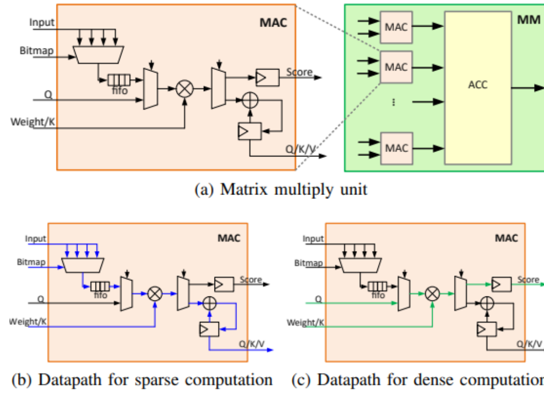
Because matrix multiplication is the most prevalent operation in the model, and Transformer models have grown to support long input sequences with many features, matrix multiplication engines are designed to be highly concurrent, memory efficient, and adaptable to make the most out of the FPGA resources. Some remarkable approaches on how to compute matrix multiplication are:

[41] explains how to perform matrix multiplication when using integer quantization, as represented in formula 5.1. Where A_q^3 is the output quantized matrix, A_q^1 and A_q^2 the input matrix S the scale factor, "Z" the zero point and $\sqrt{d_{model}/h}$ the scale factor used for the self attention computation.

$$A_q^3 = \frac{S_{A^1} S_{A^2}}{S_{A^3}\sqrt{d_{model}/h}}((A_q^1 - Z_{A^1})(A_q^2 - Z_{A^2})) + Z_{A^3} \tag{5.1}$$

Since [45] uses pruning, it exploits the sparsity of the resulting matrices by designing Multiply and Accumulate modules with support for sparse row computation, as Figure 5.1 shows. For sparse computation, Figure 5.1(b), the inputs and weight parameters are sent to the MAC unit. The bitmap input selects the non-zero input values, and the selected values are stored into the FIFO, The FIFO length is determined by the number of reserved non-zero values inside a small block of the input matrix. Once the FIFO is full, the computation between the stored values and the weights is made. For dense computation, Figure 5.1(c), the values are sent directly into the MAC unit to perform computation.

[51] implements a technique denominated DSP-packing which consist of arranging the inputs of a given DSP to compute multiple results while doing one single operation. In particular two ways of packing are given, one that realizes two 8-bit



(a) Matrix multiply unit

(b) Datapath for sparse computation (c) Datapath for dense computation

Figure 5.1: [45] Matrix Multiplication architecture

multiplications $A \cdot C$ and $B \cdot C$ where C operand is shared, and another one that supports four 4-bit multiplication where each multiplication also shares one single operand, Thus improving the throughput of matrix multiplications since a single DSP can process up to 4 unrelated multiplications. The DSP employed consist of a signed multiplier that operates a 27-bit number with a 18 bit number, and the result is then fed to an adder that takes as input a number of 35-bits and outputs a 48-bit number. The architecture of 8-bit multiplier and 4-bit multiplier are shown in Figures 5.2 and 5.3 respectively.

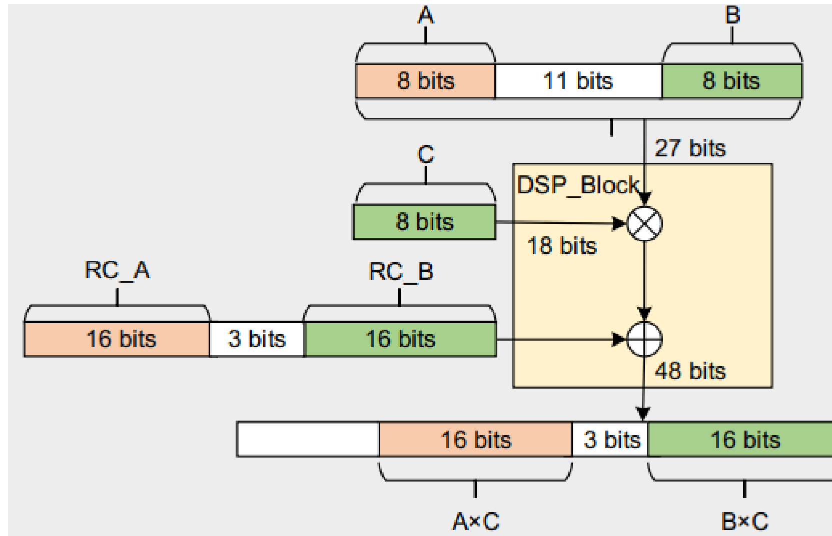


Figure 5.2: 8-bit multiplier architecture [51], where A and B are the operands multiplied by C. RC_A and RC_B are added to correct the result of the initial multiplication, RC_A is 1 when C is signed and RC_B is $-\mathrm{B}_{\mathrm{MSB}} \cdot 2^8 \cdot \mathrm{C}$.

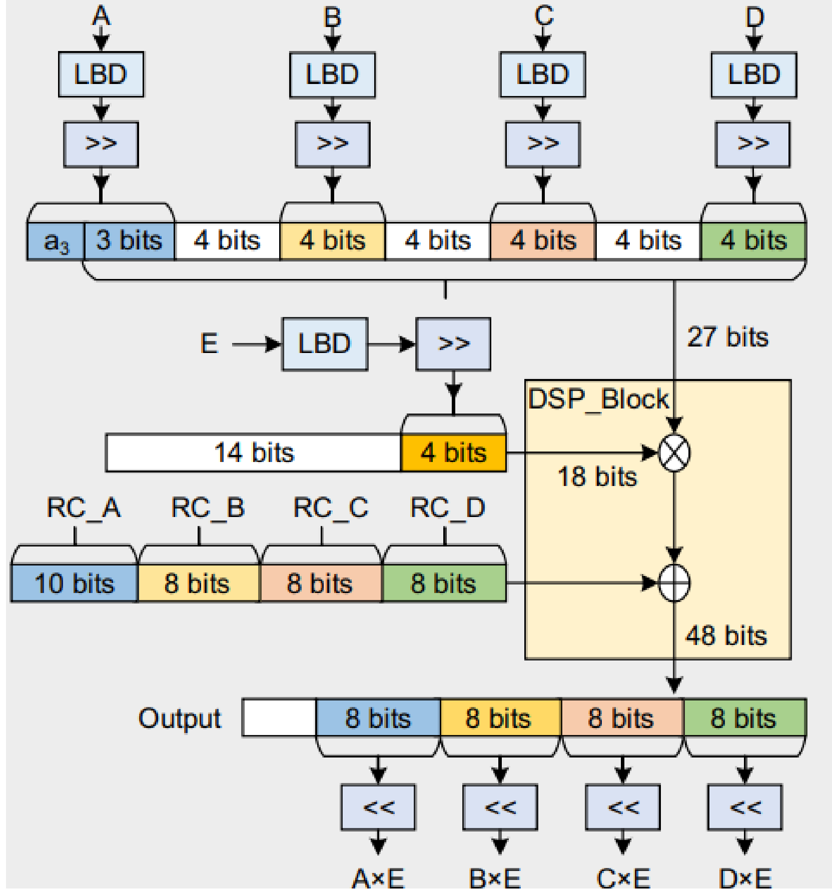


Figure 5.3: 4-bit multiplier architecture [51], where A, B, C, D are the operands multiplied by E. the shifters at the beginning truncate the inputs to 4-bit representation and at the output are restored to their original size. Much like with the $B \cdot C$ correction for the 8-bit multiplier RC_B, RC_C, and RC_D for the 4-bit are calculated obtained in the same manner. As for RC_A is $(-A_{\mathrm{MSB}} + A_{\mathrm{MSB\text{-}1}}) \cdot 2^3 \cdot E$.

