To appear at the 2023 International Joint Conference on Neural Networks (IJCNN), Queensland, Australia, June 2023.

SwiftTron: An Efficient Hardware Accelerator for Quantized Transformers

Alberto Marchisio[1,*], Davide Dura[2,*], Maurizio Capra[2,*], Maurizio Martina[2], Guido Masera[2], Muhammad Shafique[3]
[1]*Technische Universität Wien, Vienna, Austria* [2]*Politecnico di Torino, Turin, Italy* [3]*New York University, Abu Dhabi, UAE*
Email: alberto.marchisio@tuwien.ac.at, s276493@studenti.polito.it, maurizio.capra@polito.it
maurizio.martina@polito.it, guido.masera@polito.it, muhammad.shafique@nyu.edu

***Abstract*—Transformers' compute-intensive operations pose enormous challenges for their deployment in resource-constrained EdgeAI / tinyML devices. As an established neural network compression technique, quantization reduces the hardware computational and memory resources. In particular, fixed-point quantization is desirable to ease the computations using lightweight blocks, like adders and multipliers, of the underlying hardware. However, deploying fully-quantized Transformers on existing general-purpose hardware, generic AI accelerators, or specialized architectures for Transformers with floating-point units might be infeasible and/or inefficient.**

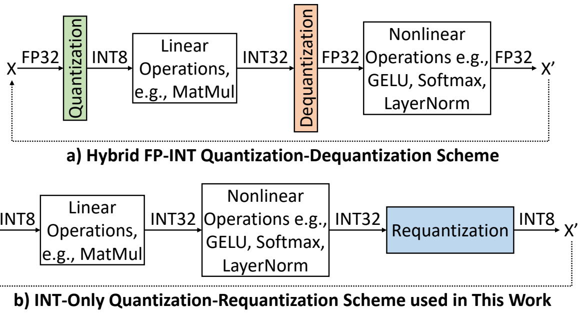
Towards this, we propose *SwiftTron*, an efficient specialized hardware accelerator designed for Quantized Transformers. *SwiftTron* supports the execution of different types of Transformers' operations (like Attention, Softmax, GELU, and Layer Normalization) and accounts for diverse scaling factors to perform correct computations. We synthesize the complete *SwiftTron* architecture in a 65 nm CMOS technology with the ASIC design flow. Our Accelerator executes the RoBERTa-base model in 1.83 ns, while consuming 33.64 mW power, and occupying an area of 273 mm^2. To ease the reproducibility, the RTL of our *SwiftTron* architecture is released at https://github.com/albertomarchisio/SwiftTron.

***Index Terms*—Hardware Architecture, Transformers, Machine Learning, ASIC, Quantization, Attention, Softmax, Layer Normalization, GELU.**

I. Introduction

Among advanced Machine Learning (ML) models, Transformers are becoming mainstream for several applications like natural language processing and computer vision. However, they involve several compute-intensive operations like Multi-Head Self Attention and Layer Normalization with massive data streams. Hence, their execution is highly power-consuming when conducted on general-purpose hardware. Specialized architectures would be desirable to accelerate the execution of Transformers' operations and improve the energy-efficiency.

Several types of ML accelerators have been recently integrated with the most common chips to execute massive matrix multiplications that are typical in convolutional and fully-connected layers [1] [2] [3] [4] [5]. However, such generic architectures do not support some Transformer-specific operations, such as Attention, Softmax, Gaussian Error Linear Unit (GELU), and Layer Normalization. Hence, the neural accelerators need to be tailored for such unique operations involved in Transformers. Towards this, some architectures have recently been proposed. For instance, OPTIMUS [6] optimizes the execution of matrix multiplications in transformers, and A^3 [7] accelerates the execution of the Attention operation. However, these architectures only execute a certain part of a given Transformer, and do not provide a holistic acceleration platform for the complete Transformer. Note that other functions like Softmax, GELU, and Layer Normalization involve nonlinear operations that cannot be easily implemented in integer arithmetics. A common way of handling these operations is to iteratively quantize the input to compute matrix multiplication with integers and de-quantize the intermediate results to calculate the nonlinear operations in floating point arithmetic [8]. A simplified scheme of this method is depicted in Figure 1a. A similar approach is also adopted in the recent Transformer Engine of the NVIDIA H100 Tensor Core GPU [9], which implements the operations in FP8 arithmetic. However, this approach would require the design of several floating point arithmetic logic units with significant resource overheads compared to the integer counterparts.


X
FP32
Quantization
INT8
Linear Operations, e.g., MatMul
INT32
Dequantization
FP32
Nonlinear Operations e.g., GELU, Softmax, LayerNorm
FP32
X'
X
INT8
Linear Operations, e.g., MatMul
INT32
Nonlinear Operations e.g., GELU, Softmax, LayerNorm
INT32
Requantization
INT8
X'

a) Hybrid FP-INT Quantization-Dequantization Scheme

b) INT-Only Quantization-Requantization Scheme used in This Work

Fig. 1. Simplified diagrams showing the computation flows for **(a)** the Quantization-Dequantization scheme of [8], and **(b)** the quantization-based scheme adopted in this work that employs *Requantization* blocks.

