POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design and analysis of VLSI architectures for Transformers

Supervisors

Prof. Maurizio MARTINA

Prof. Guido MASERA

Prof. Alberto MARCHISIO

Candidate

Davide DURA

December 2022

Abstract

Neural networks have been a big innovation field recently, with more and more applications addressing Machine Learning algorithms. A big part of these is made of Natural Language Processing (NLP) algorithms, which handle words, sentences and group of sentences. Machine translation, text generation, sentiment analysis and question and answering are just some examples of the NLP tasks. In this scope, the model that has gained more popularity is clearly the Transformer, with its great adaptability to different objectives.
This network architecture is based on the attention mechanism and it has exceeded the performances of previously-used recurrent and convolutional neural networks. There are already several different models based on the Transformer: its encoder-decoder nature gives a lot of room to explore by changing the values of the parameters or the layer configuration. BERT (Bidirectional Encoder Representations from Transformers) and Universal Transformer network are two particular models derived from the Transformer. However, Transformer has a big structure and a lot of parameters and that's why any hardware implementation is difficult and expensive to realize. In fact these drawbacks translate into complex resources, great memory footprint and latency.
This work analyzes state-of-the-art situation on hardware realizations of the Transformer and proposes some ideas to design the network as a whole. Divide-and-conquer approach is used to design single layers and sub-layers in the architecture, but considerations on reusing resources and different structure possibilities are still taken into account. Quantization is key to have an integer-only architecture and to reduce both memory requirements and resources. Starting from an entirely-quantized model, the hardware design is developed for a single Encoder layer; it is legit to assume that different configurations can be realized by replicating the architecture.
Main focus is on the matrix multiplication and the non-linear functions. The former is the most important operation since it covers majority of the network computation, besides being heavy from the area point of view. To implement it, the choice is a matrix of Multiply-and-Accumulate (MAC) elements, which is simulated and synthetized for different dimensions to see the trend for estimate bigger structures. Non-linear functions on the other hand are complex due to the type of operations that they need. Linear algorithms approximating them are taken from literature and translated into hardware solutions, whose behaviour has been compared to software model to see the correctness of their results.
Connecting the separate sub-layers is duty of the control part of the design, which

is also described to see possible solutions. Eventually, adaptability of the design to other types of Transformer is evaluated.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

ML
Machine Learning

NLP
Natural Language Processing

BERT
Bidirectional Encoder Representations from Transformers

UT
Universal Transformer

MHSA
Multi-Head Self Attention

FFN
Feed-Forward Network

MatMul
Matrix Multiplication

PE
Processing Element

CU
Control Unit

Chapter 1

Introduction

1.1 Context and Problem Statement

Recently Natural Language Processing (NLP) algorithms have been executed by means of an innovative neural network called Transformer. This network has overcome some previously challenges and it has improved performances in quite all benchmarks. There are a lot of Transformer models, able to process at the best level several tasks, like text generation, sentiment analysis or machine translation. They are all based on the Transformer structure with small differences between them.

Transformer's architecture is based on the *attention* mechanism, the true foundation of its performances since it can extract the sentence's content and the relationship between words. The network is an Encoder-Decoder structure that firstly elaborates the input sentence and then analyses it. Encoder and Decoder are very similar, formed by a stack of base blocks that reuse a lot of the same operations. By changing the parameters of these two components, different models are generated, with different results. Further models are the BERT, which uses only the Encoder part, and GPT (Generative Pre-Training Transformer), based on the Decoder part. More recent works propose Universal Transformer, which substitutes the stack forming the Encoder and the Decoder with the iteration of the same block for a predetermined amount of times.

1.2 Research Objective

Transformer network is very intensive operation-wise, posing a real challenge for its acceleration on limited platforms. In fact it has many parameters and many operations that require several resources. Plus, exponential and non-linear functions make the implementation more difficult.

The objective is to find a way to design a hardware accelerator for Transformers, able to implement all of Transformer layers' operations. This is not easy as almost every Transformer model, for the sake of precision, use floating-point representation, which is inefficient in terms of memory footprint and resources. Complex operations slow down hardware execution and require a lot more components and data handling.

Looking at the literature, there are not many works with this objective; some hardware proposals keep the floating-point computation while others run their design on generalized hardware like GPUs or CPUs.

1.3 Thesis Contribution

This thesis proposes a design of each Transformer sub-layer, using mostly lightweight resources like multipliers and adders.

Quantization is used to reduce data into integer representation, simplifying operations and lighten the parameter burden. The quantized functions are based on a previous model named I-BERT[1]. Complex functions are approximated with polynomials to ease the computation, even though there's a little loss in the precision.

The design is described for all main layers of the Transformer architecture, starting from the top level going down to the single blocks. Matrices of Multiply-and-Accumulate are used for the most common operation in the network, that is the Matrix Multiplication. This is the biggest component because Transformer's dimensions can be large and this affects primarily the linear matrix transformations. Other than that, the other complex part is given by the non-linear functions, which are hard to design yet essential to the network's performance. Approximations proposed in the above-mentioned [1] are adopted and reported onto hardware following the entire network design concept.

These architectures are then simulated with respect to the software model and eventually synthesized. Area, power and timing results are reported and analysed to better understand the feasibility of such design.

1.4 Thesis Structure

Starting from an overview of the Transformer architecture and its different types, some state-of-the-art solutions were analysed to find some already-developed ideas and a reference model to build the design accordingly. Then the quantized model, which this work is based on, is presented function by function, along with their corresponding algorithm.

After these introductions, the main design is described, exploring the architectures component after component. Each sub-layer has its scheme and description

about the data handling and the computation flow. Furthermore, the synthesis results are extracted and briefly analysed with some considerations about them.

Once the architecture is designed, there are some thoughts on different possible configurations given that there are a lot of Transformer dimensions. In fact, the design can be changed to execute other Transformers and with some parametrization on the control part, it is legit to think that a configurable model can be realized. In conclusion, results and possible continuations of the work are discussed.

Chapter 2

Types of Transformer

2.1 Transformer

Transformers networks have recently become the go-to models in different NLP tasks such as machine translation or text generation. The main innovation compared to previously-used RNNs is the parallelization of the architecture. In fact, by definition RNNs process the data recurrently, where a computational step corresponds to positions in the sequence; this becomes critical at longer sequence lengths due to memory constraints. Correlation between distant sequence positions is also difficult to exploit analysing them one at a time, bringing to a necessity to parallelize the process.

Key mechanism of the network is the attention, the core of the whole network. Attention compute representations of the input and the output, highlighting the relation between words in the sequence. In particular, self-attention is used in the Transformer as the inputs of the attention are all taken from the common input through a matrix transformation.

2.1.1 Structure

The Transformer structure is formed by an encoder and a decoder, similar to each other. In figure 2.1 the structure taken from [2] is showed.

The encoder is in charge of taking the input sequence and exploit the relations between each sub-sequence under different criteria, representing these informations in order to be analyzed in the decoder phase.

Main components of both parts are the Multi-Head Self Attention (MHSA) and the Feed-Forward Network (FFN), explained below.

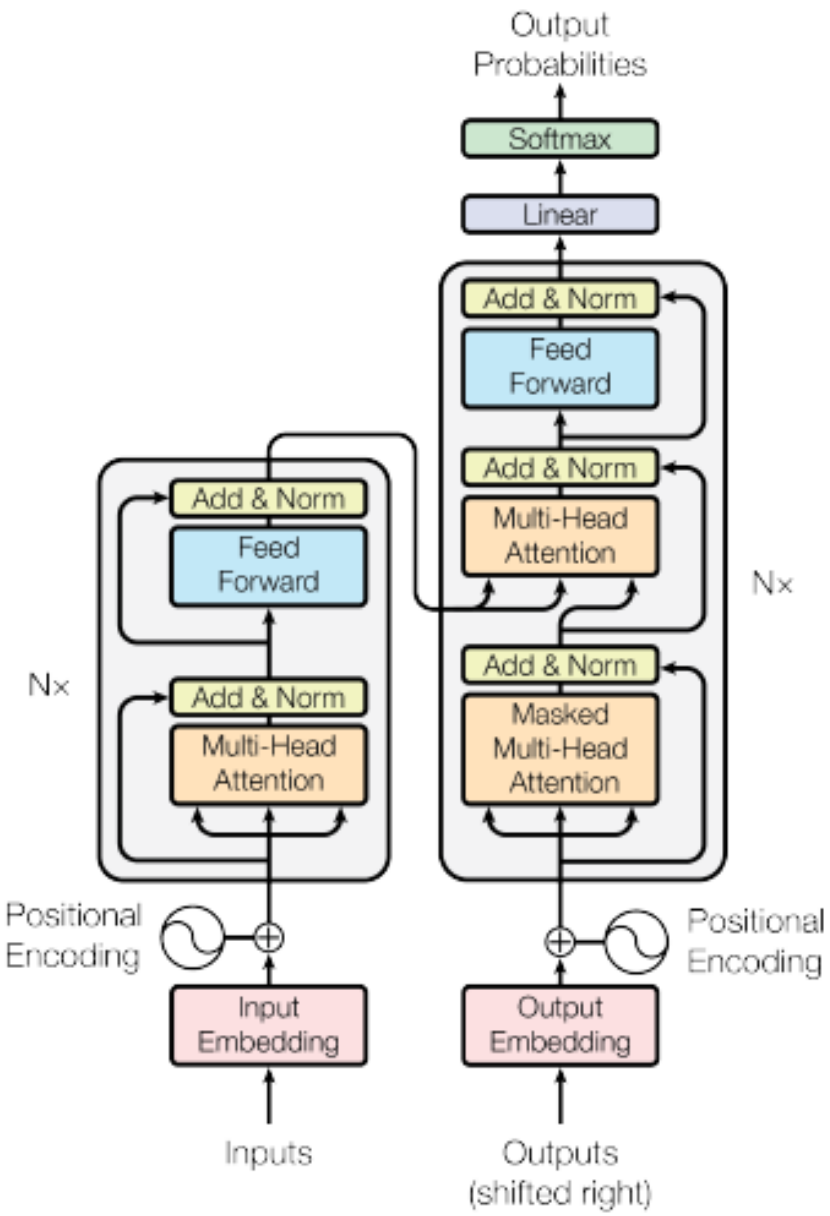

Figure 2.1: Transformer scheme

2.1.2 Input Representation

Starting from an input sentence, every word or sub-part of a word, that is named *token*, can be represented by a one-hot vector. This vector has dimension equal to

the vocabulary dimension ($d_{vocabulary}$) with a one only in the position corresponding to the token position in the vocabulary.

The available vocabulary denotes the recognizable tokens that can be interpreted by the network.

Since the vocabulary dimension can be too large, one-hot vectors are transformed into smaller vectors during what's called Word Embedding process. In fact, through a weight matrix multiplication an Embedding Vector is obtained, with d_{model} dimension. This vectors remove the sparse nature of the one-hot vectors and they can also carry some more information due to the trainable Embedding weight matrix. Different choices are possible for this weight matrix: it can be initialized with pre-trained values and kept fixed, or initialized randomly and trained during the network learning process.

Putting together all the token vectors, an input matrix is obtained and passed on to the network itself starting from the Positional Encoding. Dimension of this matrix is sequence length (here noted as m) per embedding dimension (d_{model}).

2.1.3 Positional Encoding

After the Embedding no information on the relative positions of the tokens inside the sentence is present; a word has the same embedding representation regardless if it's the first word or if it's the fourth one. That's why another input transformation is needed before entering the encoder/decoder structure.

Here stands one of the difference between Transformer and a RNN or CNN. Positional Encoding becomes necessary having no recurrence and processing several tokens at the same time. In a RNN, that by definition processes one token at a time, sentence position is implied by the processing step.

Also this step can be learned or fixed, reaching similar performances according to [2]. The choice of fixed Positional Encoding avoid the presence of additional parameters and allow flexibility to longer sequences than the one exploited in the training phase.

To have fixed values, an equation is required and that's the one in 2.1.

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{model}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{model}}\right) \end{aligned} \tag{2.1}$$

pos is the token position and i is the dimension. The results of this operation have the same dimensions of the input matrix, so the two can be summed together to obtain the final matrix ($m \times d_{model}$).

2.1.4 Encoder

It is made of k identical layers connected to each other. Each encoder layer has two sub-layers, namely the Multi-Head Self Attention and the Feed-Forward Network.

Multi-Head Self Attention As underlined before, the attention is the crucial part of the whole network. It is called self-attention because the attention components are all taken from the same input.

In this sublayer the attention itself is the inner part, as the computation is divided in k heads each going through the attention to form the final result.

The idea behind having multiple heads is to have different representation subspaces for different positions; if there was a single attention this could not be possible.

Input X of this MHSA in the encoder is the output of the previous encoder layer. This input X is transformed into three smaller matrix representations through a linear layer, that is a multiplication by a weight matrix (see 2.3, 2.2, 2.4). These three matrices are called Key (K), Value (V) and Query (Q).

$$K = X * W^K \quad W^K \in \mathbb{R}^{m \times d_K} \tag{2.2}$$

$$Q = X * W^Q \quad W^Q \in \mathbb{R}^{m \times d_K} \tag{2.3}$$

$$V = X * W^V \quad W^V \in \mathbb{R}^{m \times d_V} \tag{2.4}$$

So, instead of a single d_{model} dimensional query, key and value, there are multiple versions of these matrices with reduced dimensions. Value matrix can have different dimension with respect to Query and Key, but it is convenient to assume d_V equal to d_K. Furthermore, d_K is usually equal to d/k.

After obtaining Queries, Keys and Values, the attention of each head can be computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.5}$$

The first step is to multiply Queries and Keys together, with the dot products represented as a matrix multiplication (transposing K matrix). The result is then scaled by a reducing factor to avoid having huge numbers after products, especially increasing the d_k dimension. That's also why the square root of this value is chosen as the reducing factor.

Once all heads attentions are computed, the results are concatenated together and passed to another linear combination that produces the final output of the MHSA sublayer, as described in the formula 2.6.

$$MultiHead(Q, K, V) = Concat(head_1, head_2, ..., head_k)W^0$$
$$where\ head_i = Attention(QW_i^k, KW_i^K, VW_i^V)$$
$$with\ W^0 \in \mathbb{R}^{d_{model} \times d_{model}} \quad (2.6)$$

Figure 2.2 depicts the scheme of the MHSA sublayer and its Attention component.

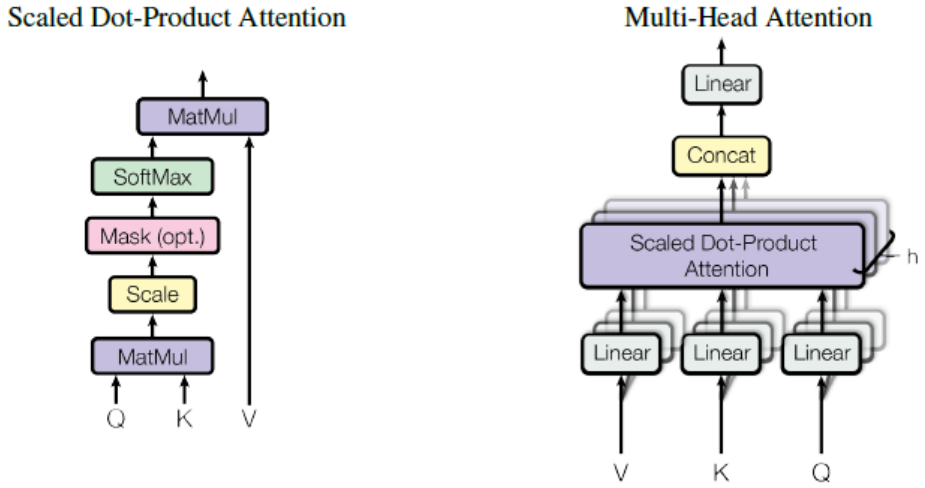

Figure 2.2: Attention (left) and Multi-Head Self-Attention (right) schemes

Softmax Another critical operation of the MHSA is the Softmax, whose equation is 2.7. Given a N-vector of values z_j, the softmax espression is:

$$Softmax(z_j) = \frac{\exp z_j}{\sum_{k=1}^{N} \exp z_k} \quad per\ j = 1, ..., N \quad (2.7)$$

The sum at the denominator in the Attention is performed along the rows of the QK^T matrix, meaning the N value corresponds to d_{model}.

The effect of this mathematical operation is to transform input values into a range between 0 and 1. In the Transformer the softmax outputs represent the probability associated to each token and focusing on the attention utilization, it is the percentage of relation between the tokens.

Softmax, as can be seen in scheme 2.1, is used also at the output of the network, transforming the raw numbers exiting the computation in percentage values in order to pick the highest ones (i.e. the most probable) in the selection of the output phrase.

Exponential operations required in this part of the network can be critical and expensive for an hardware implementation, and this will be analyzed later.

Feed-Forward Network Simpler sublayer (described in 2.3) with respect to the Multi-Head Self Attention, it is formed by two linear transformation with a ReLU activation in between. Mathematical expression is reported in 2.8.

$$FFN(x) = ReLU(xW_1 + b_1)W_2 + b_2 \quad where\ ReLU(x) = max(0, x) \tag{2.8}$$

with $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$ and $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$. d_{ff} is named the *hidden dimension* and it is usually a multiple of d_{model}, e.g. $2\times$ or $4\times$.

ReLU is a simple activation layer, that is in charge of keeping only the useful parts, namely activating them.