[42] implements yet another type of matrix multiplication module made of Processing Elements that support 8-bit x 8-bit operations and 1-bit x 8-bit operations, this is due to the mixed quantization used for parameters where parameters classified as important, represented in 8-bit quantization, and the unimportant, represented using only 1-bit, Figure 5.4 displays the architecture of the aforementioned PEs.

[43], [37], [52], [32], [46], and [33] perform matrix multiplication using a systolic array, this architecture consist of a grid conformed by identical cells or processing elements (PE) that calculate concurrently partial results of matrix multiplication that then pass the partial result to following PEs, this implementation theoretically

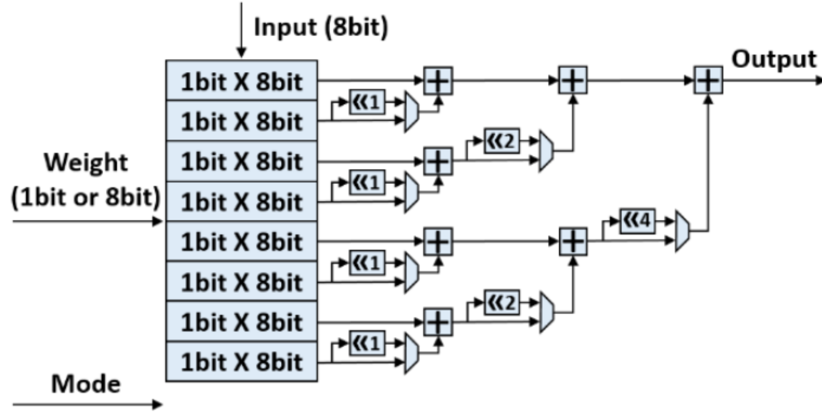


Figure 5.4: Processing element for matrix multiplication [42]. When multiplying 8-bit by 1-bit values the PE operates 8 multiplications concurrently accumulating them at the end. Instead, when operating 8-bit × 8-bit mode, the PE can only performs one single multiplication at a time.

reduces the matrix multiplication time complexity to $O(n)$, and produces at each iteration a column of the output matrix. Thus the aspect that differentiates each work is the structure of their respective PEs: [32] designs processing elements to support multiplication between matrices of different integer types, more specifically to support multiplications of 16-bit by 8-bit and multiplications of 8-bit by 8-bit integers, this is due to how intermediate activations are stored with a higher bit-width than parameters and inputs; Sharing the same reasoning [43] designed PEs support multiplications between FP16 for the activations and INT4 for the weights and can process in parallel vectors instead of individual elements; On the other hand, [46] implement PE that support dense or sparse row multiplication, given that it uses N:M pruning optimization; [33] and [37] uses the straightforward PE that consist only of a MAC; whereas [52] uses the DSP48E2 embedded IPs inside the target Xilinx FPGA.

[53] and [54], Differentiate two distinct modules to handle matrix multiplication, the first one handles matrix multiplications inside of the attention self-attention product. It consist of a two accumulators, dividers and exponential units that support the scale and Softmax operations required by multi-head attention. The second module is used for the linear layers since trainable weights are stored as the fourier transform of block circulant matrices. A block circulant matrix W consist of b^2 square circulant sub-matrices $B_{i,j}$ where each row of $B_{i,j}$ is a cyclic reformulation of the other rows. This implies that to store a circulant matrix it is only needed to store one of it's rows $p_{i,j}$. Aside from this advantage, matrix to vector multiplication can be computed as shown in Formula 5.2, where the $\circ$

is the Hadamard product. This optimization is valuable since the computational complexity of the linear layers gets reduced to $O(b\log(b))$.

$$W_{i,j}x = p_{i,j} \circledast x = IFFT(FFT(p_{i,j}) \circ FFT(p_{i,j})) \tag{5.2}$$

[44] implements another novel approach, since it uses binary transformers where weights can take values of either -1 or 1 matrix multiplication gets much more simpler, and drastically reduces the amount of resources, since matrix multiplication becomes a matter of performing additions between the elements in a row. The proposed module is shown in Figure 5.5, The reasoning behind this approach is to reuse as much as possible the accumulate result of row items. Notice that Figure 5.5 includes the binary softmax implementation which is going to be explained in the softmax subsection.

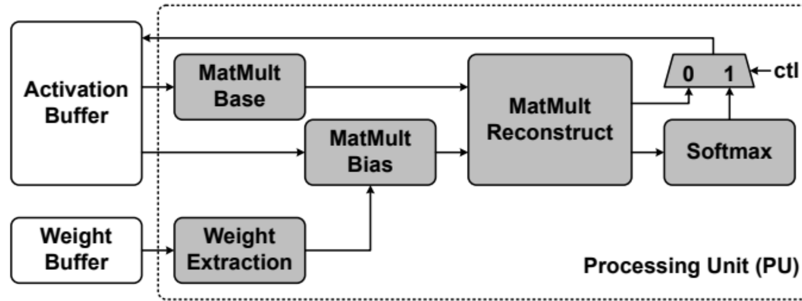


Figure 5.5: Binary transformer Matrix multiplication unit of the proposed accelerator [44]

Regarding the MatMul operation in [44], it can be sumarize in three steps. In the first step, the original weight matrix is decomposed into based matrix which has all the elements set to one and bias matrix. The meaning of each element inside bias matrix depends on the amount of +1 in the original weight matrix. When there are more +1 than -1 in the original weight matrix, the value of 0 correspond to +1 and the value of 1 corresponds to -1, in case there are more -1 the opposite occurs. In doing this, the number of 1s in the bias matrix can be reduced. For the second step, the activation matrix is multiplied by the base matrix and the bias matrix separately. Since the base matrix consist of only ones, what happens in reality is that the elements of each rows are accumulated and saved to be reused multiple times, whereas the multiplication with the bias matrix is done normally. The final step consist of reconstructing the result of MatMult, this is done by subtracting the result of the bias matrix multiplication to the base matrix multiplication twice, then choosing weather to invert the result or keep it as it is. This last choice is based on the majority of +1 in the original weight matrix, if

there is majority +1 the resulting matrix becomes the output else the resulting matrix is inverted. Figure 5.6 illustrates this process with an example.

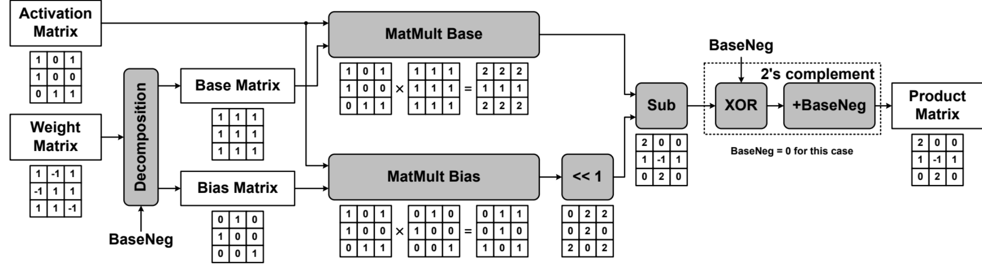


Figure 5.6: Matrix multiplication process for a binary Transformer example [44].