As a motivating experimental analysis comparing different arithmetics, we synthesize an INT8-adder, an INT8-multiplier, an FP32-adder, and an FP32-multiplier [10] in a 65 nm CMOS technology node with the Synopsys Design Compiler and evaluate latency, power, and area. Figure 2 shows the respective overhead of FP32 arithmetic operators compared to their respective INT8 implementations. These experiments

*These authors contributed equally to this work.

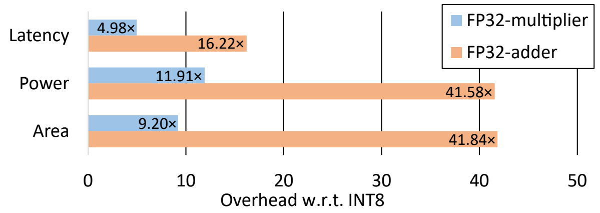


Fig. 2. Latency, power, and area overhead of a single adder and a single multiplier implemented in FP32 arithmetic, compared to their respective INT8 implementations.

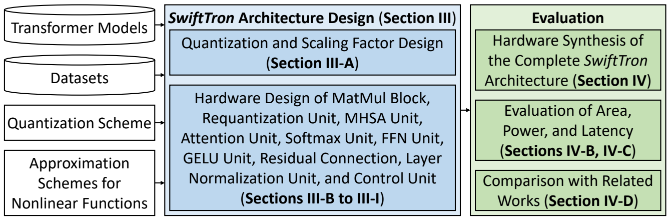

Transformer Models
Datasets
Quantization Scheme
Approximation Schemes for Nonlinear Functions
SwiftTron Architecture Design (Section III)
Quantization and Scaling Factor Design (Section III-A)
Hardware Design of MatMul Block, Requantization Unit, MHSA Unit, Attention Unit, Softmax Unit, FFN Unit, GELU Unit, Residual Connection, Layer Normalization Unit, and Control Unit (Sections III-B to III-I)
Evaluation
Hardware Synthesis of the Complete SwiftTron Architecture (Section IV)
Evaluation of Area, Power, and Latency (Sections IV-B, IV-C)
Comparison with Related Works (Section IV-D)

Fig. 3. Overview of our novel contributions in this work.

show that the potential savings are about one order of magnitude.

On the other hand, computing nonlinear operations with quantized integers without incurring significant accuracy loss is nontrivial. The state-of-the-art in this regard is represented by the I-BERT [11], a framework that implements specific approximations to execute all the nonlinear operations of the Transformers with integer-only arithmetic. However, this work implemented the Transformers on general-purpose hardware, i.e., GPU. A specialized accelerator executing all the Transformer layers, including nonlinear operations, using only efficient integer arithmetic is missing, and highly required.

A. *Our Novel Contributions*

The above-discussed limitations motivate us to propose *SwiftTron*, an efficient hardware accelerator that executes quantized Transformers with integer-only operations (see Figure 1b for our integer-only arithmetic flow), while focusing on multiple different operations of a given Transformer to provide a high degree of performance/energy efficiency. Our contributions are discussed in the following list (see Figure 3).

- We design the *SwiftTron* hardware architecture, a specialized accelerator composed of several hardware units to execute different operations of the Transformers. (**Section III**)
- We design and implement a quantization scheme for Transformers with scaling factors to correctly execute the linear operations with INT8 and the nonlinear operations with INT32 arithmetics. (**Section III-A**)
- We synthesize the complete *SwiftTron* architecture in a 65 nm CMOS technology node with the Synopsys Design Compiler and conduct gate-level simulations to measure the area and power consumption. (**Section IV**)
- We compare the key features of our accelerator with the related work to highlight that our *SwiftTron* is the first architecture that complies with all the desired features. (**Section IV-D**)
- Towards encouraging fast advancements in the neural hardware accelerator and ML community, and to ease the reproducibility of our experiments, we open-source the complete RTL of our *SwiftTron* accelerator architecture at https://github.com/albertomarchisio/SwiftTron.

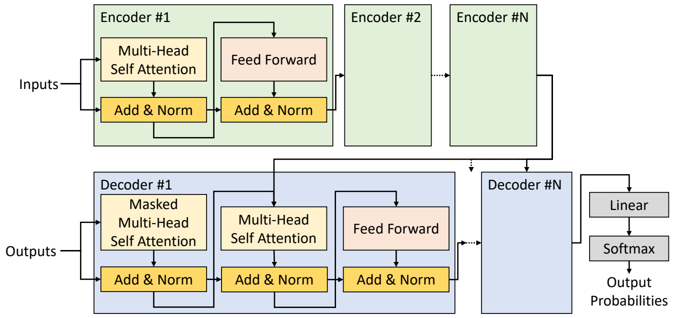


Fig. 4. Architectural model of the Transformer [12], where inputs and outputs are taken after the positional encoding operations.