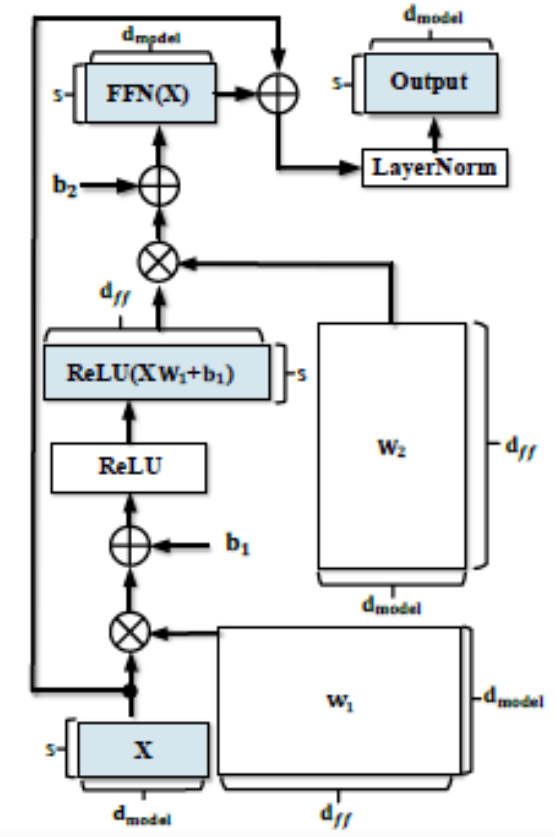

Figure 2.3: Feed-Forward Network illustrating scheme

Residual Connection These two sub-layers compute only the residual values that have to be added to the input to obtain the final values. Putting it in an

expression, 2.9 represents this passage.

$$Output = LayerNorm(x + Sublayer(x)) \tag{2.9}$$

So, after each sub-layer (MHSA or FFN) the result is added to the input and passed through a Layer Normalization. This operation is always the same dimensionally-speaking, as all sub-layers (and the Positional Encoding at the beginning) produce matrices $m \times d_{model}$. In figure 2.4 there's the graphical scheme of this operation to better understand the connection. In this network "Residual" block are the MHSA or the FFN.

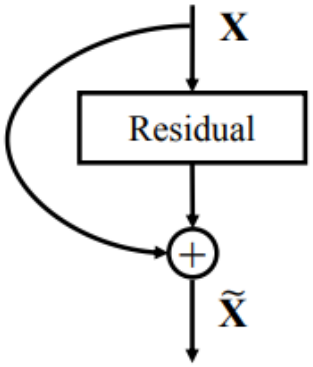

Figure 2.4: Residual connection illustration

Layer Normalization Normalization consists in standardizing the inputs, that means that inputs to any layer should have approximately zero mean and unit variance. Before Layer Normalization ([3] [4]), Normalization along batches was used, but it implies dependencies between training cases and that's not easy to apply to a recurrent network. Layer Normalization on the other end allows to make normalization within a hidden layer on a single training case, as the statistics are calculated from the layer output.

It is proved that this passage improves the training speed and the stability of the state dynamics.

2.10 is the formula for the Layer Normalization.

$$LayerNorm(X(i,j)) = \frac{X(i,j) - E[X,i]}{\sqrt{var[X,i] + \epsilon}}\gamma_j + \beta_j \tag{2.10}$$

As hinted above, mean (*E[X]*) is subtracted to inputs and standard deviation, or the square root of the variance (*var[X]*), is used as the divisor.

$$E[X,i] = \frac{1}{d}\sum_{k=1}^{d} X(i,k) \quad var[X,i] = \frac{1}{d}\sum_{k=1}^{d}(X(i,k) - E[X,i])^2 \tag{2.11}$$

2.1.5 Decoder

Similarly to the Encoder, it is formed by N identical layers, but these layers have one additional sublayer.

In fact, between the Multi-Head Self Attention and the Feed-Forward Network there is the Encoder-Decoder Attention.

Encoder-Decoder Attention It is identical to the MHSA speaking of required operation, the difference stands in the Q, K, V computation.

Queries are still obtained from the input X while Keys and Values are obtained from the Encoder output.

This allows the output sentence that is processed in the Decoder to address the input sentence informations produced in the Encoder in order to find the correct tokens to compose the final sentence.

Another important difference is in the Decoder inputs handling, that are actually the outputs of the network. Since at the position i tokens should not depend on future positions but only to the ones up to i, the Decoder inputs are shifted to the right by one position and the MHSA is masked. Masking the MHSA means that, before the Softmax, results of the QK^T multiplication are filtered by positions. Values at positions after i are substituted with a 0. This is described in scheme 2.2.

2.2 Universal Transformer

Motivations Generalized tasks do not perform well with the Transformer network, e.g. algorithmic tasks like strings copy, reverse and addition. So to have also this possibility without giving up the Transformer features, the Universal Transformer has been proposed. This network combines the parallelizability of the Transformer with the recurrent inductive bias of the RNNs.

Main change, indeed, is the fact that Universal Transformer has only one Encoder sub-layer and one Decoder sub-layer that are used continuously in a fixed or adaptive iteration. Sub-layers are identical to the Transformer so it can be said that a Universal Transformer with N iterations is equal to a Transformer with N Encoder and N Decoder layers with all of them having the same parameters.

That's the concept variation with respect to RNNs; in fact Recurrent Neural Networks recur over positions in the sequence while Universal Transformer recurs over vector representations of each position, namely over "depth". This way, the

iterations are not bounded to the number of positions in the sequence and the representations can be revised an arbitrary amount of times. Adaptive methods are also allowed, as explained later in the Dynamic Halting paragraph.

Structure Changes As already said, there are only one Encoder and one Decoder layer, identical to the Transformer ones. Structure is depicted in figure 2.5.

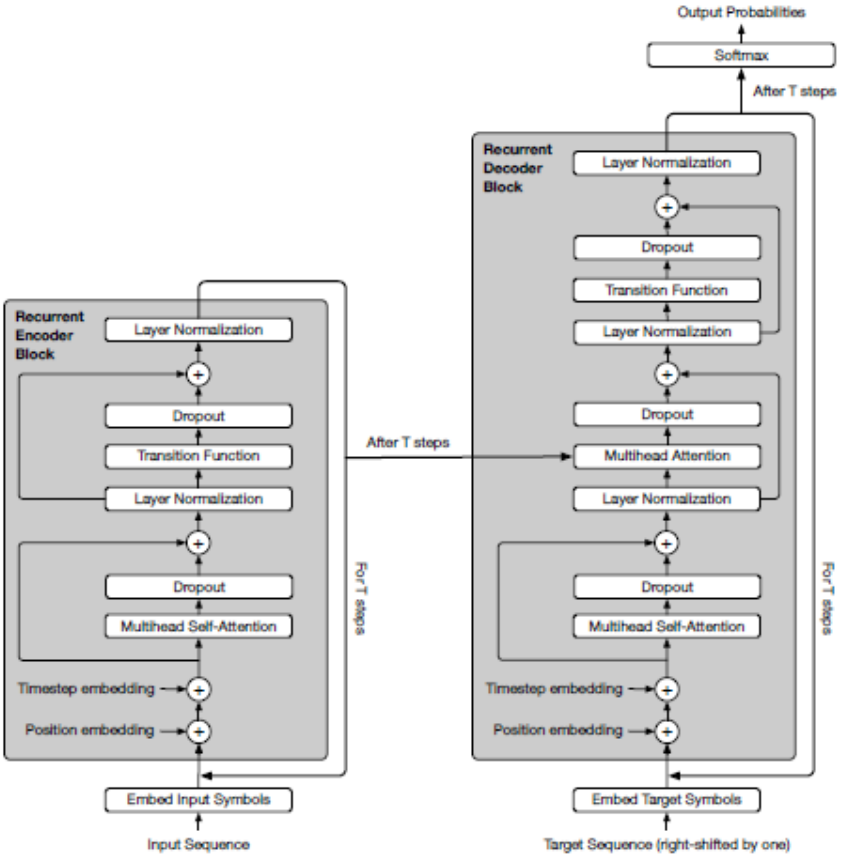

Figure 2.5: Universal Transformer structure

So the main components of the network have been already explained: Multi-Head Self Attention and Encoder-Decoder Attention are identical to the Transformer one as it is the Positional Encoding.

Transition Function, according to the paper [5] can be either a Separable Convolution or the above-mentioned Feed-Forward Network, with choice depending on the task.

The new step is the Time-Step Encoding process after the Positional Encoding one.

Time-Step Encoding The principle is similar to the Positional Encoding; in fact without the stack of encoding and decoding layers the information on the processing step can be lost. For that, a very similar Encoding is added, described by the formulas 2.12.

$$\begin{aligned} TimeEnc_{(t,2i)} &= \sin\left(t/10000^{2i/d_{model}}\right) \\ TimeEnc_{(t,2i+1)} &= \cos\left(t/10000^{2i/d_{model}}\right) \end{aligned} \tag{2.12}$$

The two encodings can be seen together and merged into one single expression, 2.13.

$$\begin{aligned} P^t_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{model}}\right) + \sin\left(t/10000^{2i/d_{model}}\right) \\ P^t_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{model}}\right) + \cos\left(t/10000^{2i/d_{model}}\right) \end{aligned} \tag{2.13}$$

$P^t \in \mathbb{R}^{m \times d_{model}}$ has same dimensions of the inputs so it is simply added to have the final representations going into the Encoder or the Decoder layer.

Universal Transformer is autoregressive, producing one output symbol at a time, with the Decoder feeded at every step with the previously computed symbols.

Dynamic Halting Reasonably, in a sentence there are some parts more critical than others and therefore they can require more processing than the simpler ones. Taking advantage of the iterating nature of this network, Dynamic Halting addresses this possibility, trying to adapt the recursion to the symbols necessity.

The mechanism to apply this principle is called Adaptive Computation Time (ACT): it adapts the computational steps to process each input symbol basing on a "halting probability". This probability represents how much the symbol is close to been interpreted, and with the help of a threshold it decides which ones have to be stalled and which ones needs further processing. Halting probability is predicted and updated at every step; positions that reach the halting threshold are blocked and their state is copied onto each successive step. When every step reach the blocking probability or after a predefined maximum number of steps the computation is finished.

Pros and Cons According to the paper, Universal Transformer can improve performances in some Natural Language Processing tasks with respect to the Transformer, having as well state of the art performances in algorithmic tasks. These are great pros to the network, but there are also some disadvantages. Most of them are the same of the Transformer, like intensive computation (many matrix processing), high latency due to the recurrent nature and large number of parameters that lead to big memory usage. Moreover, to apply ACT, further resources and processing are required.

2.3 BERT

BERT, *Bidirectional Encoder Representations from Transformers*[6], is a Transformer based model for NLP tasks. It was developed by *Google* and it is used a lot, in different configurations.

Major difference from the Transformer model is the fact that BERT is composed only by Encoder layers. That's because it is a language representation generator, so it needs to analyze only the relationships between the input tokens, without any type of decoding. In fact, for example, this model cannot be used in Machine Translation tasks as it was for the Transformer.
Key training tasks of BERT model are the *Masked Language Model* and the *Next Sentence Prediction*. In the former some input tokens are substituted with the mask token and the network has to complete this spaces with a word in the correct context. The latter is about sentences and the temporal relationship between them. For instance, starting from two sentences the model has to identify which one comes before the other one.
BERT is *Bidirectional* because it receives both left part and right part of the sentence with respect to the unknown element, so it has the whole context in which inserting the guessed token and it can work better than unidirectional models. This are not the only tasks that BERT can be used on, since it is adaptable to many objectives just like the Transformer. In fact after the pre-training phase, BERT can be fine-tuned on other tasks and it advances state-of-the-art performances in eleven NLP tasks, as stated in [6].

Structure BERT *base* and *large* structures are depicted in figure 2.6. As already said, BERT has only the Encoder block of a Transformer and therefore is formed by a certain number of stacked Encoders.

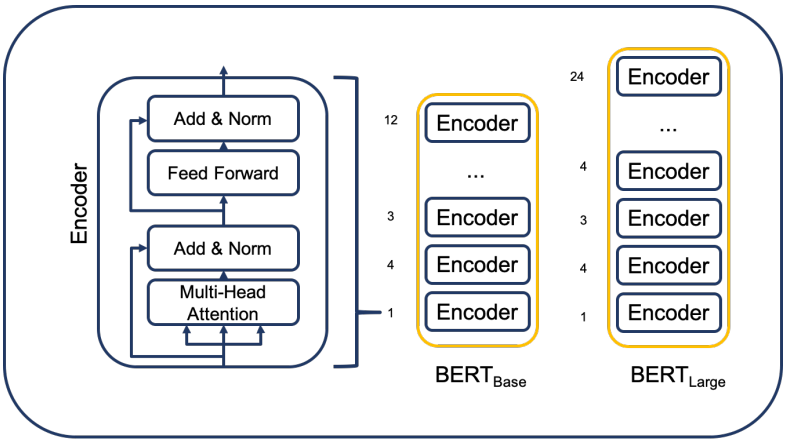

Figure 2.6: BERT architecture

One thing that differs from the previous Encoder description is that in the BERT model the activation function is the GELU; it substitutes the ReLU due to a better behaviour.

GELU GELU stands for *Gaussian Error Linear Unit* and it is an activation function used in the neural models, proposed in paper [7]. Differently from the above-mentioned ReLU that selected inputs by their sign, GELU weights input by their value. GELU is derived from the ReLU, but instead of multiplying the input by zero or one based on the input sign, it is multiplied by zero or one based on a probability, i.e. the cumulative distribution function (CDF) of the standard normal distribution. Its mathematical expression is therefore the one in 2.14.

$$GELU(x) = xP(X \leq x) = x\Phi(x) = x \times \frac{1}{2}[1 + erf(x/\sqrt{2})] \tag{2.14}$$

Using different CDFs would create different activation functions, but authors in [6] states that GELU outperforms other choices and it does not add further hyperparameters. GELU behaviour compared to ReLU and ELU activations is plotted in 2.7.

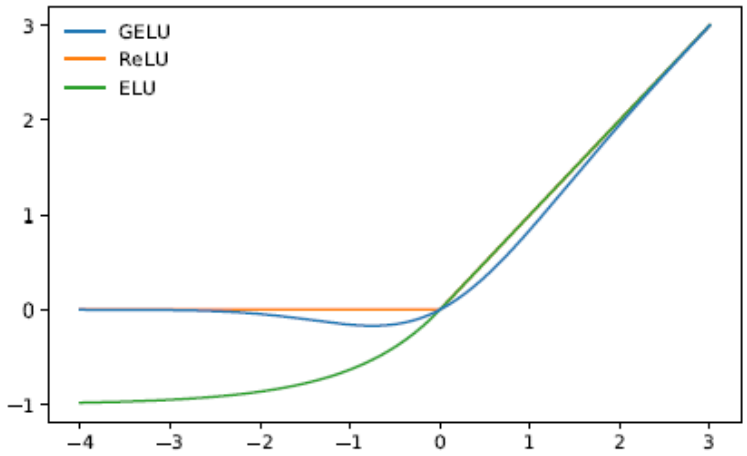

Figure 2.7: Behaviour of GELU, ReLU and ELU activation functions

From this plot, it can be seen that GELU is smoother than ReLU around zero and it is also more curve and non-monotone, giving a bigger non-linearity that is important to the network. Furthermore, GELU can be negative meaning that some inputs are not killed and can contribute to the network, contrasting one of the drawbacks of ReLU.

2.4 Parameters

In the structure description many parameters were used to define dimensions of various layers. Different type of Transformer networks are determined by these values.

Here's a quick recap of the parameters and what they represent.

1. $m \longrightarrow$ Sequence length, number of symbols processed in parallel.
2. $d \longrightarrow$ Embedding dimension, main dimension for the representation of the symbols throughout the network.
3. $d_{ff} \longrightarrow$ Hidden dimension, used in the Feed-Forward Network between the two linear transformations.
4. $k \longrightarrow$ Number of heads in the attention computation, number of inputs subdivision to exploit different intra-symbols relation.

Hidden dimension is usually a multiple of the embedding dimension; reasonable values are twice or four times the embedding dimension. Different combinations of the parameters realize different Transformer models, e.g. Base Transformer has 8 heads, embedding dimension equal to 512 and hidden dimension equal to 2048.

Table 2.1 reports some combinations, taken from a pair of Github repositories about NLP and Transformer/Universal Transformer models.

	TRANSFORMER			**UNIVERSAL TRANSFORMER**			**BERT**	
Model	**Base**	**Big**	**Tiny**	**Base**	**Big**	**Tiny**	**Base**	**Large**
m	256	256	256	256	256	256	256	256
d	512	1024	256	1024	2048	128	768	1024
d_{ff}	2048	4096	1024	4096	8192	128	3072	4096
k	8	16	4	16	16	2	12	16
d/k	64	64	64	64	128	64	64	64
Layers	6			*Variable*			12	24

Table 2.1: Parameters value in different Transformer, Universal Transformer and BERT dimensions.

Last line refer to number of encoder/decoder layers used in the models. Recall that Universal Transformer has one encoder and one decoder layer, iterated a adaptive number of times.

These are only some examples; there are many different options to realize this kind of network from a parameters perspective. In fact, other values can be chosen but usually only multiple of those are used and feasible.

Chapter 3

Layers State-of-the-art

Transformer layers require many complex operations and that can be very critical for an hardware implementation. Hardware resources are limited leading to a necessity to look for alternative ways to do some things to lighten the burden on the resources. In the literature there are different expressions for what the Transformer need in the computation. This is the aim of this chapter: dig into papers and articles to find alternative options to simplify operations.

Some of these ideas are also hardware-proven but majority of them are explained only at software level since, like already said, hardware applications lack.

3.1 Softmax

Computationally speaking, one of the critical layers in the network is the Softmax operation. Recalling formula 2.7 it has division, natural exponential and the need to scan the input to accumulate the denominator. Furthermore, it is required both in the encoder and the decoder as well as at the output to obtain final probabilities.

Firstly, some mathematical manipulations can be made to handle better the equation and the resources.

As in an exponential operation range of the values can get too big, one of the principles is to reduce the input range to obtain also smaller outputs. Subtracting the maximum value among the inputs to all inputs allows to achieve this idea. In equation 3.1 the concept is applied; drawback of this approach is the additional need to compute the maximum value before doing the actual processing, but anyway the iteration on the inputs was already required by the accumulation.

$$Softmax(D(i,j)) = \frac{\exp(D(i,j) - \underset{j}{max} D(i,j))}{\sum_{k=1}^{N} \exp(D(i,j) - \underset{j}{max} D(i,j))} \quad per\ j = 1, ..., N \tag{3.1}$$

where D is the input matrix, for each row i a Softmax on the columns j is applied. Since rows represents tokens, this means that for each token, column values are transformed into probabilities to understand connection between symbols.