[38] uses a more traditional approach consisting of integer multiplier, adder trees and, accumulators to perform matrix multiplication. given that it uses different INT8 types for activations and INT4 types for weights, the matrix multiplier must have support for 8-bit by 8-bit multiplication and 4-bit by 8-bit, Figure 5.7. It consist of and array of multipliers capable of handling sign multiplication if needed and shifters that actuate depending on the types of the opperand. From Figure 5.7 it can be noticed that the multiplier can accumulate the products of four 8-bit by 4-bit multiplications or two 8-bit by 8-bit multiplications. In the first case no shifting is needed whereas in the second case the inputs are rearranged and shifted according to formula (5.3), where A, B, C, D are all eight bit operands.

$$\begin{aligned} AB + CD &= ((A_h \cdot 2^4) + A_l)B + ((C_h \cdot 2^4) + C_l)D \\ &= ((A_h B + C_h D) \cdot 2^4) + (A_l B + C_l D) \end{aligned} \tag{5.3}$$

[39] tries to optimize matrix multiplication performance by reducing the amount of accesses to external memory using output block stationary data-flow (OBS), this means performing block matrix multiplication while attempting to reduce the writes to external memory, this is done by adding the partial results of block matrix multiplication internally leveraging the internal built in memory of modern FPGAs BRAM.

5.2.2 Softmax

Given that the softmax function is non linear comprising exp() function and division operations, most of the efforts when implementing this part of the self-attention mechanism goes to approximating or reducing the amount of resources needed to compute the exponential function. In this current work we are going expose how some work have solved this issue:

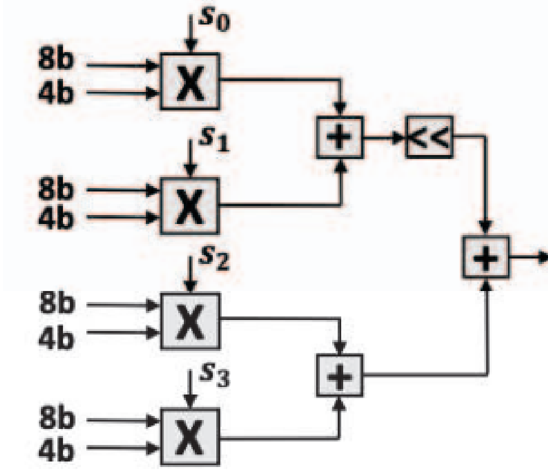


Figure 5.7: Processing Element capable of handling 8-bit by 4-bit and 8-bit by 8-bit [38].

[41] adopts a Softmax implementation based on lookup tables for computing exp() for all possible inputs. This approach relies on two lookup tables: $NLUT$ for numerators and $DLUT$ for denominators. exp() outputs values in the interval (0, 1], to ensure numerical stability they offset inputs X_q by the maximum value of the input array, yielding $\hat{X}_q$. Subsequently, the exponential value are derived through $E = \exp(\hat{X}_q)$. they also introduce a scaling factor S_E and a zero point Z_E as defined in Equations (5.4) and (5.5), accommodating the dynamic range within the constraints of integer precision. The $DLUT$ and $NLUT$ are calculated using Equations (5.6) and (5.7), respectively, ensuring bit-widths of $2b$ for $DLUT$ and $3b$ for $NLUT$ to maintain sufficient precision. The final output A_q is obtained applying Equation (5.8). where $i \in [1, n]$.

$$S_E = \frac{1}{((2^{2b-1} - 1) - (-2^{2b-1}))/(n^2 h)} \tag{5.4}$$

$$Z_E = 2^{2b-1} - \frac{1}{S_E} \tag{5.5}$$

$$DLUT(\hat{X}_q) = \text{clamp}(\text{round}\frac{E}{S_E}), -2^{2b-1}, 2^{2b-1} - 1) \tag{5.6}$$

$$NLUT(\hat{X}_q) = \text{clamp}(\text{round}\frac{E}{S_E S_A}), -2^{3b-1}, 2^{3b-1} - 1) \tag{5.7}$$

$$A_q = \frac{NLUT(\hat{X}_q)(i)}{\sum DLUT(\hat{X}_q(i)) - Z_E} \tag{5.8}$$

[51] and [51] Treat the softmax function as a piece-wise linear function. The hardware architecture is shown in Figure 5.8 and it is taken from the work of [55]. It consists of a locating unit to identify which interval the input belongs, an N-entry LUT to store approximation parameters (k_i, b_i), a multiplier and an adder to calculate the aproximated result. the locating unit is composed of N parallel subtractors where N is the number of segments the piece-wise function has, the sign bits resulting of each subtraction between the input and the maximum values of each segment are then used to select (k_i, b_i), where k_i is multiplied with the input and b_i added to the subsequent result producing the approximated softmax output.

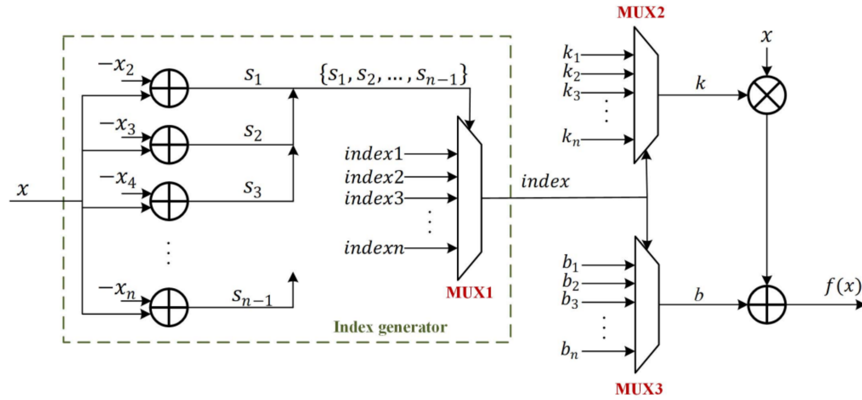


Figure 5.8: Piecewise Softmax architecture [55]

[42] copies the softmax architecture present in [56] where m Softmax components are instantiated and work concurrently, being m the number of rows in the attention matrix. A single softmax operator is shown in Figure 11. It can be divided into three stages, search for the maximum value, calculate the exponential, and produce output as shown in Figure 12. This implementation aims at restricting the range of values with whom the exponential function needs to be computed, by offsetting all inputs by the maximum value $\tilde{x} = x - x_{max}$, the exp() operates only with negative numbers, Consequently, negative numbers can be decomposed as $\tilde{x} = (-\ln 2)z + p$ where $z = \lfloor -\tilde{x}/\ln 2 \rfloor$ is a positive integer and $p = \tilde{x} + z \ln 2$ is a number restricted to $(-\ln 2{,}0]$ [57], this implies that for the case of negative real numbers exp() can be expressed as formula (5.9).

$$\exp(\tilde{x}) = 2^{-z} exp(p) = \exp(p) >> z \tag{5.9}$$

from (5.9) the range in which the exp() function will operate is the same as p allowing to approximate it to the polynomial expression of form $a(x+b)^2 + c$ shown in formula (5.10) and replacing it into (5.9) yields formula (5.11).