II. Background and Related Work

A. *Transformer Models*

The Transformer network has been introduced in [12], which is the reference point of the related works and subsequent models. Transformers are formed of two main blocks, the encoder and the decoder. They are composed of the following layers[1]:

- Multi-Head Self Attention (MHSA):
  - Linear Transformation to compute Query (Q), Key (K), and Value (V) matrices for each head.
  - Attention:
    - $Q \cdot K^T$ multiplication, where K^T denotes the transposed matrix K
    - $Softmax(Q \cdot K^T) \cdot V$ multiplication
  - Linear Transformation after concatenating every head Attention output.
- Residual Connection & Layer Normalization in the MHSA
- Feed-Forward Network (FFN):
  - Linear Transformation
  - Activation Function
  - Linear Transformation
- Residual Connection & Layer Normalization in the FFN

When repeated multiple times, these layers form the structure of the encoder and the decoder, as shown in Figure 4. Note that the typical activation functions for Transformers[2] are the *ReLU* [13] and the *GELU* [14]. The baseline Transformer [12] is composed of $N = 6$ Encoder layers and $N = 6$ Decoder layers, while many different architectures

[1] Note that, since there is little difference between the encoder and the decoder, the composition of their layers is very similar.

[2] There exist a few other options, but these are the most commonly used activation functions in Transformers.

have been proposed. The work of [15] proposed a tunable Transformer model where the same Encoder/Decoder layer is used until the computation reaches the desired results. This design led to having only one instance of these layers and the same parameter set for each iteration, which either can be fixed a priori or adapted dynamically during inference. On the other hand, the BERT-like architectures [16] rely only on the Encoder part of the network, achieving state-of-the-art performances in several tasks.

The *Attention* is the key operation of a Transformer, unleashing overall better performance than other ML architectures. However, due to the necessary non-linearities, its compute-intensive operations challenge typical neural hardware accelerators. Moreover, Transformers' massive sizes, like the OpenAI GPT-3 [17] with 96 hidden layers and 175 B parameters, lead to large memory footprint, latency, and power consumption, making their hardware execution extremely resource-demanding.

Some works [18] [19] proposed mathematical manipulations for reducing the complexity of the needed operators in functions like Softmax and Layer Normalization. Other works inspired by compression techniques like pruning or knowledge distillation reduce the model's operations and parameters [20]. *Note that these approaches are orthogonal to our work.*

A potential technique that has the higher impact on implementing Transformer networks onto hardware devices is *quantization*, which transforms the floating-point values, universally used in the Transformer models, into integer values while trying to minimize the consequent precision loss. For instance, the I-BERT [11] implements the entire BERT network [16] with integer operations. This process consists of an efficient way to have simpler operators and lighter number representation, helping both the resources and memory constraints for an efficient accelerator design. Similarly, the I-ViT [21] quantizes the Vision Transformer [22] for image classifiation. *However, these works are deployed on GPUs, while our focus is on specialized hardware accelerators.*

B. AI Hardware Accelerators Executing Transformers' Operations

Generic AI hardware accelerators are based on arrays of Processing Elements (PEs) [23]. Adapting an existing accelerator to compute the dot product between the query, key, and value matrices in the MHSA mechanism would require several modifications in its architecture and would not be straightforward. Therefore, designing specialized hardware architectures for Transformers is highly desirable.

Recent works proposed hardware designs for executing some of the Transformer's operations efficiently. However, they primarily use floating-point representation, which is complex and expensive in terms of hardware resources. To the best of our knowledge, there are no accelerators in prior works implementing the entire Transformer network using only integer computations and simple approximations for nonlinear operations. Still, the work in [24] presents an interesting hardware architecture for MHSA and FFN. Besides several novel additions, our work also takes some inspiration from these works regarding the column-oriented computations. In fact, *our data flow is designed to use one column at a time of the matrices under processing. Since the matrix multiplication, which is the most used operation in Transformers, requires the input to be read column-by-column according to its algorithm, this data flow structure enables a simple interface between blocks.* The architecture proposed in [6] proposes an optimization for the hardware matrix multiplication in the Transformer. The work in [25] analyzes the network design considering the hardware latency in the process. The work in [26] evaluates the performance of a floating-point implementation on CPUs and GPUs.

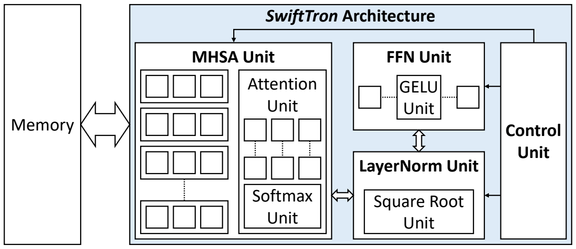


Fig. 5. Top-level overview of our *SwiftTron* architecture.

The main drawbacks of these related works from the hardware execution perspective are the following:

1) The Floating Point representation is employed in some computations [8] [27] [28]. This is a simulated quantization process where almost every variable is quantized. However, the GELU, Softmax, and LayerNorm are computed with floating-point operations. It not only increases the hardware complexity, but it also requires both quantization and dequantization layers to convert data between blocks.
2) Complex operators like exponential and square root are implemented with expensive LUTs [29] [30], or using different approaches that involve FFTs [31].

To overcome these issues, our proposed SwiftTron architecture executes all the layers and functions in transformers using only integer computations.