Exponential properties can be also applied to this expression, leading to 3.2 where the division is replaced by a natural logarithm. This is described in [8].

$$\begin{aligned} Softmax(D(i,j)) = \exp(D(i,j) - \max_j D(i,j) + \\ - \log(\sum_{k=1}^{N} \exp(D(i,j) - \max_j D(i,j)))) \end{aligned} \tag{3.2}$$

Avoiding divisions can be very useful because they would require special hardware units and dividers are complex. However, this approach moves the problem from the divider to the logarithm, that is complex as well and probably the best way to implement it is a Look-Up Table (LUT).

[9] starts from the expression in 3.2 and approximates it neglecting the log term (that would be the denominator of the original formula) totally or partially. The result are in equation 3.3.

$$\begin{aligned} Softmax(D(i,j)) = \exp(D(i,j) - m \quad where\ m = \max_j D(i,j) \\ Softmax(D(i,j)) = \exp(D(i,j) - m - \sum_{k=1}^{p} \exp(m_k - m) + 1) \end{aligned} \tag{3.3}$$

The first line avoid any further computations beyond the numerator, while the second line is a partially abortion of the logarithm term. In fact the logarithm is linearized and only the p maximum values are taken into consideration.

First solution brings some problems as the approximation error can be very large; this is effective only if the inputs have the right distribution to allow it. The second one is a little more accurate but it also depends on the inputs, but the parameter p can help to tune it and have less approximation error.

This kind of approaches are much interesting for the hardware application but they can be unsustainable for the performances.

Paper [10] addresses the exponential and logarithmic units from a hardware perspective. The expression that authors use is in 3.4: range reduction is not applied.

$$Softmax(D(i,j)) = \exp(D(i,j) - \log(\sum_{k=1}^{N} \exp D(i,j))) \tag{3.4}$$

Figures 3.2 and 3.3 describe the hardware units required in the proposed architecture (figure 3.1).

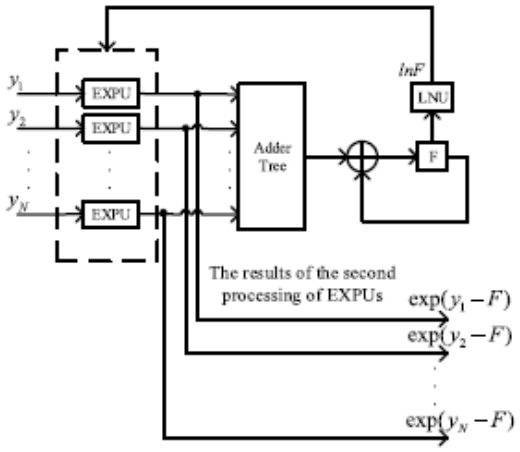

Figure 3.1: Example of Softmax architecture []

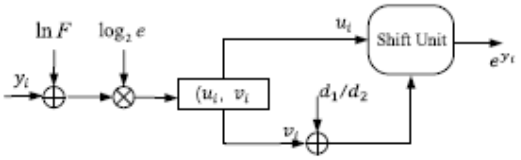

Figure 3.2: Exponential unit proposed in [10]

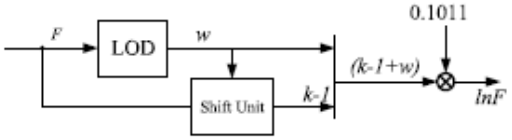

Figure 3.3: Logarithmic unit proposed in [10]

The interesting part of the exponential unit is that, to simplify the operations to be made, a base transformation is applied; in fact base 2 in hardware application is easier than base e. 3.5 describes the mathematical property that does this.

$$e^{y_i} = 2^{y_i \log_2 e} \tag{3.5}$$

So after the initial multiplication by $\log_2 e$ the exponent is divided into integer and decimal parts (respectively u_i and v_i); if the value is a fixed point value the separation is immediate. As u_i is an integer value, 2^{u_i} is simply a shift operation, therefore $2^{u_i+v_i} = 2^{u_i}2^{v_i} = 2^{v_i}\ shifted\ by\ u_i$, where shift can be leftward or rightward depending on the sign of the number. 2^{v_i} on the other hand is approximated to a sum by a bias value (d_1 on the first iteration and d_2 on the second), because it's in

the range between 0 and 1. Moreover, multiplication by $\log_2 e$, being a constant multiplication, can be optimized with numerical methods and looking it bit by bit.

Similarly, also the logarithmic unit exploits mathematical properties and some approximations, that are in 3.6.

$$\begin{aligned} \log F = \log 2 * \log_2 F = \log(2)(w + \log_2 k) \quad where\ k, w: \ F = 2^w + k \\ \log_2 k \simeq k - 1 \quad k \in [1,2) \\ \log F = \log(2)(k - 1 + w) \end{aligned} \tag{3.6}$$

LOD in figure 3.3 is a Leading One Detector to divide the input value into k and w, in fact the position of the highest one in F and the given decimal point allow to find w and then with a shift operation k. $log_2 k$ is computed with a linear fitting, considering the range of k between 1 and 2. Eventually, $k - 1 + w$ does not need adders since $k - 1$ is the fractional part of k and w is an integer value. Final multiplication by $log2$ is again a constant multiplication and can be implemented in a customized way.

Another option for exponential unit is to avoid dedicated hardware operators using Look-up Tables. This approach has the drawback of burdening the memory requirements; in fact these tables can be very large and they should be entirely stored in memory.

[11] and [12] propose two different LUT solutions, trying to lighten the memory requirements. The former uses a exponential property, namely $e^x = e^{x_1} \times e^{x_2} \times e^{x_3}$ with $x = x_1 + x_2 + x_3$, obtaining smaller sub-LUTs computing smaller exponentials and a final multiplier for the result. This way, tables occupy less memory space and in some cases not all of them are required for the final multiplication. The latter approximate the exponential function with a Piecewise Linear Function made by S continuous pieces. $f^s(x) = \alpha^s * x + (y_l^s - \alpha^s * x_l^s \quad x \in [x_l^s, x_r^s] \quad y_l^s = e^{x_l^s} \quad s \in [1, S]$. Only x_l^s, α^s and the parenthesis term have to be stored in a LUT, that along with an adder and a multiplier performs the whole computation.

Most interesting solution is Softermax [13]. Two main ideas are applied to this: base replacement and online normalization computation.

Base replacement has been already explained, power of 2 is simpler than power of e from a hardware point of view.

Online normalization computation addresses the need of scanning the input vector to find the maximum value. This would lead to three iterations on the vector, one to find the maximum value, one to compute the normalization value, namely the denominator, and the last to compute the final results. In this paper the normalization sum is continuously updated in the first iteration with the temporary

maximum value, and whenever a new maximum is found a correction factor is applied to compensate the error made previously. Figure 3.4 reports the passages to reach the final algorithm for Softermax.

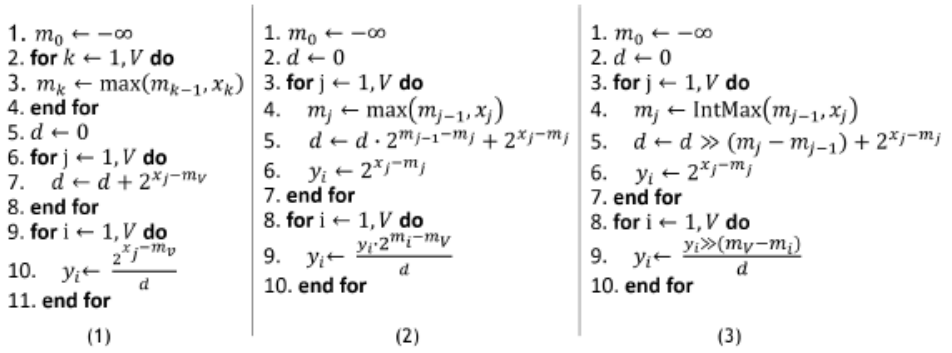

Fig. 3: The algorithmic changes proposed in `Softermax` consist of: (1) replacing e^x with a low-precision implementation of 2^x, (2) replacing an explicit pass to calculate the max with an online version, and (3) replacing the maximum function with an integer-based version to simplify the renormalization calculations.

Figure 3.4: Softermax algorithm

To simplify corrections and computation the maximum is transformed into an integer value, so the exponential by 2 becomes a simple shift operation. Speaking of hardware implementation, the figure 3.5 is taken from the paper[13] and reports the needed units to perform the Softmax.

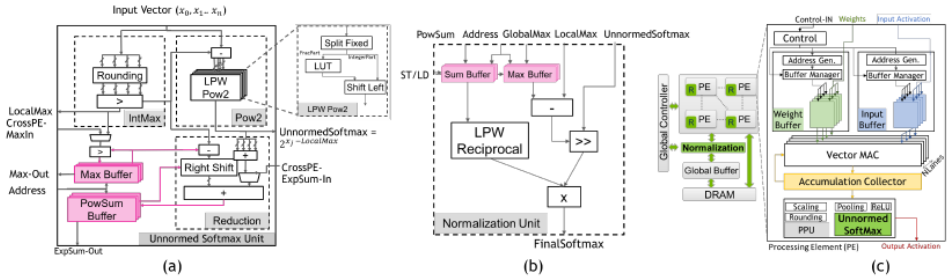

Fig. 4: (a) The Unnormed Softmax Unit determines the local max, performs the power of 2 calculation using the current max, and accumulates the denominator. (b) The Normalization Unit performs the renormalization of the numerator, as well as the final division of the numerator by the accumulated sum. (c) In an example accelerator [17], the Unnormed Softmax can be integrated into the post-processing vector unit on a per PE basis, while the Normalization Unit can be shared across multiple PEs and integrated between the PEs and the Global Buffer

Figure 3.5: Softermax architecture

These units are:

- IntMax: compute local maximum and convert it into integer value.
- Power of Two Unit: splits integer and fractional part of the input value and performs the exponential using LUTs for m and c of a piecewise linear function.
  $lpw = m_{LUT}[int(x_{scaled})] * frac(x_{scaled}) + c_{LUT}[int(x_{scaled})]$
  $x_{scaled} = frac(x << 2)$
- Reduction Unit: it reduces the result of power of two and updates the running sum, possibly renormalizing it if a new maximum is there.
- Normalization Unit: normalizes the final sum if necessary and implement the division through a LPW reciprocal unit.

This proposal has the advantage of simplifying the maximum value and the normalization value computation, even though an additional correcting mechanism is necessary. Then the conversion to integer make this modification feasible avoiding some complications. Critical parts can be the presence of LUTs in the Power of Two Unit and the LPW Reciprocal to perform the division.

3.2 Layer Normalization

The other sub-layer that contains computationally expensive operation is the Layer Normalization. In fact, recalling formula 2.10 it requires to calculate mean value and variance along the rows of the input matrix, plus a square root operation.

In this section, two solutions are described, taken from two papers.

First one is the *RMS Normalization*[14] that substitutes mean and variance with a single statistic over the inputs, namely the Root Mean Square. Resulting expression is the following:

$$RMSNorm(a) = \frac{a_i}{RMS(a)} * g_i + b_i \; RMS(a) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} a_i^2} \tag{3.7}$$

In other words, this is simply a Layer Normalization with a mean value equal to zero: this is a real gain in a hardware-oriented application as it would translate to less operations and resources.

Further optimization is the partial RMS (*pRMS*) Normalization, where the authors in [14] states that, depending on the statistics of the input, not all inputs are needed for the RMS computation.

$$pRMS(a) = \sqrt{\frac{1}{k}\sum_{i=1}^{k} a_i^2} \; k = \lceil n * p \rceil \tag{3.8}$$

p is the percentage of inputs (starting from the first) involved in the RMS; according to the paper, models can succeed in convergence with a partial ratio of 6.25%. This clearly would benefit a hardware application, making the whole computation faster. Drawbacks are on the loss of accuracy and the dependency on the input statistics.

The other solution worthy of mention is the one proposed in [15]. In this paper, authors discuss three normalization changes: *PreNorm*, *ScaleNorm* and *FixNorm*. *PreNorm* is about the position itself of the layer in the network: instead of normalizing the output of the residual connection they suggest to put the normalization before the sublayer. *Scale Norm* replaces LayerNorm with a "scaled l_2 normalization", whose equation is 3.9.

$$ScaleNorm(\mathbf{x}, g) = g\frac{\mathbf{x}}{||\mathbf{x}||} \tag{3.9}$$

g is learned but it is the same for the entire vector to normalize This is similar to the RMSNorm, in fact tying g_i of RMSNorm and dividing by $\sqrt{d}$ would bring the two methods to correspond. *FixNorm* eventually is applied only to the last linear layer and it has g fixed, leading to the expression $FixNorm(\mathbf{w}) = g\frac{\mathbf{w}}{||\mathbf{w}||}$.

3.3 Hardware references

Some hardware accelerators have already been developed and can be useful references for any work in this direction.

One of them is [8], where the Feed-Forward network and the Multi-Head Self Attention are taken into account for an hardware solution.

By looking at the scheme in 3.6 it's clear that it's built around a *Systolic Array* (SA) block of dimensions $s \times 64$ (s is the sequence length). It is shared between the two sublayers and it is in charge of every matrix multiplication needed. Larger matrices are partitioned to fit into this array: this increases the number of operations but keeps a single block.

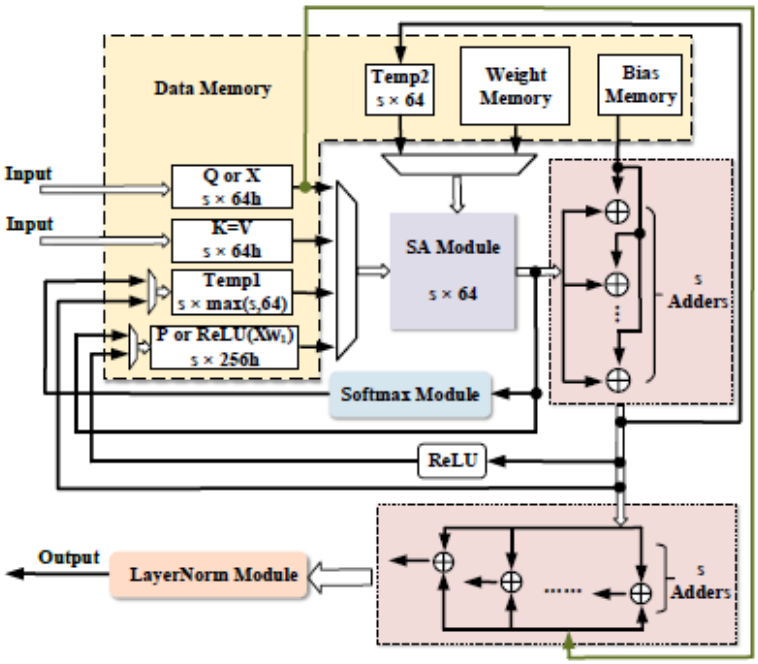

Figure 3.6: Architecture of the MHSA and FFN accelerator

SA array is a 2D array of Processing Elements with s rows and 64 columns. It is designed to output the result one column at a time so that the bias addition is made by s adders, one for each column element. Moreover, it is by far the most complex part in the architecture and, supposing that the Softmax component can give the output while the SA is computing the Value matrix, it determine also the latency of the architecture.

Softmax's arithmetic manipulations have been already cited in equation 3.4, resulting in architecture 3.7. Computation of this module is divided in four phases, all described in the figure. Exponential and logarithmic units remain quite complex, and authors have used solutions in [16] to realize them.

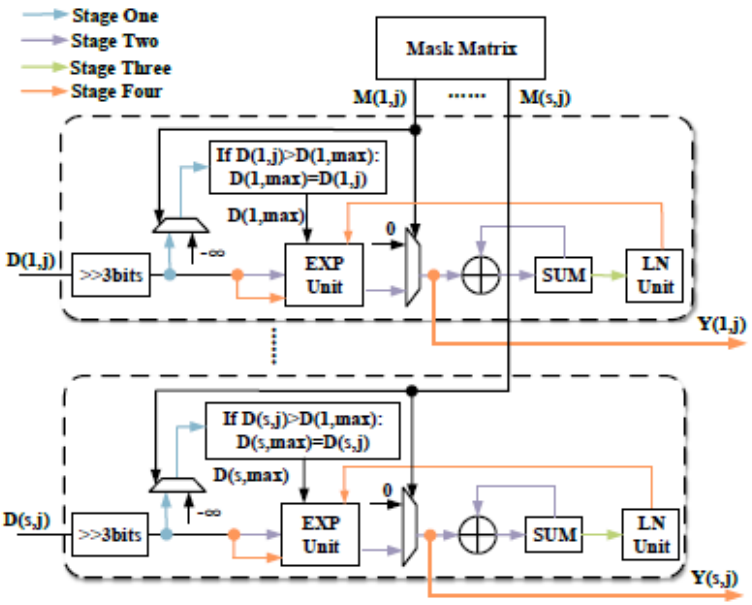

Figure 3.7: Softmax architecture

For the Layer Normalization, figure 3.8, a way to save some cycles in the computation is exploited. In fact, using a different expression for the variance (3.10) and some extra resources, all contribution for the variance can be calculated along with the mean value.

$$var(G,i) = E(G,i)^2 - \frac{1}{d_{model}} \sum_{k=1}^{d_{model}} G(i,k)^2 \tag{3.10}$$

If the resources can be afforded, this is a good solution to accelerate the LayerNorm and reduce the delay of the layer. However, this paper performs internal calculations with FP32 representation; the following sections will describe a different approach to have the whole network quantized.

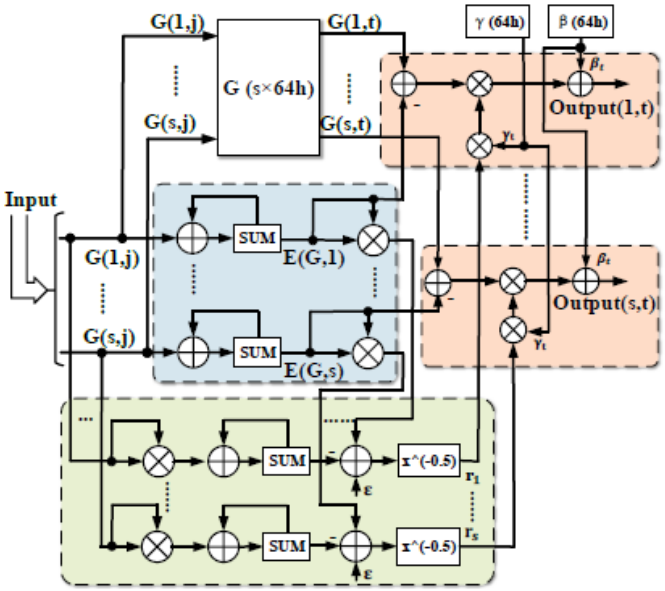

Figure 3.8: LayerNorm architecture

Chapter 4

Quantized Model

Transformer models have state-of-the-art performance in NLP tasks, but their memory footprint, latency and power consumption make their hardware application harder, especially for real-time inference. To simplify the implementation and lighten the network's burden there are many techniques and approaches. One of them is quantization, that try to reduce the data footprint by using low precision to store and handling parameters and activations.