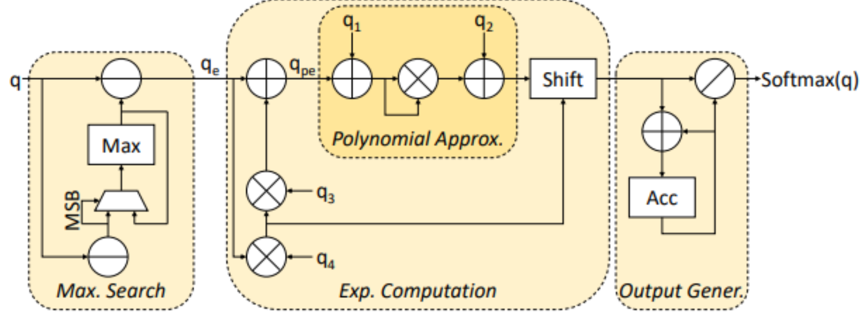


Figure 5.9: Architecture of the softmax operator. from [56], from Algorithm 1 "$q_1 = b/S_{pe}$, $q_2 = c/aS_{pe}^2$, $q_3 = \ln 2/S_{ec}$, and $q_4 = -1/q_3c$, where S_e is the scaling factor of the exponential computation input (relative to q_e), S_{pe} is the scaling factor of the polynomial approximation input (relative to q_{pe}) and a, b, c are the coefficient of the second-order polynomial that approximates the function, i.e., $a(x + b)2 + c$. Therefore, $q_{1,2,3,4}$ can be computed at design time and provided as constant values".

$$L(p) = 0.3585(p + 1.353)^2 + 0.344 \approx \exp(p) \tag{5.10}$$

Finally Algorithm 1 summarizes the behaviour of the softmax unit in [42] by taking into account the effects of quantization.

$$\exp(\tilde{x}) = L(p) >> z \tag{5.11}$$

[46] Presents a re-configurable and scalable softmax architecture that has two adjustable parameters, P and Q, where P denotes the parallelism of the architecture and Q represents the pipeline depth, as well as the output precision. The module operation is as follows: P input data is streamed in parallel and used to calculate the exponential function, which is approximated using a look up table and a first order Taylor-approximation consisting of a multiplier and an adder. The exponent outputs are temporarily stored in the data buffer and also used as input for the accumulator that computes the denominator of the softmax function. Once the accumulation is complete, the divider uses the exponent output and the accumulator result to perform Q-level pipelined division where each stage consist of a subtractor and a shifter, that grouped together generate P softmax function outputs with Q bits of precision. As shown in Fig. 5.10, the scalable softmax module is composed of three major parts: the exponent function, the accumulator, and the scalable divider.

[33] merges the mask operation and the softmax function from the attention function into a single module, this combination is expressed with formula (5.12).

Algorithm 1 Integer-only Exponential and Softmax [57]

Require: q, S: quantized input and scaling factor
Ensure: $q_{\text{out}}, S_{\text{out}}$: quantized output and scaling factor

```
 1: function I-POLY(q, S)                                  ▷ qS = x
 2:     q_b ← ⌊b/S⌋
 3:     q_c ← ⌊c/aS^2⌋
 4:     S_out ← ⌊aS^2⌋
 5:     q_out ← (q + q_b)^2 + q_c
 6:     return q_out, S_out                ▷ q_out S_out ≈ a(x + b)^2 + c
 7: end function
 8: function I-EXP(q, S)                                   ▷ qS = x
 9:     a, b, c ← 0.3585, 1.353, 0.344
10:     q_ln2 ← ⌊ln 2/S⌋
11:     z ← ⌊−q/q_ln2⌋
12:     q_p ← q + z q_ln2                                  ▷ q_p S = p
13:     q_L, S_L ← I-POLY(q_p, S) with a, b, c
14:     q_out, S_out ← q_L ≫ z, S_L
15:     return q_out, S_out                      ▷ q_out S_out ≈ exp(x)
16: end function
17: function I-SOFTMAX(q, S)                               ▷ qS = x
18:     q̃ ← q − max(q)
19:     q_exp, S_exp ← I-EXP(q̃, S)
20:     q_out, S_out ← q_exp/sum(q_exp), S_exp
21:     return q_out, S_out                  ▷ q_out S_out ≈ Softmax(x)
22: end function
```

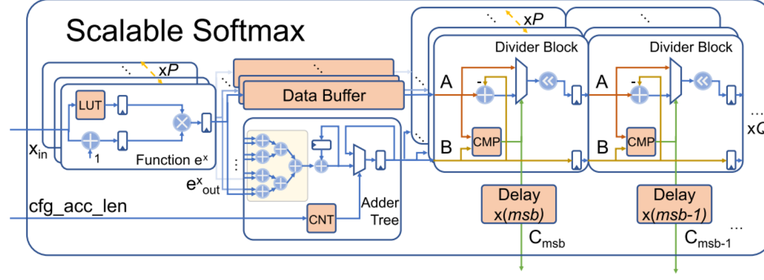


Figure 5.10: Architecture of the scalable softmax unit [46].

$$Y(i,j) = \begin{cases} \frac{\exp\left(\frac{X(i,j)}{8}\right)}{\sum_{j=1, Mask(i,j)=0} \exp\left(\frac{X(i,j)}{8}\right)} & \text{if } Mask(i,j) = 0, \\ 0 & \text{if } Mask(i,j) = 1. \end{cases} \quad (5.12)$$

Apart from the merging of the softmax and mask operations another particular technique employed is the application of the log sum-exp trick which reformulates the softmax function as shown in 5.13, this reformulation takes away the division operation and instead uses ln() operation.

$$\text{Softmax}(x_i) = \frac{\exp(x_i - x_{\max})}{\sum_{j=1}^{d_k} \exp(x_j - x_{\max})} = \exp(x_i - x_{\max} - \ln(\sum_{j=1}^{d_k} \exp(x_j - x_{\max}))) \quad (5.13)$$

The architecture of the accelerator is shown in Figure 5.11, this architecture computes the soft max algorithm in four steps: the first one gets $x_{\max}$, the second computes $\sum_{j=1}^{d_k} \exp(x_j - x_{\max})$, the third steps the log function, and the fourth and final computes computes the outputs. It is worth noting that the fixed 3-bit shift at the beginning of the unit, corresponds to the application of the scale factor of the attention mechanism that in this particular case is equal to eight.

To implement the ln() and the exp() function [33] replicates the works shown in [58]. To implement the exponential function, first it is expressed with a base of two as formula (5.14). Then the expression $x \log_2 e$ is decomposed into an integer part $u = \lfloor x \log_2 e \rfloor$ and $w \in (0,1]$ the remaining decimal part, replacing $x \log_2 e$ with u and w in (5.14) yields formula (5.15).

$$\exp(x) = 2^{x \log_2 e} \quad (5.14)$$

$$\exp(x) = 2^{u+w} = 2^u \cdot 2^w = \begin{cases} 2^w << u & u > 0 \\ 2^w >> (-u) & u > 0 \end{cases} \quad (5.15)$$

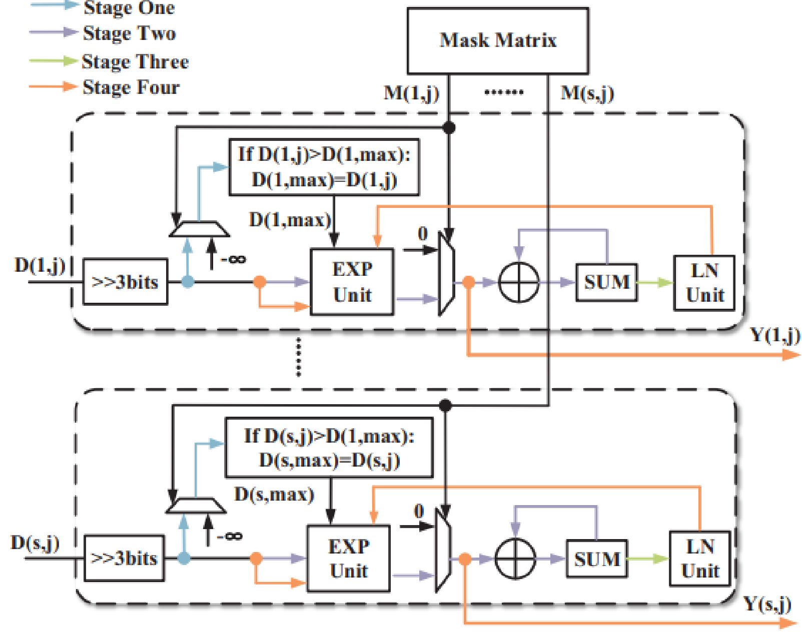


Figure 5.11: Softmax implementation with each stage datapath highlighted [33].