III. SwiftTron: Hardware Architecture Design

This section describes each hardware block designed for executing the Transformers' layers (as discussed in Section II-A). The layers are mainly composed of linear transformations and nonlinear functions. Since the linear layers are based on matrix multiplications, a *MatMul* block is designed. On the other hand, to execute nonlinear functions, namely the *Softmax*, *GELU*, and *Square Root* for the Normalization, our work deploys second-order polynomial approximations and recursive implementation, which are based on the concepts from [11]. A top-level view of our proposed *SwiftTron* architecture is shown in Figure 5.

The Transformer's main parameters are the model dimension d, the number of heads k, the sentence length m, and the feed-forward dimension d_{ff} (usually two or four times the value of d).

A. Quantization and Scaling Factor Design

Before going into detail of each component, it is important to highlight that quantized values have their corresponding scaling factors derived during the quantization process.

Formally, given a a floating-point value and q_a its quantized value, the scaling factor S_a is defined such that $a = Q_a S_a$. Scaling factors allow the transformations from floating point to integer and vice versa, and determine the correct operation between two integers. For example, two numbers with different scaling factors cannot be directly summed together, but an extra component is needed for the *Residual Connection*. Furthermore, the scaling factors are fundamental for computing the algorithm's coefficients, especially in the nonlinear functions. Some constraints on the scaling factor values are applied to have representable coefficients and to limit the risk of overflow. A scaling factor is a floating-point number that is not directly included in the architecture to avoid FP operators, but its value is fixed for each layer at design time. Another critical aspect to consider is the data representation in the architecture. The matrix multiplications are conducted with INT8 inputs and INT32 accumulators. The nonlinear functions operate on INT32 to avoid excessive accuracy loss. Therefore, a *Requantization* block is needed to bring the INT32 values back to INT8 as input to the subsequent MatMul operations.

Dealing with the scaling factor in linear operations is straightforward. For instance, the multiplication between to numbers (a and b) with different scaling factors is defined as $a \cdot b = q_a S_a \cdot q_b S_b = (q_a \cdot q_b)(S_a \cdot S_b)$. This property holds for matrix multiplications (MatMul), since its resulting expression is $MatMul(Sq) = S \cdot MatMul(q)$.

However, transformers contain several nonlinear operations, such as GELU, Softmax, and Layer Normalization, for which this property does not hold. For this reason, these operations can be either approximated using a second-order polynomial, as in the case of GELU and Softmax, or computed iteratively, like for the square root of the Layer Normalization.

B. Hardware Architecture of the MatMul Block

The MatMul block is extremely important since it is used extensively in the following designs. It is formed by $\#rows \times \#columns$ Multiply-and-Accumulate (*MAC*) elements that, at every iteration, receive their corresponding row and column input and update the accumulator. Figure 6 reports an example with a 4×4 MAC array. After all the inputs are scanned, they store the final output that can be read column-by-column by tuning the selector of the output multiplexer. In the following layers, several MatMul operations are required with different dimensions, but depending on what type of Transformer workload to execute, these components can be shared and/or reused.

The bias addition for the linear transformations can be incorporated into the component, like in this example. The bias is added when reading the output matrix. Hence, a different value is added to each column. For multiplications that do not require bias, this process can be ignored.

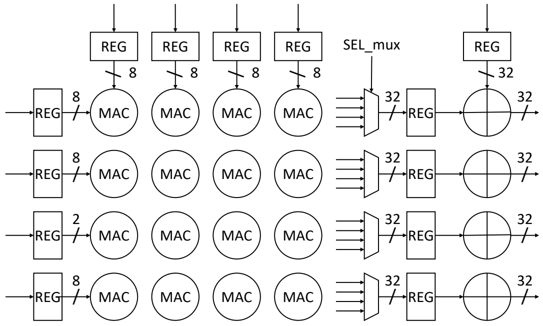


Fig. 6. Architectural view of a 4×4 MatMul block with bias.

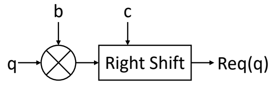


Fig. 7. Architecture details of the Requantization (Req) unit.

C. Hardware Architecture of the Requantization Unit

A scaling factor change is necessary to reduce the precision from 32 bits to 8 bits. To perform this transformation, the *Dyadic Numbers* concept [32] is involved. Starting from the 32-bit representation of a value a, denoted with its quantized value q_a and its quantization scale S_a such that $a = q_a S_a$, the final representation should be $o = q_o S_o$ with q_o on 8 bits. Hence, equalling a and o since the real value must remain unchanged, the formula is derived in Equation (1).

$$q_a S_a = q_o S_o \longrightarrow q_o = q_a \frac{S_a}{S_o} \tag{1}$$

Remembering that the scaling factors are not strictly integers but can assume any real value, this expression cannot be implemented directly on integer-only resources. Hence, the scaling factor ratio is represented with a dyadic number, a rational number in the format of $b/2^c$, where b and c are two integer numbers. The final expression is shown in Equation (2).

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

This convention also avoids the need to use dividers, as the required resources are only an INT32 multiplication and a one-bit shifting, as shown in Figure 7.

D. Hardware Architecture of the MHSA Unit

The MHSA block is responsible for computing the correspondent operation in a Transformer. A single head, whose architecture is shown in Figure 8, is composed of three MatMul blocks with inputs connected to Query, Keys, and Values, and an Attention operator. Figure 9 shows an example of the complete MHSA architecture composed of 4 heads and another MatMul block that generates the outputs.