4.1 I-BERT

This section is based on the reference paper "I-BERT: Integer-only BERT quantization"[1].

This paper describes the full quantization of the BERT network, in order to have a integer-only model to be easily implemented in hardware. To achieve their objective, authors review each layer composing the network and quantize them using different algorithms.
Differently from many other papers where there is a lot of quantization but the final solution requires anyway some floating-point arithmetic, I-BERT allow to use only integer operands. Avoiding data conversion from integer to floating-point between layers is very important for an hardware implementation, as integer arithmetic unit are way simpler and smaller. Giving that the Transformer structure is big and requires a lot of operations this could be a huge gain in terms of area and latency.

Most critical layers for the quantization are the non-linear ones, since the matrix multiplication part is quantized in several other proposals. In I-BERT *Softmax* and *GELU* are approximated through the use of second-order polynomials, while for *Layer Normalization* the focus is on the square root operation, performed using an already-known iterative algorithm. The downside of this operation is the loss of accuracy due to the lower precision in data representation; this obviously is not

much relevant in the matrix multiplication operation, but in the non-linear layers the loss is bigger and more critical. The precision choice made by the authors is to perform the integer matrix multiplication on 8 bits and accumulate on 32 bits, that is also the parallelism of the non-linear layers. So there is the need of only one data transformation from 32-bit integers to 8-bit integers.

One important thing to remember in the quantization method is the *scaling factor*: this is the factor that allow the transformation from floating point representation to the integer one and vice versa. Given that a is the floating point value, and q_a is the quantized one, the scaling factor is defined as $S_a = a/q_a$ meaning that $a = q_a S_a$. Scaling factor follows the steps in parallel to the quantized values. For example it is not possible to add two numbers with different scaling factor: the result would be useless, as it would have no mathematical sense.

In the following paragraphs there are some algorithms that involve several coefficients whose expressions contain the scaling factor. For the inference, this parameter is set, so these coefficients can be precomputed and used as constants.

4.1.1 Matrix multiplication

On the opposite, the multiplication between two numbers with different scaling factor is allowed, since $a * b = q_a S_a * q_b S_b = (q_a * q_b)(S_a * S_b)$. So in a multiplication step, the two quantized values can be simply multiplied with the output scaling factor corresponding to the product of the two input scaling factors. This is legal for all linear operations: *MatMul* is one of them as explained above. Therefore the resulting expression for the matrix multiplication is $MatMul(Sq) = S * MatMul(q)$.

This property does not apply to the non-linear functions (e.g. $GELU(Sq) \neq S * GELU(q)$). Now, here's an overview of the proposed algorithms to better understand how the non-linear operation are effectively approximated.
Refer to [1] for any mathematical explanation.

4.1.2 Second-order Polynomial

To convert each non-linear layer into an integer version, the idea is to use a second-order polynomial that simulates the behaviour of non-linear functions as good as possible. Obviously higher order polynomials have lower accuracy errors but they are more complex for the eventual implementation, so the authors picked 2 as the polynomial order. In algorithm 1 it is described the simple steps to compute the polynomial. It only takes two addition and a multiplication, so it can be easily used for an hardware implementation.

Algorithm 1 Integer-only Computation of Second-order Polynomial $a(x+b)^2+c$

Input q, S: quantized input and scaling factor
Output q_{out}, S_{out}: quantized input and scaling factor

function I-Poly(q, S) ▷ $qS = x$
    $q_b \leftarrow \lfloor b/S \rfloor$
    $q_c \leftarrow \lfloor c/aS^2 \rfloor$
    $S_{out} \leftarrow \lfloor aS^2 \rfloor$
    $q_{out} \leftarrow (q+q_b)^2 + q_c$
    return q_{out}, S_{out} ▷ $q_{out}, S_{out} \approx a(x+b)^2+c$)

4.1.3 Softmax

Using the *Softmax* expression in 3.1, the exponential operation is the one to approximate in order to have an integer-based solution. It is important to reduce input range by subtracting the input maximum value before the exponential, so the values are not going to "explode".

By decomposing the input and reducing its range, the exponential can be substituted by a second-order polynomial and a shift operation, as described in algorithm 2.

Algorithm 2 Integer-only Exponential and Softmax

Input q, S: quantized input and scaling factor
Output q_{out}, S_{out}: quantized input and scaling factor

function I-Exp(q, S) ▷ $qS = x$
  $a, b, c \leftarrow 0.3585, 1.353, 0.344$
  $q_{ln2} \leftarrow \lfloor \frac{ln2}{S} \rfloor$
  $z \leftarrow \lfloor -\frac{q}{q_{ln2}} \rfloor$
  $q_p \leftarrow q + zq_{ln2}$ ▷ $q_pS = p$
  $q_L, S_L \leftarrow$ I-Poly(q_p, S) with a, b, c
  $q_{out}, S_{out} \leftarrow q_L >> z, S_L$
  return q_{out}, S_{out} ▷ $q_{out}, S_{out} \approx exp(x)$

function I-Softmax(q, S) ▷ $qS = x$
  $\tilde{q} \leftarrow q - \max(q)$
  $q_{exp}, S_{exp} \leftarrow$I-Exp$(\tilde{q}, S)$
  $q_{out}, S_{out} \leftarrow \frac{q_{exp}}{sum(q_{exp})}, S_{exp}$
  return q_{out}, S_{out} ▷ $q_{out}, S_{out} \approx Softmax(x)$

Once the exponential is approximated with an integer-only computation, the search of the maximum value, an accumulation and a division are required to complete the *Softmax* algorithm. Unless an online computation of the maximum is performed, the whole processing requires three phases: search of the maximum value, accumulation of the $sum(q_{exp})$ and the final division.

4.1.4 GELU

Another important non-linear function used in BERT is the GELU activation function, already described in 2.3. Authors refer to using the *Sigmoid* to approximate the *erf* function but it would drop accuracy too much; so they go to the polynomial optimization problem to find a good solution. In algorithm 3 there is

the pseudo-code of this solution. *erf* function is implemented with a second-order polynomial (1) with some sign handling and data clipping, while $GELU$ needs only one extra addition and one extra multiplication.

Algorithm 3 Integer-only GELU

Input q, S: quantized input and scaling factor
Output q_{out}, S_{out}: quantized input and scaling factor

function I-ERF(q, S) ▷ $qS = x$
  $a, b, c \leftarrow -0.2888, -1.769, 1$
  $q_{sgn}, q \leftarrow sgn(q), clip(|q|, max = -b/S)$
  $q_L, S_L \leftarrow$ I-Poly(q_S) with a, b, c
  $q_{out}, S_{out} \leftarrow q_{sgn} q_L, S_L$
  return q_{out}, S_{out} ▷ $q_{out}, S_{out} \approx erf(x)$

function I-GELU(q, S) ▷ $qS = x$
  $q_{erf}, S_{erf} \leftarrow$ I-ERF$(q, S/\sqrt{2})$
  $q_1 \leftarrow \lfloor 1/S_{Erf} \rfloor$
  $q_{out}, S_{out} \leftarrow q(q_{erf} + q_1), SS_{erf}/2$
  return q_{out}, S_{out} ▷ $q_{out}, S_{out} \approx GELU(x)$

4.1.5 Layer Normalization

Last big non linear layer is the Layer Normalization (2.10), whose only element of non-linearity is the square root operation. This is indeed the focus of the Integer-only Layer Normalization in the paper. The proposed solution is an iterative algorithm, described in 4, taken from [17].

It is based on the Newton's method and it searches for the exact value of $\lfloor \sqrt{n} \rfloor$. As a consequence, to have the final value multiple cycles are required, the drawback is that the number of these cycles is not previously known, so the execution of the Normalization has to wait to have the correct value.
As in Softmax, the computation is divided in different phases: firstly the average has to be calculated, then the standard deviation and eventually the layer output can be computed.

4.1.6 Requantization

This layer is an additional operation not present in the original floating-point architecture; in fact this is the process to represent the data back to 8 bits from

Algorithm 4 Integer-only Square Root

Input n: input integer
Output integer square root of n, i.e. $\lfloor \sqrt{n} \rfloor$

function I-SQRT(n)
    if $n = 0$ **then**
        return 0
    Initialize x_0 to $2^{\lceil Bits(n)/2 \rceil}$ and i to 0
    repeat
        $x_{i+1} \leftarrow \lfloor (x_i + \lfloor n/x_i \rfloor)/2 \rfloor$
        if $x_{i+1} \geq x_i$ **then**
            return x_i
        else
            $i \leftarrow i + 1$

the 32 bits used in the accumulation and non-linear functions. Therefore this is needed before every matrix multiplication coming from a 32-bit processing.

To perform this transformation the *Dyadic Numbers* concept is involved, described in article [18]. Starting from the 32-bit representation, denoted for example with q_a so that $a = q_a S_a$, the final representation should be $o = q_o S_o$ with q_o on 8 bit. So, equalling a and o since the real value must not change, the expression is:

$$q_a S_a = q_o S_o \longrightarrow q_o = q_a \frac{S_a}{S_o} \tag{4.1}$$

Remembering that the scaling factors are not integers, this expression can not be implemented directly on integer-only resources. That's why the scaling factor ratio is represented with a dyadic number, that by definition are rational numbers with the format of $b/2^c$, where b and c are two integer numbers. The final equation is 4.2.

$$q_o = q_a \frac{S_a}{S_o} = q_a DN(\frac{S_a}{S_o}) = q_a * \frac{b}{2^c} \tag{4.2}$$

Required resources are now a INT32 multiplication and a bit shifting. This has also the advantage of avoiding a division, that is substituted by a simple shifting.

Chapter 5

Hardware Design

In this chapter the proposed architecture of the network is described, layer by layer. This is the result of the state-of-the-art research and the reference paper about *I-BERT*. Reminding the base model of the *BERT*, in figure 2.6, a divide-and-conquer approach can be used to simplify the problems and the understanding of the architecture. Firstly, the two biggest layers are the *Multi-Head Self Attention* and the *Feed-Forward Network*, that combined with the *Residual Connection* and the *Layer Normalization* form the entire structure. Then, inside each macro-layer, the focus is progressively tuned on smaller parts in order to have less operations to implement concurrently.

The idea which the whole architecture is based on comes from the paper [8] and that is processing matrices one column at a time. Having single columns going from a layer to the next helps in the multiplication (*MatMul*) blocks, that is described below.

5.1 Matrix Multiplication

Here the basic component is a simple *Multiply and Accumulate*, depicted in figure 5.1, that corresponds to a single element in the matrix product.

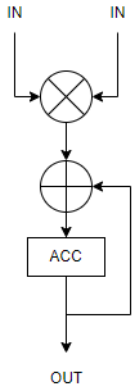

Figure 5.1: MAC architecture

Forming a matrix of this *MAC* blocks, as in figure 5.2 the matrix multiplication is performed feeding it with columns of matrix A and rows of matrix B one at a time.

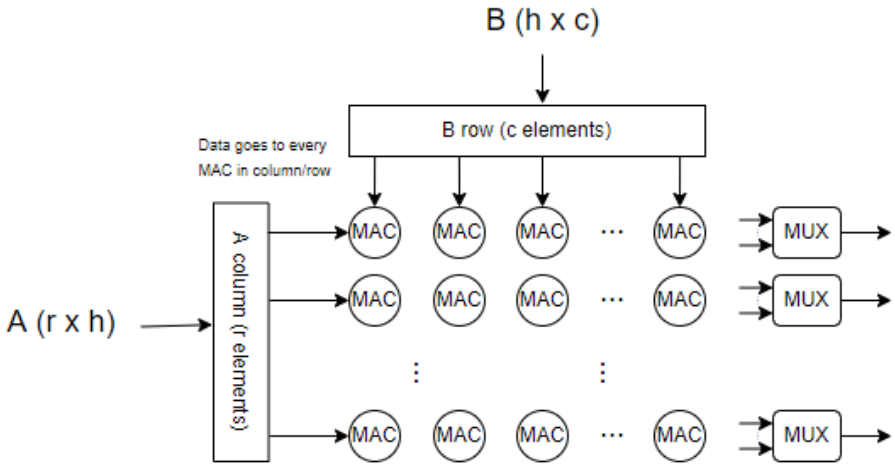

Figure 5.2: MatMul architecture

Doing so, at every cycle the *MAC*s accumulate the partial results and when there are no more inputs they will have stored the result. Since *MAC* has the accumulate register, outputs are kept in storage until the successive execution, so the matrix can be scanned one column at a time to give correct values to the next blocks. Multiplexers are employed to select the right column to output, so there is one multiplexer per row taking all values coming from its row's elements in input and choosing the selected one.

Additional signals needed are Clock, Reset and Enable, to activate the registers in the MAC when performing a multiplication with correct inputs.

Considering the dimensions of the matrix multiplications needed in a Transformer, these blocks are big contributions to the network dimensions and performance.

5.2 Multi-Head Self Attention

MHSA is composed by four MatMuls and the Attention operation, as it can be seen in figure 5.3.

One important thing to underline is that the V matrix has to be read one row per cycle when needed, because it will go to the B input of MatMul (referring to 5.2) after the Softmax operation. For the K matrix this does not apply because the multiplication in the Attention is $Q * K^T$, so K goes to the B input but with its columns, that are K^T rows.

Bias addition is performed when the matrices are computed and exit the MatMul component. Red dotted line represents the positions where *Requantization* has to be performed to bring the values to INT8 representation.

Figure 5.3 however depict only the computation of one single head; to have multiple heads calculated in parallel this datapath has to be replicated, with consequences on the area and complexity. This makes a lot of room to operate in the design of the architecture as different solutions can be adapted, affecting on the management of other layers too. In scheme 5.4 there is an example of MHSA architecture with 4 heads. In the block "head" the MatMul for Q, K and V and the following Attention are grouped together.

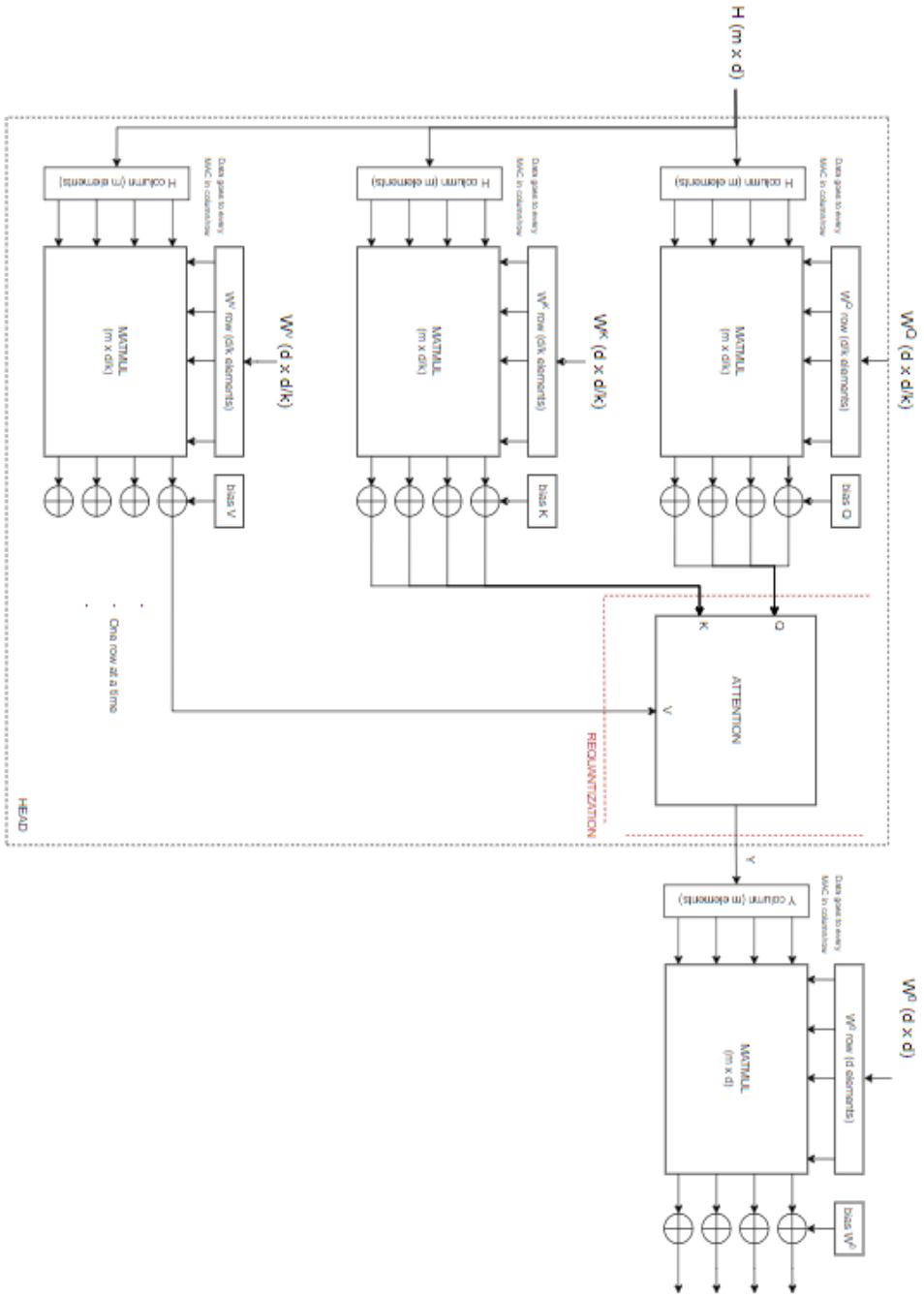

Figure 5.3: MHSA architecture

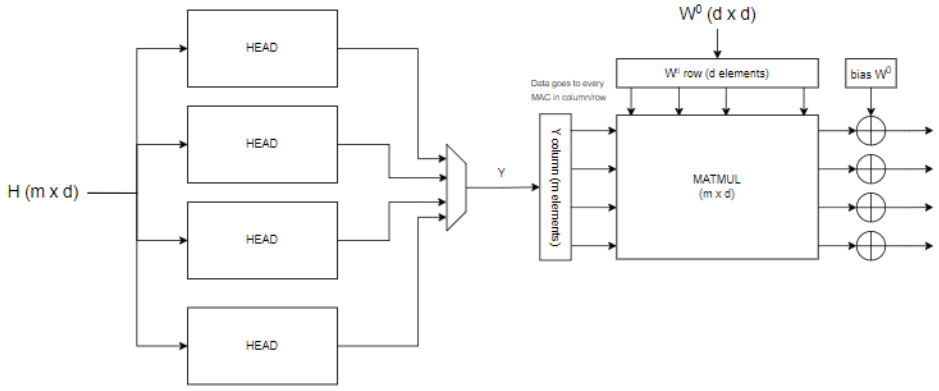

Figure 5.4: MHSA architecture with multiple heads

5.2.1 Attention

As already hinted above, *Attention* involves the multiplication of Q and K that produces the *attention scores*, which are, after going through the Softmax operation, multiplied by V eventually. This flow is represented in figure 5.5.