with this transformation the exponential function only has to be applied in the range of w, therefore it can be approximated with a LUT or with linear function, in the case of [33] it uses the linear approximation $2^w = w + d$ where d can be two different values one is used for the exp() of individual inputs and the second value is used for the accumulated result. The architecture implementing (5.15) the decomposed base 2 exp() function (5.15) is shown in Figure 5.12. Notice that there is another adder of $\ln F$ where F is the result of all the $\ln(\sum_{j=1}^{d_k} \exp(x_j - x_{\max}))$, which is used in the step four of [33] softmax architecture.

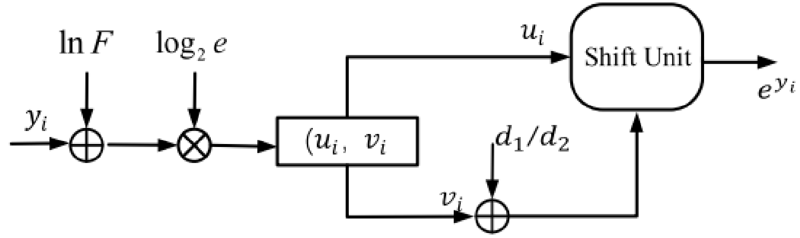


Figure 5.12: Base two exponential function architecture [58]

In the case of the ln() function the first reformulation implies that F is a value greater than one and can be represented as $F = 2^w \cdot k$ where w is a signed integer

representing the first leading one index (index has negative value if it is below the fixed point) and k is F shifted w times, making $k \in [1,2)$. Replacing F in $\ln(F)$ yields the formula (5.16). Given the interval of k $\log_2 k$ can be approximated to $\log_2 k \approx k - 1$, which greatly simplifies the logic, applying this expression into (5.16) gets a hardware friendly ln() approximation, as formula (5.17) shows. The architecture of the approximated ln() of (5.17) is shown in Figure 5.13 where LOD module obtains the leading one of F.

$$\ln(F) = \ln 2 \cdot \log_2 F = \ln 2 \cdot (w + \log_2 k) \tag{5.16}$$

$$\ln(F) \approx \ln 2 \cdot (k - 1 + w) \tag{5.17}$$

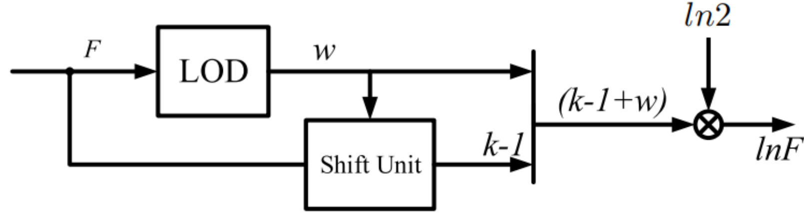


Figure 5.13: Base two exponential natural logarithm architecture [58]

[44] implements an accelerator for the attention mechanism of a binary transformer, allowing for a simpler softmax module that exploits the fact that its outputs can be only one or zero. this is done by rounding values greater than 0.5 to one and otherwise setting them to zero, Since the sum of all QK^T rows gets rounded to 1, only the maximum value in a row may get a softmax result greater than 0.5 which then gets rounded to one. therefore the softmax unit only has to perform the accumulation of the exponential results and obtaining the maximum value to then determine if the result will be greater than 0.5 meaning that division is not needed. For the exp() a Taylor series expansion is used. Figure 5.14 shows the architecture employed to calculate the softmax function. It is divided into two stages: the first one computes the the exponential accumulation and gets the index and value of the maximum value, the second stage consist of comparing the sum and the maximum multiplied by two and comparing them thus computing the output of the softmax module.

[49] is another work that changes the behaviour of the softmax layer since it implements a modified transformer called cosFormer [59], where the softmax layer consist of divided each row element by the row-wise sum of elements, thus dropping the $exp()$ function altogether.

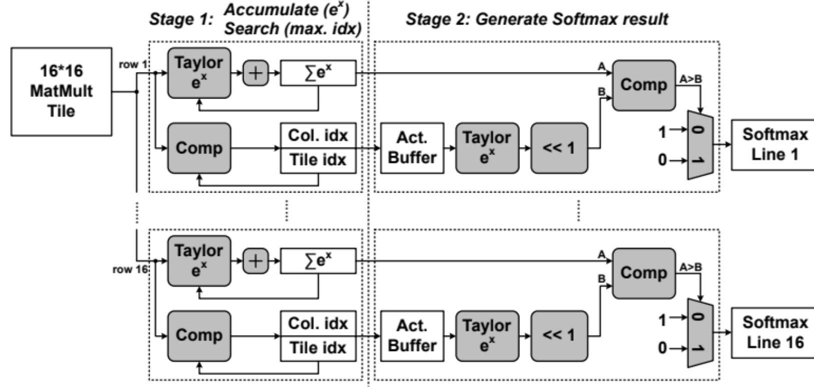


Figure 5.14: Binary softmax architecture [44].

5.2.3 LayerNorm

Being the second non-linear function belonging to the Transformer, LayerNorm function is challenging to implement since it's inputs are data dependent between each other. Apart from this the computation of the mean and variance,and also the presence of square root makes this particular function resource hungry and not straight forward to parallelized.

[41] switches the LayerNorm (1.5) layers with the BatchNorm layers, taking away the need to compute $E[x]$, Var$[x]$, square root, and division, since $E[x]$ and $Var[x]$ become static parameters. The former statements are reflected by making $\hat{\gamma} = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}}$ and $\hat{\beta} = \beta - \frac{E[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}$ then replacing these expressions into (1.5) yields formula (5.18) which greatly reduces the complexity of the Transformer model when accelerating inference.

$$y = \hat{\gamma}x + \hat{\beta} \tag{5.18}$$

Given that [41] accelerates transformers using integer quantization, $\hat{\gamma}$ and $\hat{\beta}$ are also quantized meaning this parameters have their own scale factor ($S_{\hat{\gamma}}; S_{\hat{\beta}} = S_{\hat{\gamma}}S_x$) and zero-point ($Z_{\hat{\gamma}}, Z_{\hat{\beta}}$) respectively. Taking into account quantization in (5.18) this expression becomes 5.19.

$$y \approx \frac{S_{\hat{\gamma}} \cdot S_x}{S_y} \cdot (\hat{\gamma} - Z_{\hat{\gamma}})((x_q - Z_x) + \hat{\beta}) + Z_y \tag{5.19}$$

[41] implements again the same structure as [56] ASIC, this time the only transformation done to the operation is the usage of integer square root to keep all the operations in the accelerator of the same domain. to compute integer square root the accelerator performs an iterative algorithm base on newtons method and

only requires integer arithmetic. This algorithm is computationally lightweight, as it converges within at most four iterations for any INT32 inputs and each iteration consists only of one integer division, one integer addition, and one bit-shifting operation [57]. The rest of the the non-linear operations in LayerNorm such as division and square root are straightforwardly computed with integer arithmetic. The aforementioned algorithm is shown in Algorithm 2. The overall architecture of the Layer norm module is displayed in Figure 5.15.