The choice of the number of heads to be computed in parallel depends on the available hardware resources. Different architectural configurations can be designed, from processing one head at a time to computing all heads concurrently. Therefore, data can be either processed concurrently or

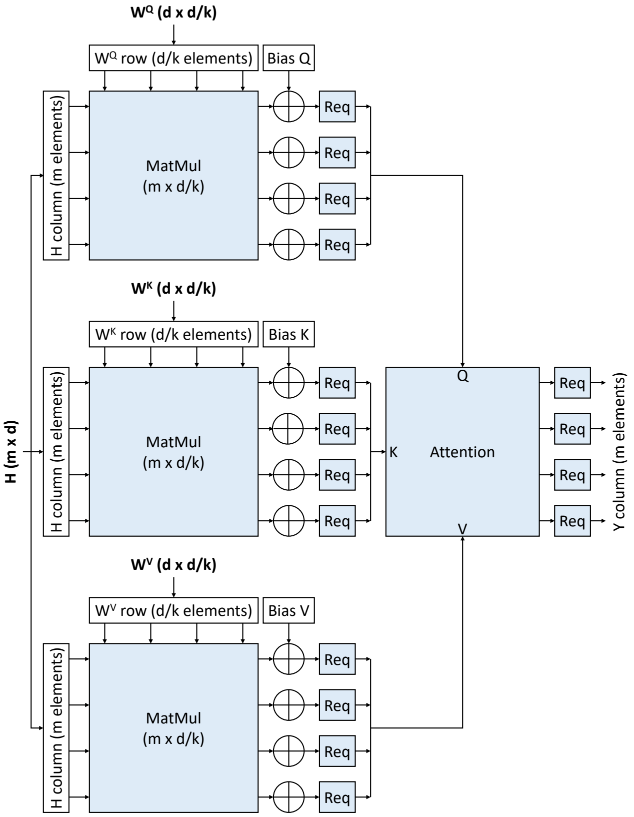


Fig. 8. Architecture details of a single head in the MHSA, composed of three MatMul blocks, performing computations on Query (Q), Key (K), and Value (V) matrices, and one Attention block.

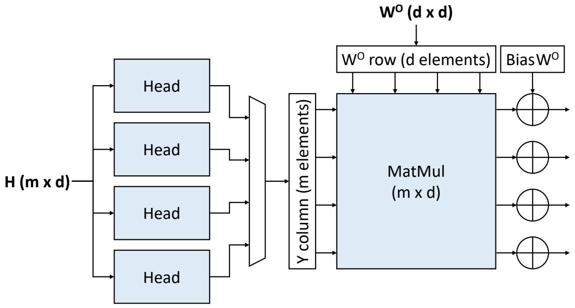


Fig. 9. Architecture details of the complete MHSA unit with multiple heads. The example in the figure contains 4 heads and another MatMul block that generates the outputs.

sequentially by reusing the head blocks. Consequently, the computations of the final MatMul can be conducted with multiple batches of data. Whenever it receives as input one head output (coming in order), it updates its accumulators.

E. Hardware Architecture of the Attention Unit

The Attention architecture is shown in Figure 10. It is composed of two plain MatMuls with a *Softmax* in between. The *Scale* simply consists of a division by the model dimension d. If the value of d is a multiple of 2, it becomes a simple shift operation. The *Requantization* is required to keep the input of the second MatMul to INT8.

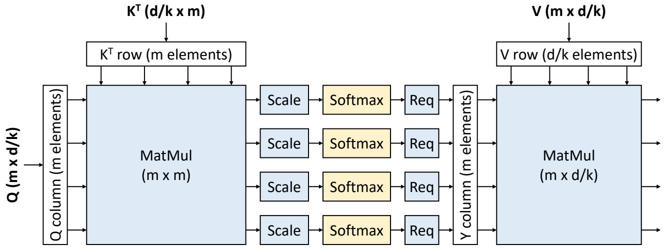


Fig. 10. Attention architecture, composed of MatMul, Scale, Softmax, and Requantization operators.

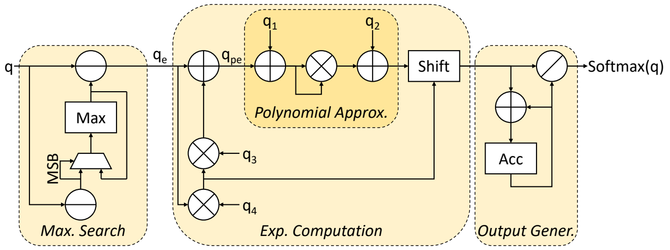


Fig. 11. Architecture of the Softmax operator. According to [11], $q_1 = \lfloor b/S_{pe} \rfloor$, $q_2 = \lfloor c/aS_{pe}^2 \rfloor$, $q_3 = \lfloor ln2/S_e \rfloor$, and $q_4 = \lfloor -1/q_3 \rfloor$, where S_e is the scaling factor of the exponential computation input (relative to q_e), S_{pe} is the scaling factor of the polynomial approximation input (relative to q_{pe}) and a, b, c are the coefficient of the second-order polynomial that approximates the function, i.e., $a(x+b)^2+c$. Therefore, $q_{1,2,3,4}$ can be computed at design time and provided as constant values to the *SwiftTron* architecture.