This MatMuls are not followed by a bias addition because they are not linear transformation but they involve only the three head-related matrices. Also here a Requantization layer is required, values exiting the Softmax are on 32 bit but in the following MatMul block the parallelism must be 8 bit. *Scale* stands for a division by a constant. Recalling 2.5, this constant in a Transformer is usually is equal to the square root of the embedding dimension ($\sqrt{d}$) and it is a power of two, allowing to perform only a shifting operation. In I-BERT however, d is 768 so its square root is around 27.

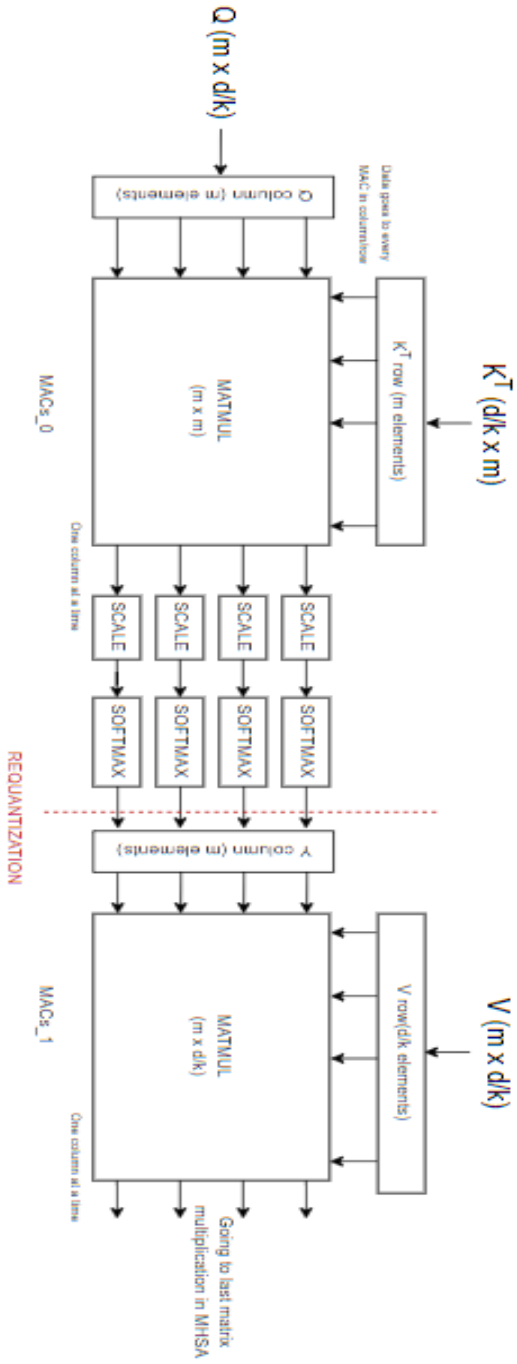

Figure 5.5: Attention architecture

Softmax

Softmax architecture is entirely based on the algorithm 2 and it is divided in the main operations to execute to perform it. Three phases are required:

- Search of maximum value
- Accumulation of exp values
- Computation of the output

In a Softmax block (5.6) inputs are coming one at a time and they refer to a single row of the matrix to be processed. So the matrix has to be scanned firstly to find the maximum values by means of a comparator. Once stored that value, the $exp(x - x_{max})$ elements can be calculated and accumulated into the denominator. At last the outputs can be computed with the final division. Input matrix has to be scanned and given in input to the block three times in total, "stalling" the execution of the network.

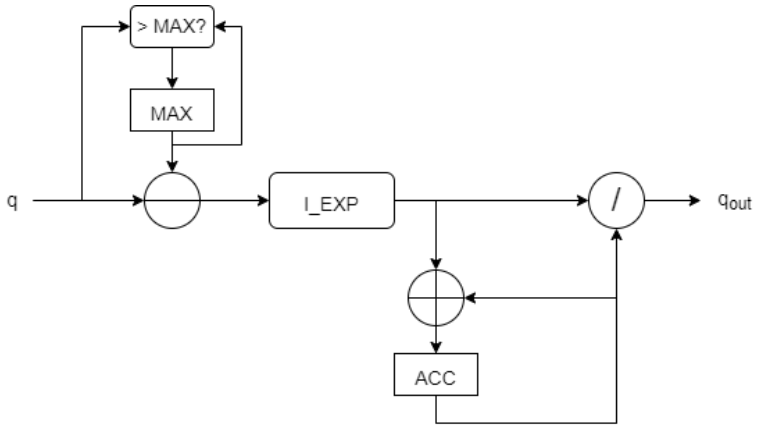

Figure 5.6: Softmax unit architecture

Exponential Exponential unit (figure 5.7) is composed by some multiplications, a polynomial unit and a shift.

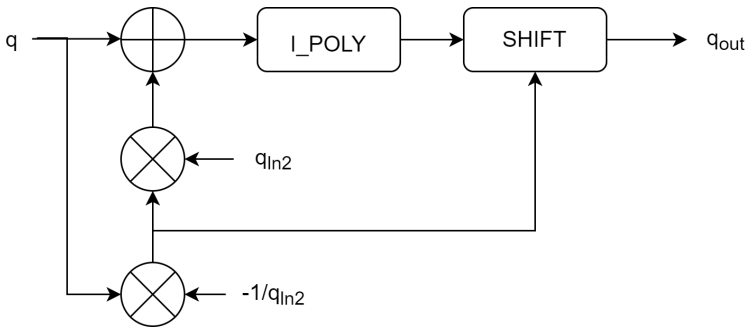

Figure 5.7: Exponential unit architecture

The polynomial component is directly implemented with two sums and a multiplication. For the exponential the polynomial version is the one in figure 5.8.

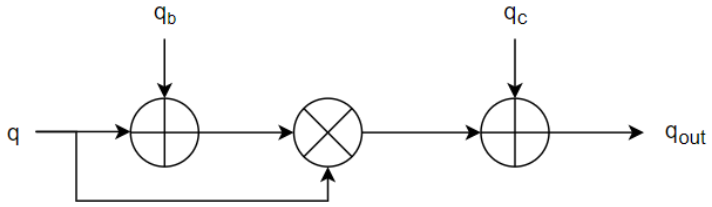

Figure 5.8: Polynomial unit architecture (exponential version)

In fact, by looking at the *Pytorch* code of the I-BERT, in this part the equation of the polynomial is $x(x+b)+c$ instead of the original $(x+b)^2+c$. To align the hardware and the reference software model, this change is taken into account and two different version of the polynomial component are made.

5.3 Feed-Forward Network

FFN is formed by two linear transformations and an activation, i.e. the GELU activation. Like in the MHSA layer, a Requantization operation is needed after the GELU activation before going into the following transformation.

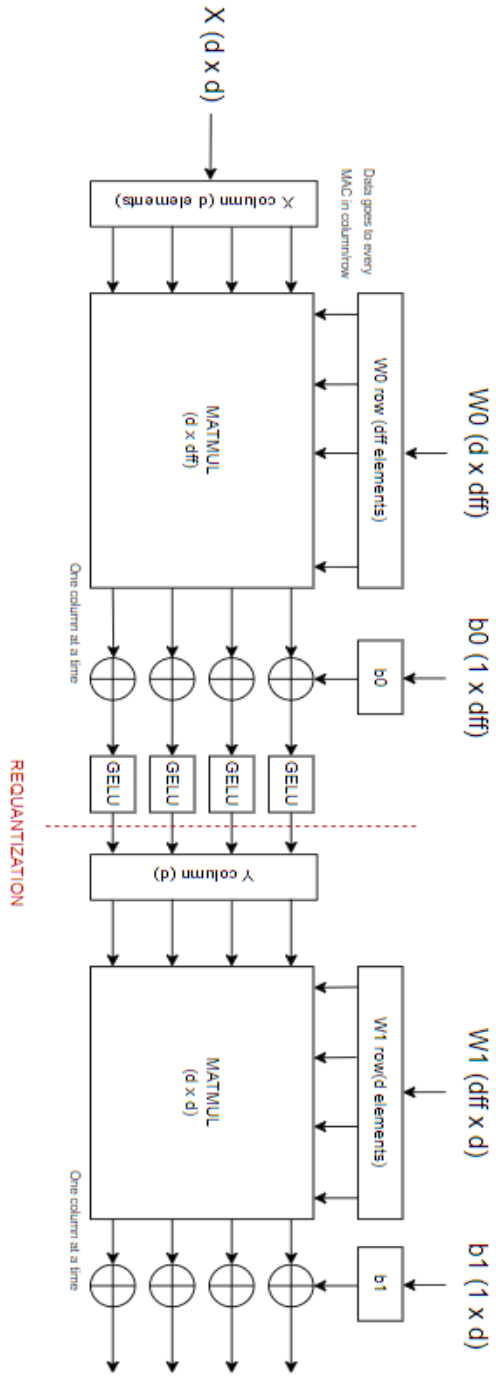

Figure 5.9: Feed Forward Network architecture

Usually d_{ff} is a multiple of d, therefore some manipulations on the structure can be made, e.g. folding the second transformations into the first. Any change has its effect on the memory requirements and on the matrix handling.

5.3.1 GELU

As it can be seen in figure 5.10, GELU architecture is simple, with only an addition and a multiplication after the *Erf* function unit.

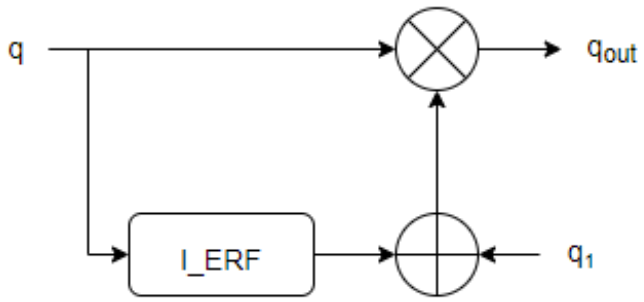

Figure 5.10: GELU unit architecture

Erf Erf function, recalling 3, is built around a polynomial unit. Input sign has to be separated by the value and used at the output; so the polynomial input is the absolute value of the Erf input upper-limited by a scaling-factor-depending constant.

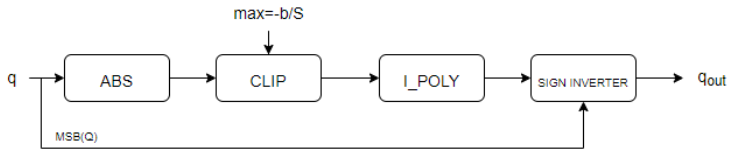

Figure 5.11: Erf unit architecture

Polynomial unit follows the original algorithm (1), differently from the above-explained exponential case.

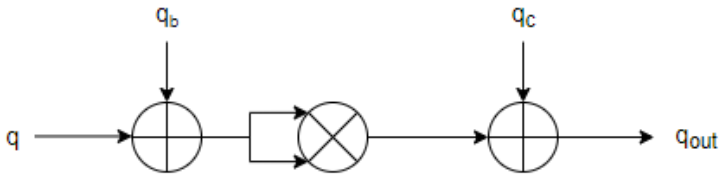

Figure 5.12: Polynomial unit architecture

5.4 Residual Connection

This could have been a simple addition (2.4) between columns of the two input matrices if the values were not quantized and related to their scaling factor. As already explained, two numbers with two different scaling factors cannot be added directly, so there is the need of this block (5.13) to make the residual connection.

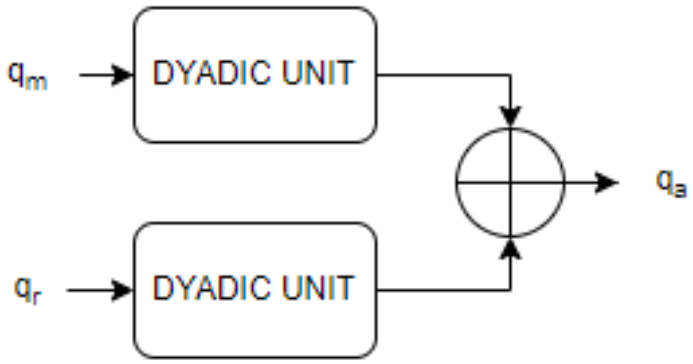

Figure 5.13: Residual Connection architecture

Having two inputs $r = q_r S_r$ and $m = q_m S_m$ and defining the output as $a = q_a S_a$ the sum is:

$$a = q_a S_a = r + m = q_r S_r + q_m S_m \longrightarrow q_a = DN(\frac{S_m}{S_a})q_m + DN(\frac{S_r}{S_a})q_r \qquad (5.1)$$

The concept behind this is the same of the *Requantization*, so it involves the Dyadic numbers to bring the two addends to the same scaling factor to sum them

together. In fact *Dyadic unit* is identical to the one used in the Requantization (5.16), taking two coefficients (b and c) and performing a multiplication and a shift.

5.5 Layer Normalization

From its equation 2.10 it is easy to understand that this layer has to do some processing on the inputs, and in this architecture setup this means that, like the Softmax, it needs three phases:

- Average computation
- Standard deviation computation
- Output computation

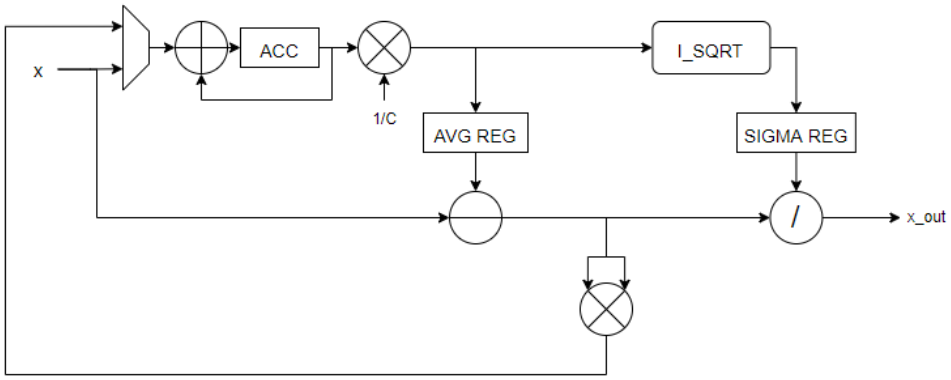

Figure 5.14: Layer Normalization unit architecture

Looking at the architecture scheme in 5.14, average and standard deviation have two dedicated registers, while the accumulator and divisor to compute the average value is shared between the two statistics involved. In fact after the input average is saved into its register, this resources are reused for calculating the variance (defined as the average of the $(x - avg)^2$ elements); a multiplexer is placed at the input of this operators to select which value to process depending on which phase is on. C value in the structure is the number of inputs to perform the average on: in the I-BERT case this is equal to the embedding dimension ($d = 768$) as the matrix that the Normalization layer receive as input have dimension $m \times d$ and it is performed along matrix's rows. In other Transformer networks this parameter is a power of two so the division can be substituted by a right shifting of $\log_2 d$ positions. To have the standard deviation a *Square Root* unit is needed and it is described below.

5.5.1 Square Root

Since it is based on the iterative algorithm described in 4, this computation requires more than one cycle to be completed. It has a constant initial value, defined as x_0, and after the first cycle the input is the previous output: this is realized using a multiplexer selecting the right input value. At every iteration, partial result x_i is compared to x_{i+1} that it is equal to $(x_i + x_i/n)/2$ (division by 2 is simply a one-position right shifting) until x_{i+1} is larger or equal to x_i. When this happens, x_i is the final result saved into the dedicated register and a *VALID* signal is asserted.

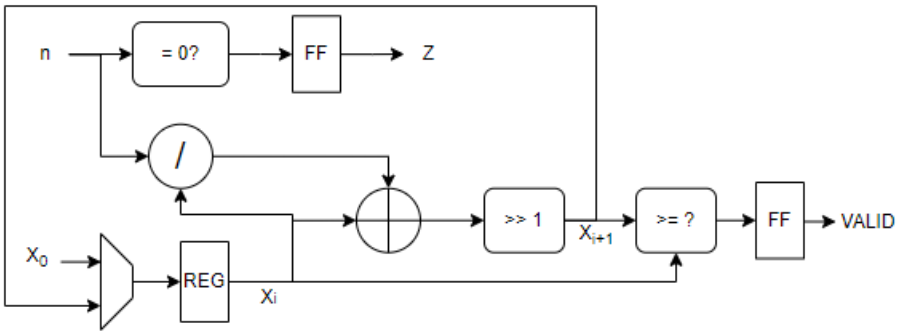

Figure 5.15: Square root unit architecture

A special case is when input n is directly 0, its square root is obviously 0 and it can be output instantly along with the Z signal.

Unknown timing of this last block make the Layer Normalization flow more complex and variable. Once the square root of the variance is available and the standard deviation is saved into the register, the last phase is performed by means of one extra divider to divide the $(x - avg)^2$ elements by σ.