Algorithm 2 Integer-only Square Root [57]

Require: n: input integer
Ensure: Integer square root of n, i.e., $\lfloor \sqrt{n} \rfloor$
1: **function** I-SQRT(n)
2:     **if** $n = 0$ **then**
3:         **return** 0
4:     **end if**
5:     Initialize x_0 to $2^{\lceil \text{dBits}(n)/2 \rceil}$ and i to 0
6:     **repeat**
7:         $x_{i+1} \leftarrow \lfloor (x_i + \lfloor n/x_i \rfloor)/2 \rfloor$
8:         **if** $x_{i+1} \geq x_i$ **then**
9:             **return** x_i
10:         **else**
11:             $i \leftarrow i + 1$
12:         **end if**
13:     **until** termination
14: **end function**

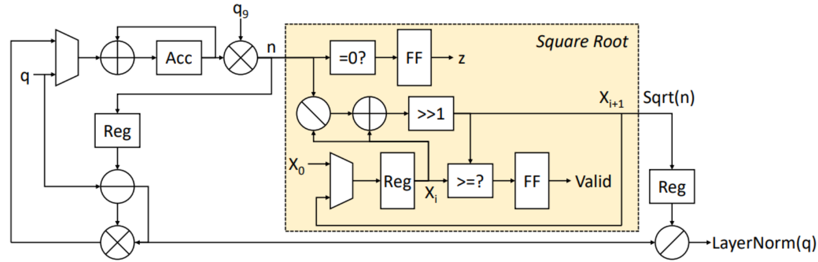


Figure 5.15: LayerNorm architecture with integer square root[56]

[33] goes into detail on how the accelerator executes the layer normalization operation, first it operates by streaming the results from the attention module to the layer norm module which operates concurrently for each row of the matrix. To speed-up the computation the module performs three operations concurrently: the

first operation consist of buffering the inputs for computing the LayerNorm outputs, the second operation consist in accumulating the inputs therefore obtaining $E[x]$, and the third operation consist of accumulating the square value of the inputs $\sum x^2/N$. When this three operations are finished the variance can be computed using the output of the latter two operations since $Var[x] = \sqrt{\sum x^2/N - E[x]}$ and finally once the variance for each row is obtained the final step consist of applying the formula (1.5) producing the output. Figure 5.16 shows the schedule and architecture of its layer norm module.

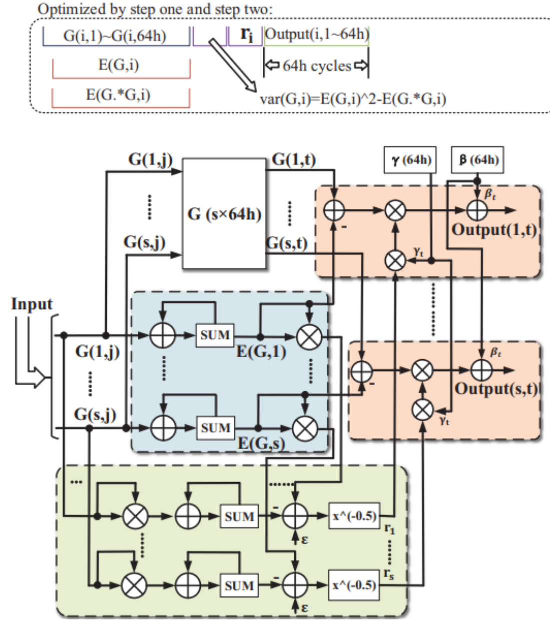


Figure 5.16: LayerNorm architecture and schedule [33].

5.3 Discussion

Having presented the diverse approaches that the reviewed works use to accelerate Transformer models, This chapter aims at summarizing the results found with their implementation. We group the reviewed papers into three categories based on how each consulted work compressed Transformer models. Table 5.1 groups the accelerators that where designed having in mind quantization schemes, whereas Table 5.2 the ones that employed pruning mechanism. It is apparent that FPGA-based accelerators for Transformer model is a very active research field, it becomes very hard to compare the performance of each model side by side. One of the reasons of this is that there is no standard benchmark that evaluates the performance of

each accelerator, leaving the designers to make their own ones. The second one is that each accelerator is deployed in different platforms making it difficult to determine if one accelerator is better to the other due to the fact that was tested on a platform with more resources or with an overall better performance.

Reference	Quantization scheme	Model	Latency (ms)	Throughput (GOP/s)	Target	Year
[32]	Activation INT16, parameters INT8	Multi Head Attention	-	1800	Alveo U250	2022
[33]	INT8 for parameters	Multi Head Attention	0.106	-	XCVU13P	2020
[34]	INT8 for parameters	Matrix-Vector multiplication	-	4696	ZCU102	2024
[35]	INT8 for parameters	Multi Head Attention	0.494	623	Alveo U55C	2024
[36]	INT8 for parameters and lower bit for Q and K	Encoder	-	3600	Alveo U280	2022
[37]	Activation INT8, parameters INT8	Multi Head Attention	4.63	-	XCVU13P	2022
[38]	Activation INT8, parameters INT8	Multi Head Attention	23.79	-	ZCU111	2021
[39]	Activation INT16, Parameters INT8	Attention Head	14.97	10.91	VC709	2022
[40]	employs uniform and non-uniform **PoT** quantization	LayerNorm and Linear ops	-	1970	ZCU102	2022
[42]	Hessian row-wise to quantize between 1/8-bit	Encoder	1.77	-	ZCU102	2024
[43]	Activations FP16, parameters INT4	Multi Head Attention	0.379	-	VCU128	2024
[41]	Uniform quantization INT8	Multi Head Attention	2.82	-	XC7S15	2024
[51]	Uniform quantization INT8	BERT	6.36	1223	Alveo U280	2024

Table 5.1: Comparison between Quantized or integer arithmetic accelerator

Reference	Pruning scheme	Model	Latency (ms)	Throughput (GOP/s)	Target	Year
[45]	block-wise balanced pruning	Multi Head Attention	-	695	ZCU102	2024
[46]	$N : M$ sparsity method	Transformer	0.15	1466	XCVU13P	2022
[47]	column-based pruning	Encoder	1.152	-	Alveo U200	2021
[48]	block-wise balanced pruning	Encoder/Decoder	7.85	-	Alveo U200	2021
[49]	unstructured pruning	cosFormer	-	28200	VC709	2024
[50]	dynamic token pruning and static weight pruning	Vit	0.868	-	Alveo U250	2024

Table 5.2: Comparison between accelerators that exploit sparsity patterns

5.4 Conclusions

In this work we presented an overview of artificial inteligence and machine learning, then explained the general principles behind Transformer models and made emphasis in the impact that this breakthrough had in the field, and why it is necessary to accelerate transformer applications. We also showed the techniques employed to reduce the size and complexity of the model, mainly by means of quantization and pruning. Furthermore we presented the general structure of FPGAs with their qualities and limitations, and proposed a design flow to accelerate machine learning models. Finally we presented the state of the art FPGA accelerators for transformers and explained the reasoning behind their architectural choices.

Bibliography

[1] Thomas M.. Mitchell. *Machine learning*. McGraw-Hill, 1997, p. 352. ISBN: 0070428077 (cit. on pp. 1, 2).

[2] Michael Nielsen. *Neural Networks and Deep Learning*. URL: http://neuralnetworksanddeeplearning.com (cit. on p. 4).