F. Hardware Architecture of the Softmax Unit

Since the Softmax is performed along the rows of the $Q \times K^T$ matrix, m Softmax components are instantiated and work concurrently. A single Softmax operator is shown in Figure 11. Its computation requires three phases, namely maximum search, exponential computation, and output generation.

The implementation approach of the Softmax unit (see Figure 12) aims at restricting the range of values in which the exponential function needs to be computed. Following the property described in Equation (3), as in the Softmax inputs are limited by their maximum value, subtracting the maximum value leads to dealing with non-positive real numbers, which can be decomposed [11]. Consequently, the exponential function must be computed only for the restricted range of $[-ln2,\ 0]$ and can be approximated with a second-order polynomial.

$$Softmax(\mathbf{x}_i) = \frac{exp(x_i)}{\sum_{j=1}^{k} exp(x_j)} = \frac{exp(x_i - x_{max})}{\sum_{j=1}^{k} exp(x_j - x_{max})} \quad (3)$$

It is evident that with this approximation, only simple operators are involved, like a comparator for the maximum, adders, and multipliers. The most complex operator is the divider, whose implementation consumes relatively more resources.

G. Hardware Architecture of the Feed-Forward Network Unit

This sub-layer has two linear transformations separated by an activation function. The transformations, implemented with MatMul blocks, are the biggest of the Transformer

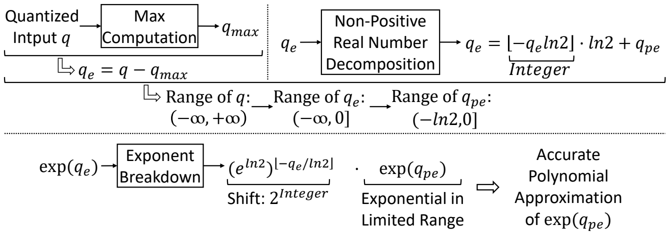

Quantized Intput q
Max Computation
q_{max}
$q_e = q - q_{max}$
q_e
Non-Positive Real Number Decomposition
$q_e = \lfloor -q_e ln2 \rfloor \cdot ln2 + q_{pe}$
Integer
Range of q: $(-\infty, +\infty)$
Range of q_e: $(-\infty, 0]$
Range of q_{pe}: $(-ln2, 0]$
exp(q_e)
Exponent Breakdown
$(e^{ln2})^{\lfloor -q_e/ln2 \rfloor}$
Shift: $2^{Integer}$
exp(q_{pe})
Exponential in Limited Range
Accurate Polynomial Approximation of exp(q_{pe})

Fig. 12. Approach used for manipulating the Softmax operator to allow a second-order polynomial approximation of the exponential function in a restricted range of values.

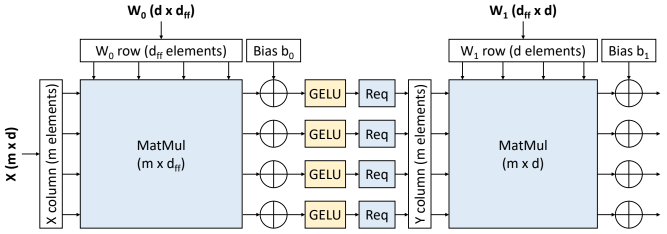


Fig. 13. Architecture of the Feed-Forward Network unit, composed of MatMul, GELU, and Requantization operators.

architecture, as d_{ff} is usually $4\times$ the dimension d. The FFN architecture is depicted in Figure 13. The activation function used in our architecture is the GELU [14], which despite being more complex, has better performances than the ReLU.

H. Hardware Architecture of the GELU Unit

The GELU operator, shown in Figure 14, contains the computation of the error function (*erf*). This nonlinear function is linearized through another second-order polynomial with limited input. With this approximation, the resulting operators are only adders and multipliers, with some sign-handling operations that complete the execution.

I. Hardware Architecture of the Residual Connection and Layer Normalization Units

Since the MHSA and FFN are residual blocks, their outputs are added to the original inputs. As we are dealing with quantized values, the two addends need to have the same scaling factor before being added together. This transformation is achieved using a Dyadic unit, already discussed in the Requantization unit (recall Equation (2)), implemented with a multiplication by a coefficient and a right-shifting. It is a combinatorial block that is replicated by the number of rows, as it receives one column at a time coming from the previous sub-blocks.

After the residual connection, the Layer Normalization (LayerNorm) is required. Its architecture is depicted in Figure 15. Similarly to the Softmax component, since the LayerNorm operation works on the row elements, d instantiations are needed. Moreover, it is composed of three phases, the mean value calculation, the standard deviation calculation, and the output generation.