5.6 Requantization

Starting from the equation of the Requantization (4.2) it is pretty straightforward to see that a multiplication and a shift are needed.

So the architecture, scheme 5.16, is formed by this two operators, taking as inputs the two coefficients of the dyadic number (m and c).

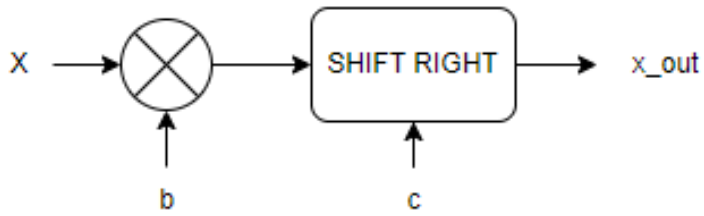

Figure 5.16: Requantization architecture

5.7 Synthesis results

5.7.1 MatMul

To estimate the impact a MatMul can have on the hardware area some synthesis were run. Using `Design Compiler`, some structures were synthesized to extract a trend both in terms of area and power in order to estimate bigger structure whose synthesis was not possible due to limitations.

Matrix dimensions were progressively increased until the synthesis was possible, by doubling at each step either the number of rows or columns, since the different Transformers structure differ by these kind of ratio. Plus, different parallelism were analysed: 8 bits, 16 bits and 32 bits.

Before looking at the synthesis results, here are some initial clarifications on the adopted method.

In figure 5.17 there's an example of a 4x4 MatMul matrix: it is important to point out that inputs are on n bits while accumulators and output are on $2n$ bits for this analysis. In the following results the number of bits will refer to n, so the input parallelism. Row/column inputs go to every MAC in the corresponding row/column; there are input and output registers to help the synthesizer and to have a more structured matrix. Mux selection signal is therefore delayed to match the timing, recalling that also MAC elements have a delaying component in the accumulation register.

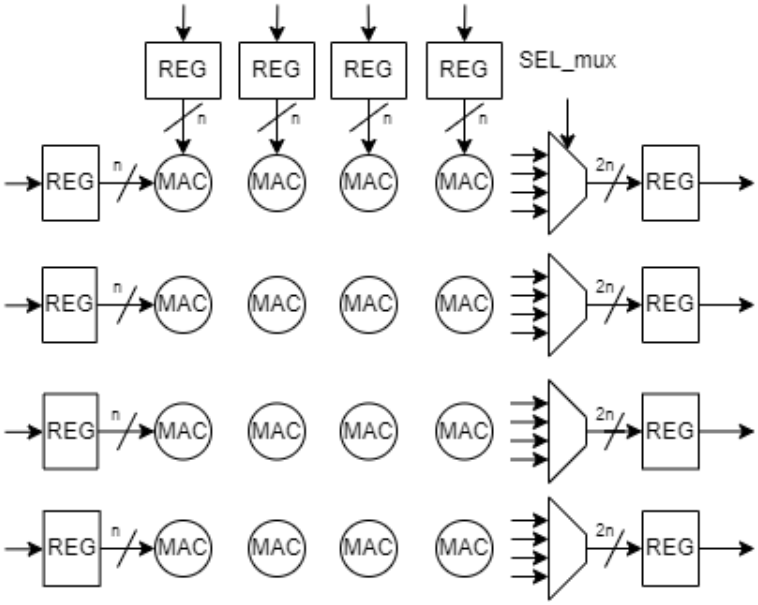

Figure 5.17: Example of 4x4 MAC matrix

Furthermore, to compile bigger structures, a bottom-up approach has been used, with some simpler matrices composing a bigger one, like in figure 5.18. With Design Compiler in fact, it is allowed to compile first the smaller blocks and then the top one starting from the already-synthesized solutions. This overcame a little the memory limitations of the software increasing the range of synthesizable matrices.

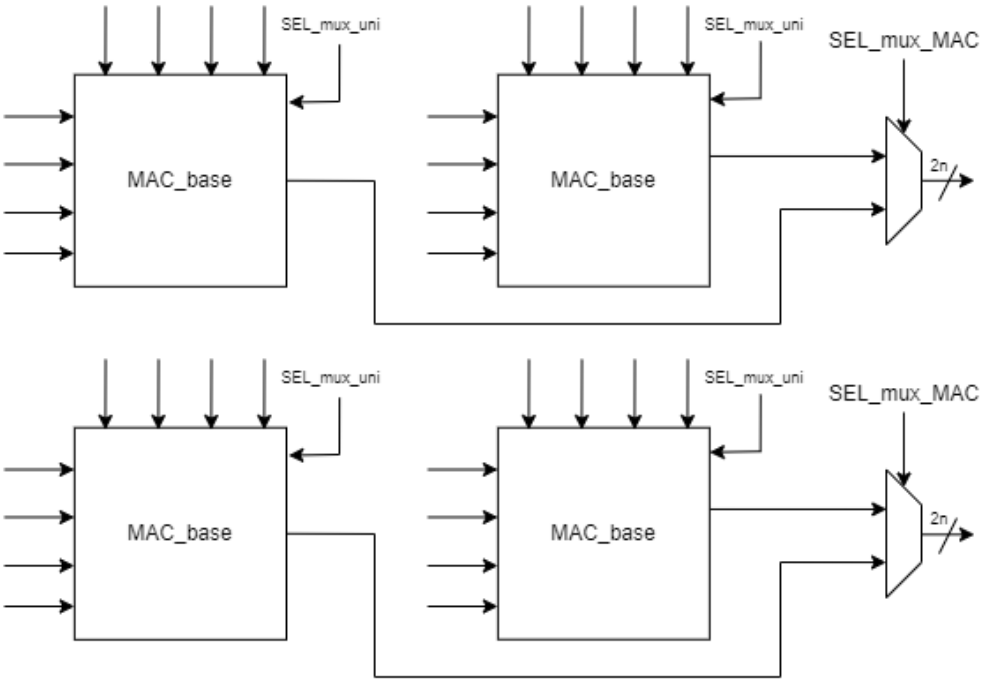

Figure 5.18: Example of a MAC composed matrix

So:

- Synthesis were performed by means of Design Compiler, using UMC65 technology in typical conditions.
- Very large structures could not be directly synthesized due to memory limitation of the software.
- Composed matrices were synthesized using a bottom-up approach, compiling first base components and multiplexers and eventually the top entity maintaining previously obtained designs.
- Following area and power results were obtained through *report_area* and *report_power* commands on the top level design.
- Clock in these synthesis was fixed at 7ns, to avoid timing issues. Value was chosen based on zero-period clock synthesis performed on small matrices; no negative slack was encountered.

Area analysis Table 5.1 reports extracted values for base MAC matrices.

There are two contributions on the area results: combinational and non-combinational. *Comb* area is given by the combinational components, i.e. the

non-clocked ones; *No Comb* area on the other end is formed by all other components, that are mainly registers and sequential elements.

MATRIX		8 BITS			16 BITS			32 BITS		
Row	Col	Comb	No Comb	Tot	Comb	No Comb	Tot	Comb	No Comb	Tot
32	8	146088	51917	198006	515631	103618	619249	1721686	207242	1928928
32	16	287004	96943	383947	1030316	193415	1223731	3470702	387184	3857886
32	32	582080	186674	768755	2042693	373485	2416178	6965795	746802	7712597
64	16	583368	192330	775697	2044095	384443	2428538	6943486	769125	7712611
64	32	1164182	370513	1534695	4107669	741208	4848878	13922859	1482508	1540567
64	64	2531498	727425	3075923	8208363	1454797	9663160	27859092	2907490	30766582

Table 5.1: Area values for base MAC matrix

From these values there are two ratios that can be observed:

- Relationships between different parallelism on same structure. (Table 5.2)
- Relationships between different structures with same parallelism. (Table 5.3)

It can be seen that ratios when doubling the parallelism are very similar, hinting that it could be a constant relationship useful for estimation.

MATRIX		RATIO 16 BITS/8 BITS			RATIO 32 BITS/16 BITS		
Row	Col	Comb	No Comb	Tot	Comb	No Comb	Tot
32	8	3.5296	1.9958	3.127	3.3389	2.000	3.1149
32	16	3.5899	1.9951	3.1872	3.3686	2.002	3.1526
32	32	3.5093	2.0008	3.143	3.4101	1.999	3.1920
64	16	3.5039	1.9988	3.13	3.397	2.001	3.1758
64	32	3.5284	2.0005	3.1595	3.3895	2.000	3.1771
64	64	3.4906	1.9999	3.1385	3.3939	1.9985	3.1839
AVERAGE		3.52	1.99	3.14	3.38	2.00	3.17

Table 5.2: Area ratios between same MAC base matrices with different parallelism

It can be seen that ratios when doubling the parallelism are very similar, hinting that it could be a constant relationship useful for estimation. But a better way to estimate non-synthesizable MatMul can be derived from the other table, 5.3. In fact if parallelism is kept constant, the area variation is almost equal to the structure variation. So, if the number of columns of a MatMul is doubled with respect to another MatMul, its area is two times the starting MatMul area. Note that *Tot Area* is simply the sum between the *Comb* and the *No comb* values, so it is not reported anymore.

MATRIX		8 BITS				16 BITS				32 BITS			
Row	Col	Comb	Ratio	No Comb	Ratio	Comb	Ratio	No Comb	Ratio	Comb	Ratio	No Comb	Ratio
32	8	146088		51917		515631		103618		1721686		207242	
32	16	287004	1.964	96943	1.867	1030316	1.998	193415	1.866	3470702	2.016	387184	1.868
32	32	582080	2.028	186674	1.925	2042693	1.982	373485	1.931	6965795	2.007	746802	1.928
64	16	583368	1.002	192330	1.030	2044095	1.001	384443	1.029	6943486	0.997	769125	1.029
64	32	1164182	1.995	370513	1.926	4107669	2.009	741208	1.928	13922859	2.005	1482508	1.927
64	64	2351498	2.019	727425	1.963	8208363	1.998	1454797	1.963	27859092	2.001	2907490	1.961

Table 5.3: Area ratios between different MAC base structure with same parallelism

Composed matrices were then analysed to see if they brought some differences to these relationship. Table 5.4 reports results for a 256x64 matrix, where it can be seen that the ratio between a composed matrix's area and its base block's area is equal to the number of base blocks required to have the top structure. 32 bits case was not possible due to memory errors of Design Compiler.

MATRIX 256x64			8 BITS				16 BITS			
Row Base	Col Base	# Base Blocks	Comb	Ratio	No Comb	Ratio	Comb	Ratio	No Comb	Ratio
32	8	64	9402021	64.36	3322761	64.01	33105086	64.20	6631582	64.00
64	16	16	9360801	16.05	3077307	16.00	32759352	16.03	6151119	16.00

Table 5.4: 256x64 composed matrix area values (ratio is referred to base matrix area that can be found in 5.1)

To double-check the validity of these result, a MatMul with feasible dimensions is estimated and confronted with actual values, extracted with Design Compiler. This matrix is a 128x32 one, that can be directly compiled and the results are in 2.1. Estimation was made by multiplying by 2 the values of the 64x32 matrix, but it can be made with any other base matrix with little difference.

Power analysis Using the same method power consumption of the MatMul components is analysed too. The first step is again the base MatMul synthesis, reported in table 5.6. In this case the two main contributions to the power consumption are the *Dynamic Power* and the *Leakage Power*; *Total Power* is simply the sum of these two values. Note that in following tables the unit is mW.

Then, the two cases analysed are the same as before: same structure with different parallelism and different structure with same parallelism (respectively tables 5.7 and 5.8.

Conclusions are the same of the area case, with ratios being very similar for

MATRIX 128x32	8 BITS			16 BITS			32 BITS		
Method	**Comb**	**No Comb**	**Tot**	**Comb**	**No Comb**	**Tot**	**Comb**	**No Comb**	**Tot**
SYNTHESIS	2359869	738479	3098348	8217431	1476691	9694123	27814588	2953000	30767588
ESTIMATED	2328363	741026	3069389	8215339	1482415	9697755	27845718	2965015	30810734
|DIFF| %	1.33	0.34	0.93	0.02	0.39	0.04	0.11	0.41	0.14

Table 5.5: Errors between estimated and synthesized area values

MATRIX		8 BITS			16 BITS			32 BITS		
Row	**Col**	**Comb**	**No Comb**	**Tot**	**Comb**	**No Comb**	**Tot**	**Comb**	**No Comb**	**Tot**
32	8	9.521	0.011	9.53	24.269	0.034	24.302	67.346	0.121	64.461
32	16	17.866	0.022	17.888	46.749	0.066	46.816	129.487	0.242	129.721
32	32	35.355	0.043	35.398	89.229	0.139	89.368	253.337	0.484	254.178
64	16	36.302	0.043	36.345	91.818	0.140	91.929	254.103	0.482	249.958
64	32	70.835	0.085	70.921	178.650	0.279	178.877	503.701	0.966	508.249
64	64	139.731	0.183	139.968	355.009	0.558	356.126	967.236	1.927	964.515

Table 5.6: Power values for base MAC matrix

different structures. But again, the other relationship can be easier to use in the estimation phase. This is described in table 5.8: if a matrix is doubled, it will consume nearly two times the power it consumed before.

Remember that ratios are computed with respect to the previous row.

For composed matrices the behaviour is the same: their power consumption is equal to the sum of the power consumption of the base matrices composing them. In table 5.9 there's the 256x64 example.

Eventually, a check on the estimation criteria is needed; the target matrix is still the synthesizable 128x32 one.

Error in power estimation is a little bigger than the area one, especially with lower parallelism, but it can still be considered negligible.

MatMul with bias Majority of the MatMul in the Transformer network are for linear transformations in the form of $X * W + b$. W is the weight matrix and b is the bias vector that is eventually added.

These additions can be included in the MatMul component, like in example 5.19. Bias vector is an horizontal one, so for a specific column the value to be added is the same; therefore one bias value is received for iteration, and by means of <number of rows> adders is summed to the column.

MATRIX		RATIO 16 BITS/8 BITS			RATIO 32 BITS/16 BITS		
Row	**Col**	**Dynamic**	**Leakage**	**Tot**	**Dynamic**	**Leakage**	**Tot**
32	8	2.549	3.027	2.549	2.775	3.607	2.652
32	16	2.617	3.066	2.617	2.769	3.655	2.771
32	32	2.524	3.259	2.525	2.839	3.477	2.844
64	16	2.529	3.249	2.529	2.713	3.439	2.719
64	32	2.522	3.275	2.522	2.819	3.456	2.841
64	64	2.541	3.041	2.544	2.724	3.454	2.708
AVERAGE		2.55	3.15	2.55	2.77	3.51	2.76

Table 5.7: Power ratios between same MAC base matrices with different parallelism

MATRIX		8 BITS				16 BITS				32 BITS			
Row	Col	Dynamic	Ratio	Leakage	Ratio	Dynamic	Ratio	Leakage	Ratio	Dynamic	Ratio	Leakage	Ratio
32	8	9.521		0.011		24.269		0.034		67.346		0.121	
32	16	17.866	1.876	0.022	1.949	46.749	1.926	0.066	1.974	129.487	1.923	0.242	2.000
32	32	35.355	1.979	0.043	1.979	82.229	1.909	0.139	2.103	253.337	1.956	0.484	2.000
64	16	36.302	1.027	0.043	1.008	91.818	1.029	0.140	1.006	254.103	0.983	0.482	0.995
64	32	70.835	1.951	0.085	1.980	178.650	1.946	0.279	1.996	503.701	2.002	0.966	2.005
64	64	139.731	1.972	0.183	2.149	355.009	1.987	0.558	1.995	976.236	1.920	1.927	1.993

Table 5.8: Power ratios between different MAC base structure with same parallelism

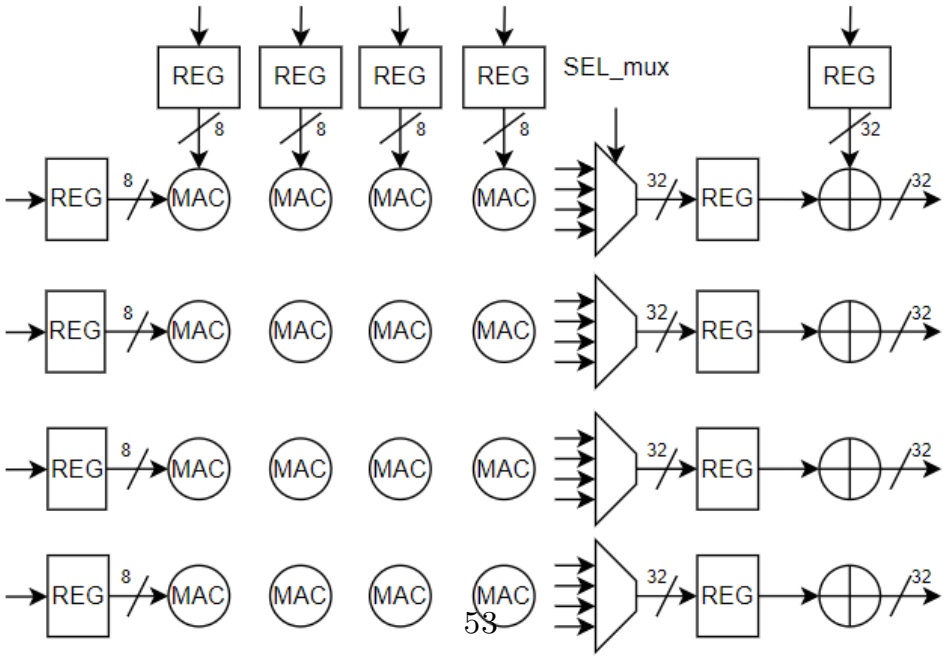

Figure 5.19: 4×4 MatMul with bias addition

MATRIX 256x64			8 BITS				16 BITS			
Row Base	Col Base	# Base Blocks	Dynamic	Ratio	Leakage	Ratio	Dynamic	Ratio	Leakage	Ratio
32	8	64	607.25	63.78	0.711	64.09	1575.8	64.93	2.157	64.26
64	16	16	571.85	15.75	0.690	16.01	1478.6	16.10	2.247	16.04

Table 5.9: 256x64 composed matrix power values (ratio is referred to base matrix power that can be found in 5.6)

MATRIX 128x32	8 BITS			16 BITS			32 BITS		
Method	Dynamic	Leakage	Tot	Dynamic	Leakage	Tot	Dynamic	Leakage	Tot
SYNTHESIS	141.217	0.184	141.387	366.718	0.559	367.421	1023.3	1.935	1008.600
ESTIMATED	141.670	0.171	141.842	357.300	0.559	357.754	1007.402	1.933	1016.499
|DIFF| %	0.32	7.07	0.32	2.57	0.02	2.63	1.55	0.09	0.78

Table 5.10: Errors between estimated and synthesized power values

A special case is analysed in order to see the overhead caused by the bias addition and to verify previous estimation results. This is the MatMul with 8-bit inputs and accumulators on 32 bits, just like the suggested IBERT implementation.