[3] Soumith Chintala. *Deep Learning with PyTorch: A 60 Minute Blitz*. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html. [Accessed 22-08-2024] (cit. on p. 5).

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 6).

[5] Larry Hardesty. *Explained: Neural networks*. https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414. [Accessed 24-08-2024]. 2014 (cit. on p. 6).

[6] Keras Team. *Keras documentation keras.io*. https://keras.io/api/layers/. [Accessed 24-08-2024] (cit. on p. 7).

[7] Keiron O'shea and Ryan Nash. «An introduction to convolutional neural networks». In: *arXiv preprint arXiv:1511.08458* (2015) (cit. on p. 7).

[8] Larry R Medsker, Lakhmi Jain, et al. «Recurrent neural networks». In: *Design and Applications* 5.64-67 (2001), p. 2 (cit. on p. 8).

[9] Michael Franke and Judith Degen. «The softmax function: Properties, motivation, and interpretation». In: (2023) (cit. on p. 9).

[10] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. «Layer normalization». In: *arXiv preprint arXiv:1607.06450* (2016) (cit. on p. 9).

[11] Ketan Doshi. *Batch Norm Explained Visually How it works, and why neural networks need it — towardsdatascience.com*. https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739. [Accessed 25-08-2024] (cit. on p. 9).

[12] A Vaswani. «Attention is all you need». In: *Advances in Neural Information Processing Systems* (2017) (cit. on pp. 10, 11, 13–15).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «Bert: Bidirectional encoder representations from transformers». In: *arXiv preprint arXiv:1810.04805* (2018) (cit. on p. 15).

[14] Wassen Aldjanabi, Abdelghani Dahou, Mohammed A.A. Al-Qaness, Mohamed Abd Elaziz, Ahmed Mohamed Helmi, and Robertas Damaševičius. «Arabic offensive and hate speech detection using a cross-corpora multi-task learning model». In: *Informatics* 8 (4 Dec. 2021). ISSN: 22279709. DOI: 10.3390/informatics8040069 (cit. on pp. 15, 16).

[15] Gokul Yenduri et al. «Gpt (generative pre-trained transformer)–a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions». In: *IEEE Access* (2024) (cit. on p. 16).

[16] Alexey Dosovitskiy. «An image is worth 16x16 words: Transformers for image recognition at scale». In: *arXiv preprint arXiv:2010.11929* (2020) (cit. on pp. 16, 18).

[17] Geoffrey Hinton. «Distilling the Knowledge in a Neural Network». In: *arXiv preprint arXiv:1503.02531* (2015) (cit. on p. 19).

[18] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. «Block pruning for faster transformers». In: *arXiv preprint arXiv:2109.04838* (2021) (cit. on p. 19).

[19] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. «Post-training quantization for vision transformer». In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 28092–28103 (cit. on p. 19).

[20] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. «A white paper on neural network quantization». In: *arXiv preprint arXiv:2106.08295* (2021) (cit. on p. 19).

[21] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. «A comprehensive survey on model quantization for deep neural networks». In: *arXiv preprint arXiv:2205.07877* (2022) (cit. on p. 19).

[22] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. «Integer quantization for deep learning inference: Principles and empirical evaluation». In: *arXiv preprint arXiv:2004.09602* (2020) (cit. on p. 20).

[23] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. «A Survey on Deep Neural Network Pruning: Taxonomy, Comparison, Analysis, and Recommendations». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024) (cit. on pp. 21, 22).

[24] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. «Snip: Single-shot network pruning based on connection sensitivity». In: *arXiv preprint arXiv:1810.02340* (2018) (cit. on p. 21).

[25] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. «Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning». In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 907–922 (cit. on p. 22).

[26] Jialin Cao, Xuanda Lin, Manting Zhang, Kejia Shi, Jun Yu, and Kun Wang. «PP-Transformer: Enable Efficient Deployment of Transformers Through Pattern Pruning». In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE. 2023, pp. 1–9 (cit. on p. 22).

[27] Krishna Teja Chitty-Venkata, Sparsh Mittal, Murali Emani, Venkatram Vishwanath, and Arun K Somani. «A survey of techniques for optimizing transformer inference». In: *Journal of Systems Architecture* (2023), p. 102990 (cit. on p. 23).

[28] Andrew Boutros and Vaughn Betz. «FPGA architecture: Principles and progression». In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29 (cit. on p. 24).

[29] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. «FPGA-based accelerators of deep learning networks for learning and classification: A review». In: *ieee Access* 7 (2018), pp. 7823–7859 (cit. on p. 24).

[30] mohd nazrin md isa mohd nazrin and Sohiful Anuar Zainol Murad. «Field Programmable Gate Array (FPGA): From Conventional to Modern Architectures». In: Jan. 2015, p. 53. ISBN: 978-967-5415-98-2 (cit. on p. 25).

[31] `https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/gen2/ai-edge-series.html`. [Accessed 21-09-2024] (cit. on p. 26).

[32] Wenhua Ye, Xu Zhou, Joey Zhou, Cen Chen, and Kenli Li. «Accelerating Attention Mechanism on FPGAs based on Efficient Reconfigurable Systolic Array». In: *ACM Transactions on Embedded Computing Systems* 22 (6 Nov. 2023). ISSN: 15583465. DOI: `10.1145/3549937` (cit. on pp. 29, 32, 33, 46).

[33] Siyuan Lu, Meiqi Wang, Shuang Liang, Jun Lin, and Zhongfeng Wang. «Hardware Accelerator for Multi-Head Attention and Position-Wise Feed-Forward in the Transformer». In: *International System on Chip Conference*. Vol. 2020-September. IEEE Computer Society, Sept. 2020, pp. 84–89. ISBN: 9781728187457. DOI: `10.1109/SOCC49529.2020.9524802` (cit. on pp. 29, 32, 33, 38, 40, 41, 44–46).

[34] Han Xu, Yutong Li, and Shihao Ji. «LlamaF: An Efficient Llama2 Architecture Accelerator on Embedded FPGAs». In: (Sept. 2024). URL: `http://arxiv.org/abs/2409.11424` (cit. on pp. 29, 46).

[35] Ehsan Kabir, Md. Arafat Kabir, Austin R. J. Downey, Jason D. Bakos, David Andrews, and Miaoqing Huang. «FAMOUS: Flexible Accelerator for the Attention Mechanism of Transformer on UltraScale+ FPGAs». In: (Sept. 2024). URL: http://arxiv.org/abs/2409.14023 (cit. on pp. 29, 46).

[36] Hongwu Peng et al. «A length adaptive algorithm-hardware co-design of transformer on FPGA through sparse attention and dynamic pipelining». In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. Vol. 33. DAC '22. ACM, July 2022, pp. 1135–1140. DOI: 10.1145/3489517.3530585. URL: http://dx.doi.org/10.1145/3489517.3530585 (cit. on pp. 29, 46).

[37] Binjing Li, Siyuan Lu, Keli Xie, and Zhongfeng Wang. «Accelerating NLP Tasks on FPGA with Compressed BERT and a Hardware-Oriented Early Exit Method». In: *Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI*. Vol. 2022-July. IEEE Computer Society, 2022, pp. 410–413. ISBN: 9781665466059. DOI: 10.1109/ISVLSI54635.2022.00092 (cit. on pp. 29, 32, 33, 46).

[38] Zejian Liu, Gang Li, and Jian Cheng. «Hardware acceleration of fully quantized bert for efficient natural language processing». In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2021, pp. 513–516 (cit. on pp. 29, 35, 36, 46).