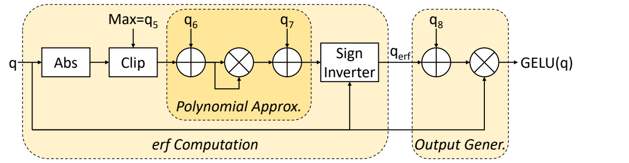


Fig. 14. Architecture of the GELU operator. According to [11], $q_5 = -b/S$, $q_6 = \lfloor b/S \rfloor$ $q_7 = \lfloor c/aS^2 \rfloor$, and $q_8 = \lfloor 1/S_{erf} \rfloor$, where S is the scaling factor of the GELU input (relative to q), S_{erf} is the scaling factor of the error function output (relative to q_{erf}) and a, b, c are the coefficient of the second-order polynomial that approximates the function, i.e., $a(x+b)^2 + c$. Therefore, $q_{5,6,7,8}$ can be computed at design time and provided as constant values to the *SwiftTron* architecture.

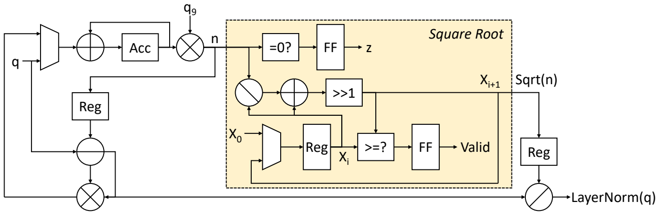


Fig. 15. Layer Normalization architecture, where the Square Root unit implements a recursive algorithm. The $Valid$ and z signals are flags for assisting the control unit with the correct timing.

The only nonlinear operation is the square root, which is implemented using an iterative algorithm as proposed in [33], also adopted in [11]. It is a recursive algorithm that needs multiple cycles to compute the output. It includes combinatorial operators and registers to store intermediate values and break the loop.

It has a constant initial value, defined as x_0. At every iteration, the partial result x_i is compared to the partial result of the next iteration x_{i+1} that is equal to $(x_i + x_i/n)/2$. Note that the division by 2 is implemented through a simple one-position right shift. The algorithm iterates until x_{i+1} is larger than or equal to x_i. When this happens, the final result is saved into the dedicated register. Since the number of cycles needed is unknown a priori, the $Valid$ and z signals are flags for assisting the control unit and generate the correct timing. In the special case when the square root input is zero, the output goes directly to zero, and no iterations are needed.

J. Hardware Architecture of the Control Unit

At each stage of the Transformers' process, the control unit generates different control signals for all the components of the *SwiftTron* architecture, according to the operations needed. Its functionality is depicted in Figure 16. For the three major operations, which are MHSA, LayerNorm, and FFN, dedicated Finite State Machines (FSMs) generate the respective control signals. A set of handshake signals (e.g., $Start$, $Done$, $Valid$) is devised to interact between different FSMs and guarantee the correct timing of the operations in all stages of the Transformers' inference.

IV. Evaluation of our SwiftTron Architecture

A. Experimental Setup

We implement the complete design of our *SwiftTron* architecture in RTL (VHDL) and evaluate it for the

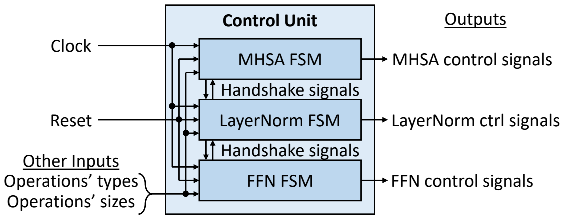


Fig. 16. Functionality of the control unit in our *SwiftTron* architecture, composed of dedicated Finite State Machines for MHSA, LayerNorm, and FFN operations.

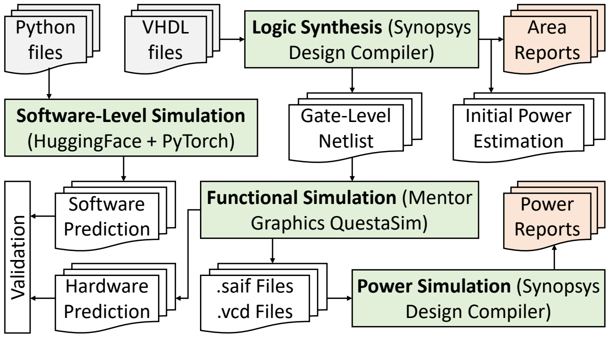

Python files
VHDL files
Logic Synthesis (Synopsys Design Compiler)
Area Reports
Software-Level Simulation (HuggingFace + PyTorch)
Gate-Level Netlist
Initial Power Estimation
Validation
Software Prediction
Functional Simulation (Mentor Graphics QuestaSim)
Power Reports
Hardware Prediction
.saif Files
.vcd Files
Power Simulation (Synopsys Design Compiler)

Fig. 17. Experimental setup and tool flow for conducting the experiments.

RoBERTa architecture [34] on the GLUE benchmark [35] and for the DeiT [36] on the ImageNet dataset [37]. We synthesize the *SwiftTron* architecture in a 65 nm CMOS technology node using the ASIC design flow with the Synopsys Design Compiler. We conduct functional and timing validation through gate-level simulations using Mentor Graphics QuestaSim. With the synthesized netlist, we obtain area, power, and performance of our design. We also run the inference on an Nvidia GeForce RTX 2080 Ti GPU for latency comparison.