From table 5.11 two things are highlighted: the area overhead due to the bias adders and the ratio of the structure with respect to a reference structure (in this case 32×8). Ratios are very similar to the results of the base component; area increment corresponds to the structure increment, tolerating a little error for estimation purposes. Overhead values are very little, and they decrease increasing the MatMul dimensions, as the MAC components cover almost entirely the area. In table 5.12 there are the power values for these blocks.

Power values don't change much between the two components, hinting that bias addition isn't a relevant contribution to the power consumption.

5.7.2 Other sub-layers' area and power

Other contributions to the hardware of the network comes from non-linear functions and the requantization block. Here synthesis results of these functions are reported.

For combinational blocks like GELU and Requantization, register were applied to their inputs and outputs in order to have also timing results. In fact by changing the clock period Design Compiler gives different synthesis results. This affects the Non Combinational part of the area, keep that in mind when analyzing following values.

Softmax and Layer Normalization are the most complex operations as they include also the division operator, which is not a simple block. This can be improved with better solutions, but for this analysis it is kept as a behavioural description,

MATRIX		BASE MATMUL		BIAS MATMUL		OVERHEAD
Row	**Col**	**Tot Area**	**Ratio wrt** 32×8	**Tot Area**	**Ratio wrt** 32×8	
32	8	286713	1	294741	1	2.80%
32	16	561600	1.96	569656	1.93	1.43%
32	32	1119432	3.90	1127348	3.82	0.71%
64	16	1129445	3.94	1145011	3.88	1.37%
64	32	2236081	7.80	2251670	7.64	0.70%
64	64	4490757	15.66	4506397	15.29	0.35%

Table 5.11: Area comparison between base MatMul block and bias MatMul block

MATRIX		BASE MATMUL		BIAS MATMUL	
Row	**Col**	**Tot Power**	**Ratio wrt** 32×8	**Tot Power**	**Ratio wrt** 32×8
32	8	15.55	1	15.76	1
32	16	29.29	1.88	29.79	1.89
32	32	57.72	3.71	57.81	3.67
64	16	59.18	3.81	59.79	3.73
64	32	115.65	7.44	115.18	7.31
64	64	228.03	14.66	223.89	14.20

Table 5.12: Power comparison between base MatMul block and bias MatMul block

leaving to the synthesizer the design choice.

Remember that these blocks have to be instantiated multiple times, since they process only one element of the corresponding row. Therefore, to have better estimation, the total area of the blocks is multiplied by the correct factor in the last column of the tables. For Softmax and LayerNorm this factor is d while for GELU this is m. Requantization is used in different part of the network, where the number of block can be either m or d; the bigger one is picked to have the bigger value that Requantization need. For example, in the following tables, these values are picked and they are respectively 256 for m and 768 for d, as in the BERT model.

For both area (5.13) and power (5.14) results, an indicative clock period is chosen, larger than the minimum period that gives slack 0. In fact, in this kind of synthesis, the tool find different solutions based on the timing constraint that it has. Usually faster designs have bigger area and power values, but increasing sufficiently the clock period these values become very similar. Therefore, after finding the lowest clock period feasible, discussed in the Timing paragraph 5.7.3, further synthesis were performed with a larger period to extract following values

for area and power.

Component	Comb Area	No Comb Area	Tot Area	Multiple Instances
Softmax	38995	691	39686	30478848
LayerNorm	36571	1566	38137	29289216
GELU	7351	2073	9425	2412800
Requantization	8302	870	9173	7044864

Table 5.13: Area of sub-layers

Component	Dynamic Power	Leakage Power	Tot Power	Multiple Instances
Softmax	755.89	3.28	759.2	583065.6
LayerNorm	234.86	3.42	238.3	183014.4
GELU	76.57	0.551	77.2	19763.2
Requantization	51.63	0.575	52.2	40089.6

Table 5.14: Power of sub-layers

5.7.3 Timing analysis

In order to analyse the timing part of the components, minimum clock period was searched. For each component, a synthesis with zero-period clock was performed to see the starting violated slack: this was the reference value for the following synthesis. Starting from this value, the period was tuned until a synthesis returned slack 0, meaning that the fastest way to design the component was found.

By far, the slowest components are the Softmax and the Layer Normalization, this is expected as they have the most combinational elements, including some division operations, which are not properly optimized.

MatMul latency slightly vary on the dimension of the matrix, but the values are around 1 and 1.5ns. Increasing the MatMul dimension the latency value does not change a lot as the critical path is formed by the multiplier and the adder between the input and the accumulation register. This remain practically unchanged when modifying the matrix dimensions.

Components like GELU and Requantization module have a slightly bigger latency, probably because they are entirely combinational. In fact the are made by simple and few operators. GELU's minimum clock period of 3ns, while the Requantization's one is 2.1ns.

As already said, Softmax and Layer Normalization have the biggest contributions to this kind of analysis. This is because they have several operators, recalling the

architectures 5.14 and 5.6. No registers were inserted apart from the ones needed for the execution, but some pipelining could be applied to have better results, as long as the network processing is modified accordingly. Without any pipelining, Softmax has a minimum clock period equal to 17.7ns, a little larger than the Layer Normalization one, which is 15.3ns. Softmax has more division operators and less registers that can break the combinational paths.

Pipelining and division optimization are two of the possible improvements that can be made in order to lower the clock period and accelerate the execution of these blocks. Clearly, the entire network has to respect all the timing constraint given by its layers, meaning that the slowest block should decide the whole architecture's speed.

5.8 Scaling factor specifics

Overflow could be an issue in the proposed architectures. Instead of dealing with it, prevention can be an option. Many operations in fact have low overflow probability due to data statistics: for instance matrix multiplication should not suffer from overflow since it has inputs on 8 bits and accumulation on 32 bits. After the multiplication between inputs in a MAC, the value is representable with 16 bits, leaving a lot of margin in the accumulation part, given that neural network's matrices are often sparse.

In non-linear functions, for example, multiplications between two n-bit inputs are made on $2n$ bits and eventually truncated. Different bit choices can be made in truncation depending on the obtained result, but any shifting operation in these results has to be made also on the corresponding scaling factor. Scaling factor is indeed one of the overflow handling method; by changing it, out-of-range values can be managed.

Scaling factor impacts also the coefficients of these functions, which have to be compliant to the architecture. Therefore, some constraints have to be applied to control coefficients' value.

5.8.1 Softmax

In softmax computation $x - x_{max}$ is processed, it is a non-positive value going to the exponential value, so the output is limited. Therefore the only things to control are the coefficients for the polynomial and the exponential.

- EXP(5.7): parameter q_{ln2} should be on 32 bit.

$$q_{ln2} = \lfloor \frac{ln2}{S} \rfloor \rightarrow -2^{31} \leq q_{ln2} \leq 2^{31} - 1 \rightarrow |S| \geq 3.23 * 10^{-10} \tag{5.2}$$

- POLY(5.8): q_b can be limited to 16 bits to reduce possible range of the square operation. Poly input is around zero since it is $q_p = q + q_{ln2} * \lfloor -\frac{q}{q_{ln2}} \rfloor$.

  $q_b = \lfloor \frac{b}{S} \rfloor$ with $b = 2.707 \rightarrow -2^{15} \leq q_b \leq 2^{15} - 1 \rightarrow |S| \geq 8.26 * 10^{-5}$
  $q_c = \lfloor \frac{c}{aS^2} \rfloor$ with $c = 1$ $a = 0.358 \rightarrow -2^{31} \leq q_c \leq 2^{31} - 1 \rightarrow |S| \geq 3.6 * 10^{-5}$

- Scaling factor is the same for both sub-blocks so the stricter condition rules.

5.8.2 GELU

- Input scaling factor is divided by $\sqrt{2}$ and then given to the ERF unit. Here S is the value after this scaling.

- ERF(5.11): here the input goes through the absolute value and then it's clipped so its range is limited between 0 and $-b/S$.

- POLY(5.12): q_b can be limited to 16 bits to reduce possible range of the square operation. Poly input is around zero since it is $-\frac{b}{S} = -q_b$ so $0 \leq q + q_b \leq q_b$.

  $q_b = \lfloor \frac{b}{S} \rfloor$ with $b = -1.769 \rightarrow -2^{15} \leq q_b \leq 2^{15} - 1 \rightarrow |S| \geq 5.39 * 10^{-5}$
  $q_c = \lfloor \frac{c}{aS^2} \rfloor$ with $c = 1$ $a = -0.2888 \rightarrow -2^{31} \leq q_c \leq 2^{31} - 1$
  $\rightarrow |S| \geq 4.01 * 10^{-5}$

- q_1 is equal to $q_c \rightarrow q_1 = \frac{1}{S_{erf}} = \frac{1}{aS^2} = \frac{c}{aS^2}$ if $c = 1$.

5.9 Control Unit

Along with the datapath, an hardware design requires also a Control Unit (CU), in charge of managing the operations' flow.

There are many ways to implement a CU, but the common starting point is the control flow, which is analysed here. In 5.20 the encoder's order of operations is depicted. Apart from the Residual Connection, which is combinational and requires the CU only to fetch the right addends, each block is described below. Residual Connection, furthermore, can be incorporated in the last part of the sub-layers, where they give in output the results column by column. This would allow to save clock cycles.

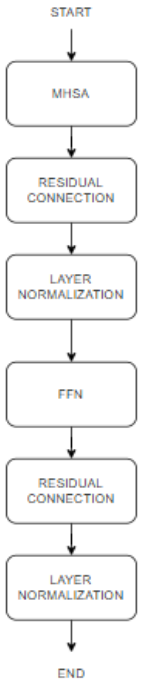

Figure 5.20: Encoder control flow

In the following charts signals *Start* and *Done* can be viewed as place-holders, in fact structure depends on which of these two choices is picked:

1. Single CU for entire Encoder
2. Separate CUs for each sub-layers, controlled by the Encoder CU

In option number 2 some interface is needed to connect the blocks; this is where Start/Done handshaking could be a solution.

5.9.1 MHSA

MHSA has to perform the matrix multiplications to compute Queries, Keys and Values for all heads, then there's the attention part and eventually the last matrix multiplication. This part depends on how many heads can be processed in parallel (k in the chart is number of total heads divided by number of processable heads). Let's see the two extreme cases:

- All heads are computed at the same time: heads computing (dotted line) doesn't need to be iterated and final matrix multiplication has all inputs available.

- One head at a time: attention output can be directly processed by the last MatMul, or stored until the whole matrix is composed. Heads computing have to be iterated for the total number of heads.

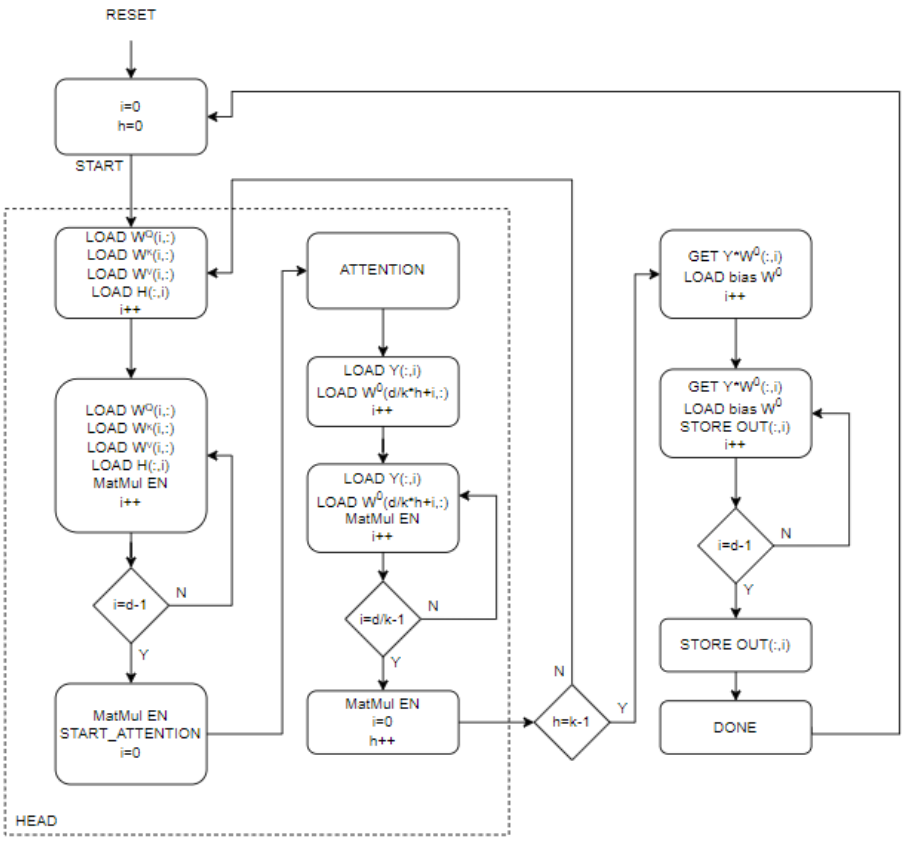

Figure 5.21: MHSA control flow

Attention

Attention operations are:

- $Q * K^T$
- Softmax
- $Softmax\_output * V$

Softmax also needs three phases to be computed and they are included here. There's some room for optimizing the process. Matrix V can be calculated while the $Q * K^T$ product is performed. This would change something also in the MHSA block and can also save resources, as long as matrices Q and K are stored elsewhere. Column selection at the end is made in order to give the attention output to the next operator, so it can be combined into the last MHSA matrix multiplication's flow.

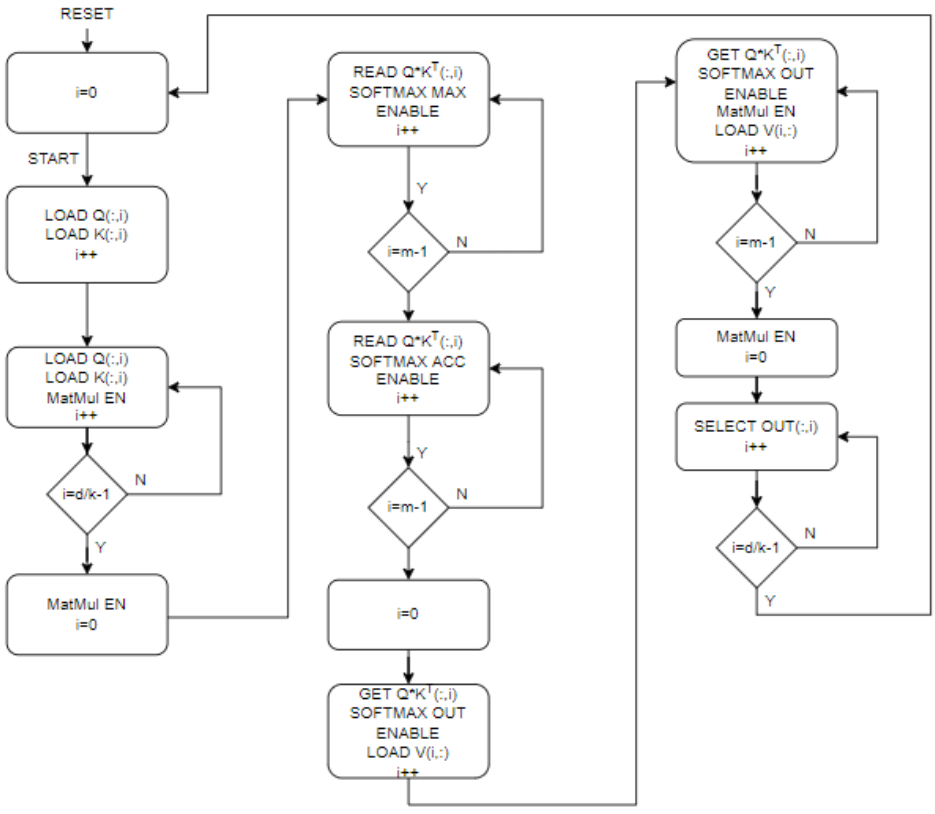

Figure 5.22: Attention control flow

5.9.2 FFN

FFN has two linear transformations separated by a GELU activation. GELU can be made in a single iteration, so it is combinational. Therefore there are three identical iterations over the matrices, two for computing the two matrix multiplication and the last one to add the biases to the FFN results and give them in output. Layer output can be either stored or transported to the next layer, with the final iteration corresponding to the entry one in the next layer.