[39] Zhongyu Zhao, Rujian Cao, Ka Fai Un, Wei Han Yu, Pui In Mak, and Rui P. Martins. «An FPGA-Based Transformer Accelerator Using Output Block Stationary Dataflow for Object Recognition Applications». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 70 (1 Jan. 2023), pp. 281–285. ISSN: 15583791. DOI: 10.1109/TCSII.2022.3196055 (cit. on pp. 29, 35, 46).

[40] «Auto-ViT-Acc: An FPGA-Aware Automatic Acceleration Framework for Vision Transformer with Mixed-Scheme Quantization». In: *Proceedings - 2022 32nd International Conference on Field-Programmable Logic and Applications, FPL 2022*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 109–116. ISBN: 9781665473903. DOI: 10.1109/FPL57034.2022.00027 (cit. on pp. 29, 46).

[41] Tianheng Ling, Chao Qian, and Gregor Schiele. «Integer-only Quantized Transformers for Embedded FPGA-based Time-series Forecasting in AIoT». In: (July 2024). URL: http://arxiv.org/abs/2407.11041 (cit. on pp. 29, 30, 36, 43, 46).

[42] Woohong Byun, Jongseok Woo, and Saibal Mukhopadhyay. «Hardware-friendly Hessian-driven Row-wise Quantization and FPGA Acceleration for Transformer-based Models». In: Association for Computing Machinery (ACM),

Aug. 2024, pp. 1–6. DOI: 10.1145/3665314.3670806 (cit. on pp. 29, 32, 33, 37, 38, 46).

[43] Mingqiang Huang, Ao Shen, Kai Li, Haoxiang Peng, Boyu Li, and Hao Yu. *EdgeLLM: A Highly Efficient CPU-FPGA Heterogeneous Edge Accelerator for Large Language Models* (cit. on pp. 29, 32, 33, 46).

[44] Congpeng Du, Seok Bum Ko, and Hao Zhang. «Energy Efficient FPGA-Based Binary Transformer Accelerator for Edge Devices». In: *Proceedings - IEEE International Symposium on Circuits and Systems*. Institute of Electrical and Electronics Engineers Inc., 2024. ISBN: 9798350330991. DOI: 10.1109/ISCAS58744.2024.10558631 (cit. on pp. 29, 34, 35, 42, 43).

[45] Saiqun Wang and Hao Zhang. «Efficient FPGA-Based Transformer Accelerator Using In-Block Balanced Pruning». In: Institute of Electrical and Electronics Engineers (IEEE), Sept. 2024, pp. 18–23. ISBN: 9798350386271. DOI: 10.1109/icccas62034.2024.10651591 (cit. on pp. 29–31, 46).

[46] Chao Fang, Aojun Zhou, and Zhongfeng Wang. «An Algorithm-Hardware Co-Optimized Framework for Accelerating N:M Sparse Transformers». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30 (11 Nov. 2022), pp. 1573–1586. ISSN: 15579999. DOI: 10.1109/TVLSI.2022.3197282 (cit. on pp. 29, 32, 33, 38, 40, 46).

[47] Hongwu Peng, Shaoyi Huang, Tong Geng, Ang Li, Weiwen Jiang, Hang Liu, Shusen Wang, and Caiwen Ding. «Accelerating Transformer-based Deep Learning Models on FPGAs using Column Balanced Block Pruning». In: *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. 2021, pp. 142–148. DOI: 10.1109/ISQED51717.2021.9424344 (cit. on pp. 29, 46).

[48] Panjie Qi, Yuhong Song, Hongwu Peng, Shaoyi Huang, Qingfeng Zhuge, and Edwin Hsing Mean Sha. «Accommodating Transformer onto FPGA: Coupling the Balanced Model Compression and FPGA-Implementation Optimization». In: *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*. Association for Computing Machinery, June 2021, pp. 163–168. ISBN: 9781450383936. DOI: 10.1145/3453688.3461739 (cit. on pp. 29, 46).

[49] Rujian Cao, Zhongyu Zhao, Ka Fai Un, Wei Han Yu, Rui P. Martins, and Pui In Mak. «An FPGA-Based Transformer Accelerator With Parallel Unstructured Sparsity Handling for Question-Answering Applications». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2024). ISSN: 15583791. DOI: 10.1109/TCSII.2024.3462560 (cit. on pp. 29, 42, 46).

[50] Dhruv Parikh, Shouyi Li, Bingyi Zhang, Rajgopal Kannan, Carl Busart, and Viktor Prasanna. «Accelerating ViT Inference on FPGA through Static and Dynamic Pruning». In: (Mar. 2024). URL: http://arxiv.org/abs/2403.14047 (cit. on pp. 29, 46).

[51] Hongji Wang, Yueyin Bai, Jun Yu, and Kun Wang. «TransFRU: Efficient Deployment of Transformers on FPGA with Full Resource Utilization». In: *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 521–526. ISBN: 9798350393545. DOI: 10.1109/ASP-DAC58780.2024.10473976 (cit. on pp. 30–32, 37, 46).

[52] Yonghao Chen, Tianrui Li, Xiaojie Chen, Zhigang Cai, and Tao Su. «High-Frequency Systolic Array-Based Transformer Accelerator on Field Programmable Gate Arrays». In: *Electronics (Switzerland)* 12 (4 Feb. 2023). ISSN: 20799292. DOI: 10.3390/electronics12040822 (cit. on pp. 32, 33).

[53] Bingbing Li et al. «FTRANS: Energy-efficient acceleration of transformers using FPGA». In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, Aug. 2020. ISBN: 9781450370530. DOI: 10.1145/3370748.3406567 (cit. on p. 33).

[54] Rodrigue Rizk, Dominick Rizk, Frederic Rizk, Ashok Kumar, and Magdy Bayoumi. «A Resource-Saving Energy-Efficient Reconfigurable Hardware Accelerator for BERT-based Deep Neural Network Language Models using FFT Multiplication». In: *Proceedings - IEEE International Symposium on Circuits and Systems*. Vol. 2022-May. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 1675–1679. ISBN: 9781665484855. DOI: 10.1109/ISCAS48785.2022.9937531 (cit. on p. 33).

[55] Hongxi Dong, Manzhen Wang, Yuanyong Luo, Muhan Zheng, Mengyu An, Yajun Ha, and Hongbing Pan. «PLAC: Piecewise Linear Approximation Computation for All Nonlinear Unary Functions». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.9 (2020), pp. 2014–2027. DOI: 10.1109/TVLSI.2020.3004602 (cit. on p. 37).

[56] Alberto Marchisio, Davide Dura, Maurizio Capra, Maurizio Martina, Guido Masera, and Muhammad Shafique. «SwiftTron: An efficient hardware accelerator for quantized transformers». In: *2023 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2023, pp. 1–9 (cit. on pp. 37, 38, 43, 44).

[57] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. «I-BERT: Integer-only BERT Quantization». In: *International Conference on Machine Learning (Accepted)* (2021) (cit. on pp. 37, 39, 44).

[58] Meiqi Wang, Siyuan Lu, Danyang Zhu, Jun Lin, and Zhongfeng Wang. «A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning». In: *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2018, pp. 223–226. DOI: 10.1109/APCCAS.2018.8605654 (cit. on pp. 40–42).

[59] Zhen Qin, Weixuan Sun, Hui Deng, Dongxu Li, Yunshen Wei, Baohong Lv, Junjie Yan, Lingpeng Kong, and Yiran Zhong. «cosFormer: Rethinking Softmax In Attention». In: *International Conference on Learning Representations*. 2022. URL: `https://openreview.net/forum?id=Bl8CQrx2Up4` (cit. on p. 42).