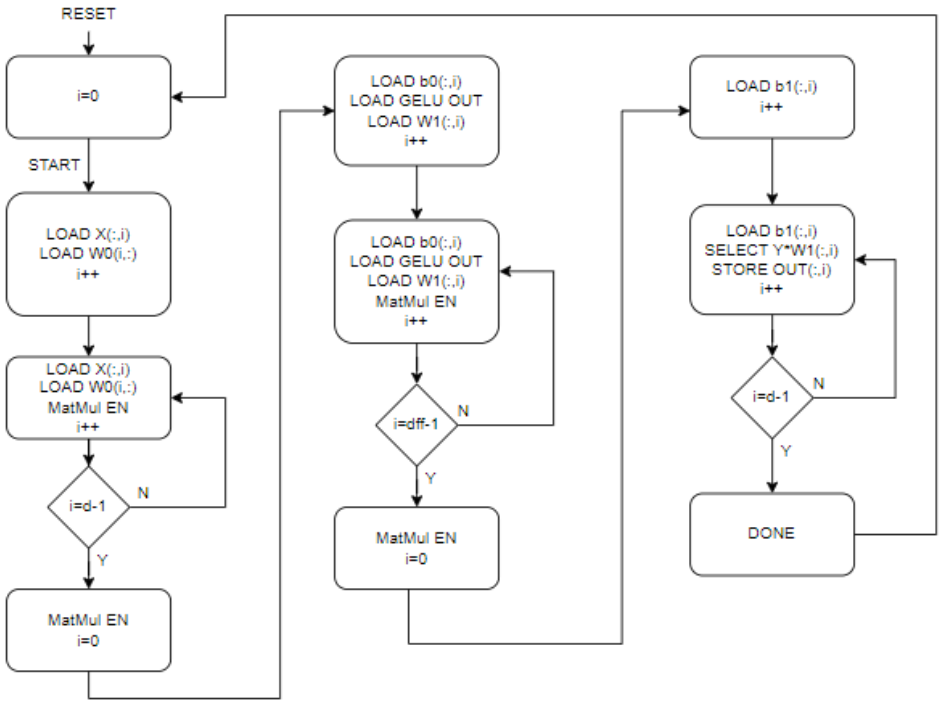

Figure 5.23: FFN control flow

5.9.3 LayerNorm

LayerNorm requires three iterations over the input to compute respectively the mean, the variance and the results. Plus, the square root unit needs an unprecised number of cycles to compute the standard deviation, and it is handled by a *VALID* signal coming from the unit.

By confronting the above charts, it is clear that one sequence is repeated: the matrix multiplication sequence. This can be realized with a sub-procedure, depending on what choices are made in the CU design.

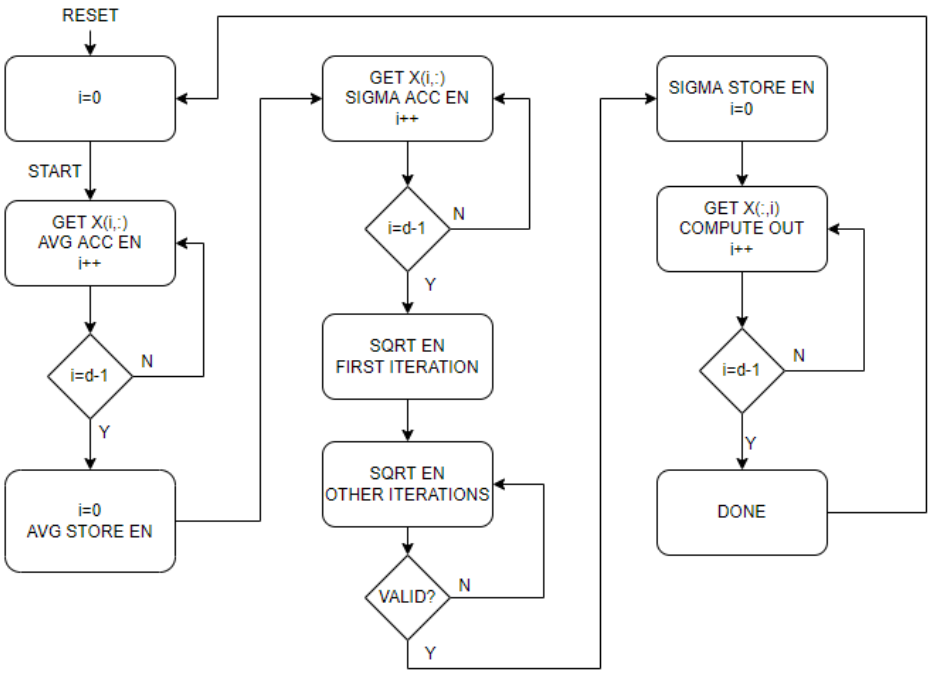

Figure 5.24: Layer Normalization control flow

Chapter 6

Configurability

6.1 MatMul Utilization

MatMul operation is by far the most needed one in the Transformer network, and to see the configurability to different models this is the first element to analyse.

Keeping in mind the network's parameters (2.4), the matrix multiplications required by a Encoder layer are:

- $m \times d/k$ in the Q,K,V computation of the MHSA (3 of them for each head).
- $m \times d$ in the MHSA final computation.
- $m \times m$ in the attention $Q * K^T$ computation (1 for each head).
- $m \times d/k$ in the Attention final computation.
- $d \times d_{ff}$ in the FFN first matrix transformation.
- $d \times d$ in the FFN second matrix transformation.

The easiest way to implement them is having one different component for each of these operations can be intensive from the area/power point of view, so in the hardware definition some choices can be made to reuse some resources and save area.

Since $m \times d/k$ is the smallest dimension among the listed operations, it can be considered the base Processing Element (PE) that can be replicated to form larger sets. Remember that subsequent sets require that the first one keeps stored its values until they are entirely received and processed by the following one. So, if an operation wants to reuse some PEs that are involved in the previous one, these values that are exchanged by the two PEs should be stored elsewhere to "clean" the MACs. For example, this can be critical for the head part where Q, K, V sets are

3 for each head and their outputs have to be processed almost at the same time in the following Attention.

Clearly there are a lot of different options in choosing the amount of PEs to implement and how to use them to exploit all needed multiplications.

Here are some examples of configuration to better understand the problem. Parameters are fixed in this example and each set have a color to highlight them in the scheme:

- $m \times d/k = 256 \times 64$ (Q, K, V) $\rightarrow$ 1 base PE.
- $m \times d = 256 \times 512 \rightarrow$ 8 base PEs.
- $m \times m = 256 \times 256 \rightarrow$ 4 base PEs.
- $m \times d/k = 256 \times 64$ (Attention) $\rightarrow$ 1 base PE.
- $d \times d_{ff} = 512 \times 2048 \rightarrow$ 64 base PEs.
- $d \times d = 512 \times 512 \rightarrow$ 16 base PEs.

Configuration possibilities:

1. **16 base PEs** $\rightarrow 256x64$ **(total** 512×512**)**, scheme in 6.1:
   - This scheme accounts for two heads computed at every cycle (1 head = 3 green PEs + 1 orange set + 1 blue set): this way their results can be kept in loco.
   - Partial results of burgundy set should be saved in order to update them at every iteration.
   - $d \times d$ is formed by all PEs.
   - $d \times d_{ff}$ set uses all PEs multiple times. IN this case 4 times to reach d_{ff} dimension.
2. **16 base PEs** $\rightarrow 256x64$ **(total** 512×512**)**, scheme in 6.2:
   - this maximizes the number of heads that can be computed, in this case 4. Since green and orange sets use same PEs, Q, K and V matrix must be stored in memory. The same happens between orange and blue sets.
3. **64 base PEs** $\rightarrow 256 \times 64$ **(total** 512×2048**)**:
   - This solution allocates every MAC needed in the FFN architecture, allowing the $d \times d_{ff}$ multiplication to be computed in one take.
   - Since a single head without reutilization requires 8 base PEs (see first example), in this solution 8 heads can be processed simultaneously.

4. **Other options:**
    - Using less PEs than the first two examples leads to some more iteration to perform MatMuls larger than the available operators.
    - Keeping the same structure, other intermediate solutions between the first and second solution can be adopted. Main changes can be made on the number of heads processed in parallel since they require the smallest set of all.

Figure 6.1: MatMul scheme example 1

Figure 6.2: MatMul scheme example 2

Options 1 and 2 are the example of how the same total structure can be used differently; in fact they are identical in the datapath, but they differ in the control part that is in charge of managing the interface between sub-layers.

$m \times d/k$ MatMul can be viewed as the fundamental block of the entire network: in fact other than being the smallest required set of MACs, by looking at the parameters table (2.1) this ratio is common among different type of Transformer models. m depends on the input sequence length, so in other words, once it is fixed, it is the maximum processable sequence length; sentences shorter than that are padded with zeros. Fixing d/k mean that only Transformer models with that ratio between model dimension and number of heads can be adapted to the hardware model. Assuming this, base PE $m \times d/k$ is the reference for every model.

Recalling the MatMul synthesis analysis, area and power values in any configuration can be estimated by looking at the total dimension of the structure. Area and power consumption are indeed two of the main criteria to choose how many base PEs to instantiate, considering the tradeoff between area and parallelizability.

One idea to ease the reutilization of the MatMul components is adding an extra register to the internal MACs: this can be used like a "retention register" that can be loaded when the corresponding set is needed for another operation but the accumulated data still has to be processed. Instead of moving the data to the memory, it can be kept in loco while the MACs are "cleared" and can be reused in another matrix multiplication. This obviously would have a cost in hardware resources, like a pipeline for the entire output matrix. It is an option that can be applied only to certain MatMul sets, depending on the design choice adopted; to the sets that are not reused between subsequent operations.

6.2 Hardware Adaptability

Once the hardware resources are fixed for a specific Transformer model, can it be used for different models?

This is the topic of this section, where the hardware proposed so far is analysed in terms of parameter variation. In fact, in 2.1 it's evident that one big difference between Transformer models are the values of their parameters.

Structure differences As already explained, Transformer, Universal Transformer and BERT have some little differences in the network structure. Transformer and Universal Transformer are identical in the Encoder and Decoder structures. However, they have two components that are not present in BERT; one is the $ReLU$ activation function used in the FFN. This is simpler than the GELU and its implementation is quite direct. ReLU in fact mask every negative input, giving in output 0 if the input is negative or the input itself if it's positive. This can be translated into hardware with some multiplexers with 0 and the input data as inputs and the input's sign bit as a selector.
It is legit to think that GELU can be the right activation function for these other networks since the behaviour of the two functions is very similar, but it should be verified in a software model.

The other change in the structure is in the Transformer Decoder, where there is a additional sub-layer, namely the *Masked Multi-Head Attention*. This requires the masking operation, that in a way is similar to the ReLU as it mask with a 0 output some neurons, but the choice is made through an external mask. So the

multiplexers can be used once again but the selection signal comes from outside, e.g. where the mask is stored.
For the *Encoder-Decoder attention* the operations are the same as in the normal Attention, but Key and Query matrices are computed from the Encoder output; therefore they should have an input different from the Value matrix.
Hence, in order to execute Transformer/Universal Transformer models, these hardware addition have to be made, otherwise the hardware solution cannot be flexible with respect to different Transformer networks.

Model Variations Adaptability to different model can be discussed by looking at how the variation of the model's parameters can be handled to fit the model onto the available hardware. Smaller models can be adapted quite easily as there's no need for extra resources, and smaller matrices can processed entirely. Surely in this case models are not optimized, with some operators that become useless, unless the control part is complex enough to handle that and use resources at the maximum efficiency for every model.
A big part of this feature depends in fact on the control part of the network, since the main changes in the execution are not datapath-related. For example, any variation on the number of columns of the to-be-processed matrices does not affect the MatMul component itself but the number of iterations needed to complete the multiplication; that's because the architecture is designed to process one column at a time.

Remind that only models with a specific ratio d/k can be taken into account, due to simplicity and hardware limitations. So, the other parameters that can vary between models are m, d and d_{ff}. m can be assumed constant and fixed; a maximum sequence length can be decided not to overcomplicate things.

Therefore, the following considerations are on the required adaptation when d or d_{ff} are increased.

1. **Increasing** d_{ff}: d_{ff} is used in the FFN alone (scheme 5.9), where the weight matrix W^0 can be partitioned in vertical slices so that $Y = x * W^0$ can be calculated with multiple iterations until all parts are covered. Partial results of the matrix Y can either be stored or go directly to the second MatMul. Final matrix multiplication can be made entirely when all its inputs are available or can follow the first matrix multiplication iterations. In the latter option whenever partitioned columns from the first MatMul are completed, they go through the GELU operators and onto the second MatMul, that accumulates results until all partial matrices are scanned. Also weight matrix W^1 should be partitioned horizontally, but this affect only the fetch part of this matrix,

since it is already scanned by rows, so it is only a problem to get the right row for each iteration.

2. **Increasing** d: d is the fundamental dimension of the Transformer so it is trickier to change. First it appears in MHSA (scheme 5.4): the final multiplication has a weight matrix W^0 that can be partitioned like in the previous case. This MatMul is already depending on how many heads are processed in parallel so this add an extra variable to the control part of this unit. For the input part of the MHSA the only thing to change is the number of iterations to read all the input matrix, as the heads blocks are unchanged due to the fixed $m \times d/k$. The other sub-layer affected is the FFN (5.9), where the matrices to be sliced are the input X and the weight W^1. Horizontal sub-matrices of the input can be processed as a standalone producing partial horizontal results that once in memory form the whole result matrix. They can also go on to the successive layers like the Residual Connection and the Layer Normalization.

Chapter 7

Conclusions

Hardware design of a Transformer network is something not too explored in previous works. That's why having some layer architectures for the Transformer's operations is a good way to progress. Accelerating this kind of models could become a real gain, but it has to be weighted with the hardware costs. In this sense, the analysis given in this work can help, with some insides on area, power and timing contributions of each layer. Depending on the wanted model, a probable estimation of these values can be made.

Fully-integer Transformer can lead to several advantages for hardware realization and this design goes in that way. It integrates some algorithms proposed in previous works with the general idea of the network to have an integer-only implementation. In fact, besides the complex sub-layers that required some mathematical manipulation to be linearized, this design proposes a solution for each level of the Transformer, from the top-level view to the inner components.

This work keeps also an eye on the general part of the starting problem. The designed layers can be used for realizing different Transformer models, as long as the precision loss is acceptable. Performances can indeed change a lot going from model to model. Moreover, different choices can be made in the same model, with a tradeoff in complexity between datapath and control unit.

7.1 Further Developments

This work leaves room for future developments, in almost every aspect of it. A fundamental part is the control unit, whose implementation is not treated here and can be realized in several ways; the choice has to be made along with the datapath decisions. To complete an entire Transformer, control unit and memory are the two keys. Transformer is known for having a lot of parameters and weights, so memory can be a complex aspect, given dimension and latency constraints.

On the architecture side, it can be worthy to try some optimizations. A lot of works proposed different expressions to compute Softmax and Layer Normalization and some ideas can be orthogonal to this design, even though their application on integer hardware it is not straightforward. Furthermore, in some layers the division operator limits the execution of the layer itself: other implementations of this operator could give some benefit. Pipelining also is a easy way to improve the timing results, paying an overhead for the insertion of more registers to improve speed.

Eventually, there's the need to see performances on the target tasks of the Transformer. It is important to analyze accuracy loss with respect to same floating-point model and overall score on the general benchmarks for NLP.

Bibliography

[1] Sehoon Kim, Amir Gholami, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. «I-BERT: Integer-only BERT Quantization». In: (2021). DOI: 10.48550/ARXIV.2101.01321. URL: https://arxiv.org/abs/2101.01321 (cit. on pp. 2, 27, 28).

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is all you need». In: *Advances in neural information processing systems*. 2017, pp. 5998–6008 (cit. on pp. 4, 6).

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

[4] Ruibin Xiong et al. «On layer normalization in the transformer architecture». In: *International Conference on Machine Learning*. PMLR. 2020, pp. 10524–10533 (cit. on p. 10).

[5] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. «Universal transformers». In: *arXiv preprint arXiv:1807.03819* (2018) (cit. on p. 12).

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «Bert: Pre-training of deep bidirectional transformers for language understanding». In: *arXiv preprint arXiv:1810.04805* (2018) (cit. on pp. 14, 15).

[7] Dan Hendrycks and Kevin Gimpel. «Gaussian error linear units (gelus)». In: *arXiv preprint arXiv:1606.08415* (2016) (cit. on p. 15).

[8] 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: *arXiv preprint arXiv:2009.08605* (2020) (cit. on pp. 18, 24, 34).

[9] Xiao Dong, Xiaolei Zhu, and De Ma. «Hardware Implementation of Softmax Function Based on Piecewise LUT». In: *2019 IEEE International Workshop on Future Computing (IWOFC*. 2019, pp. 1–3. DOI: 10.1109/IWOFC48002.2019.9078446 (cit. on p. 18).

[10] 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. 18, 19).

[11] Gaoming Du, Chao Tian, Zhenmin Li, Duoli Zhang, Yongsheng Yin, and Yiming Ouyang. «Efficient Softmax Hardware Architecture for Deep Neural Networks». In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. GLSVLSI '19. Tysons Corner, VA, USA: Association for Computing Machinery, 2019, pp. 75–80. ISBN: 9781450362528. DOI: 10.1145/3299874.3317988. URL: https://doi.org/10.1145/3299874.3317988 (cit. on p. 20).

[12] Xue Geng, Jie Lin, Bin Zhao, Anmin Kong, Mohamed Aly, and Vijay Chandrasekhar. «Hardware-Aware Softmax Approximation for Deep Neural Networks». In: May 2019, pp. 107–122. ISBN: 978-3-030-20869-1. DOI: 10.1007/978-3-030-20870-7_7 (cit. on p. 20).

[13] Jacob R. Stevens, Rangharajan Venkatesan, Steve Dai, Brucek Khailany, and Anand Raghunathan. *Softermax: Hardware/Software Co-Design of an Efficient Softmax for Transformers*. 2021. DOI: 10.48550/ARXIV.2103.09301. URL: https://arxiv.org/abs/2103.09301 (cit. on pp. 20, 21).

[14] Biao Zhang and Rico Sennrich. «Root mean square layer normalization». In: *Advances in Neural Information Processing Systems* 32 (2019) (cit. on p. 23).

[15] Toan Q Nguyen and Julian Salazar. «Transformers without tears: Improving the normalization of self-attention». In: *arXiv preprint arXiv:1910.05895* (2019) (cit. on p. 23).

[16] Meiqi Wang, Siyuan Lu, Zhu Danyang, Jun Lin, and Zhongfeng Wang. «A High-Speed and Low-Complexity Architecture for Softmax Function in Deep Learning». In: Oct. 2018, pp. 223–226. DOI: 10.1109/APCCAS.2018.8605654 (cit. on p. 24).

[17] Richard Crandall and Carl B Pomerance. «Prime numbers: a computational perspective. Vol. 182». In: *Springer Science & Business Media. https://doi.org/10.2307/3621190* 10 (2006), p. 3621190 (cit. on p. 31).

[18] Zhewei Yao et al. «Hawq-v3: Dyadic neural network quantization». In: *International Conference on Machine Learning*. PMLR. 2021, pp. 11875–11886 (cit. on p. 33).