For validation, we use the pre-trained Transformer models from the HuggingFace library [38], and implement them on the PyTorch framework using the quantization algorithms of I-BERT [11]. The complete flow is shown in Figure 17, where the grey boxes are the inputs, the orange boxes are the outputs, and the green boxes represent the main tools used. Note that this hardware design and validation flow are well-adopted by the hardware design community.

B. SwiftTron Synthesis Results

For evaluating the complete *SwiftTron* design, we evluate the architecture with $d = 768$, $k = 12$, $m = 256$, and $d_{ff} = 3072$, to efficiently execute the RoBERTa-base architecture [34]. Note that these values are arbitrary parameters in our design and can be tuned during design time to support the execution of different Transformer architectures. The clock period has been set to 7 ns, which corresponds to a clock frequency of ≈ 143 MHz. To comply with the timing requirements, the computations of the Softmax and LayerNorm units have been partitioned into three pipeline stages. Each component of the *SwiftTron* architecture has been tested separately and simulated to validate its outputs compared to the software-level implementation of [11]. Area and power consumption have been evaluated based on the reports obtained by the Synopsys Design Compiler tool, while the latency has been measured with a cycle-accurate simulator[3]. Table I summarizes the key results obtained from the synthesis. Table II reports the accuracy and latency of different Transformer models executed on our *SwiftTron* architecture and the speedup of our accelerator compared to an Nvidia GeForce RTX 2080 Ti GPU with CUDA 10. Our experiments run more than 3× faster than the GPU implementations.

TABLE I
SUMMARY OF SYNTHESIS RESULTS OF OUR PROPOSED *SwiftTron* ARCHITECTURE.

Clock Frequency	143 MHz	**Technology Node**	65 nm
Power Consumption	33.64 W	**Area**	273.0 mm^2

TABLE II
ACCURACY AND INFERENCE LATENCY FOR ROBERTA-BASE AND ROBERTA-LARGE MODELS ON THE STT-2 TASK OF THE GLUE BENCHMARK WITH SEQUENCE LENGTH $m = 256$ AND FOR DEIT-S ON THE IMAGENET DATASET WITH RESOLUTION 224×224 EXECUTED ON OUR *SwiftTron* ARCHITECTURE. THE LAST COLUMN REPORTS THE SPEEDUP W.R.T. THEIR EXECUTION ON THE RTX 2080 TI GPU.

Model	Accuracy	Latency	Speedup w.r.t. GPU
RoBERTa-base on STT-2	95.2%	1.83 ms	3.81×
RoBERTa-large on STT-2	96.4%	45.70 ms	3.90×
DeiT-S on ImageNet	79.11%	1.13 ms	3.58×

C. Area and Power Breakdown

The total values of the area and power consumption of the complete *SwiftTron* architecture are reported in the respective lines of Table I, while Figure 18 analyzes in detail the breakdown for each component. It is evident that the MatMul block is responsible for the majority (55%) of the area of the entire architecture. The difference between MatMul and other components becomes even larger for the power consumption. While the Softmax unit occupies 17% of the total area, its contribution to the total power is only 14%. An even more significant difference is noted for the LayerNorm unit, which occupies 25% area, but its power consumption is 6%. As expected, the GELU unit is a small component with only 3% area and 1% power consumption.

D. Comparison with Related Works

While it is known that a specialized accelerator brings immense advantages compared to executing the same task on a general-purpose hardware device, e.g., CPU or GPU, it is difficult to compare metrics like power and performance across completely different hardware platforms of the related works, which include GPUs, FPGAs, and other ASIC architectures synthesized for different technology nodes. Hence, we identify

[3] The simulator considers the worst-case scenario in terms of clock cycles computed by the square root operator of the LayerNorm unit.

TABLE III
SUMMARY OF COMPARISONS BETWEEN THE RELATED WORKS AND OUR PROPOSED *SwiftTron* ARCHITECTURE.

Work	HW Implementation	Bit-width	Complete Architecture	Nonlinear Function Computation
OPTIMUS [6]	✓ ASIC 28 nm	INT16	✗ No	✗ N/A
A^3 [7]	✓ ASIC 40 nm	✓ INT8	✗ No	✓ Integers (approximated)
FTRANS [31]	✓ Xilinx FPGA	INT16	✓ Yes	✗ Integers using FFT
Lu et al. [24]	✓ Xilinx FPGA	✓ INT8	✗ No	✓ Integers (approximated)
EFA-Trans [29]	✓ Xilinx FPGA	✓ INT8	✓ Yes	✗ LUT
FQ-BERT [30]	✓ Xilinx FPGA	✓ INT8	✓ Yes	✗ LUT
Lin et al. [8]	✗ TITAN V GPU	✓ INT8	✓ Yes	✗ FP32
I-BERT [11]	✗ Tesla T4 GPU	✓ INT8	✓ Yes	✓ Integers (approximated)
I-ViT [21]	✗ RTX 2080 Ti GPU	✓ INT8	✓ Yes	✓ Integers (approximated)
Transformer Engine [9]	✓ ASIC 4 nm (inside H100 GPU)	✗ FP8	✓ Yes	✗ FP16 / FP32
SwiftTron (ours)	**✓ ASIC 65 nm**	**✓ INT8**	**✓ Yes**	**✓ Integers (approximated)**

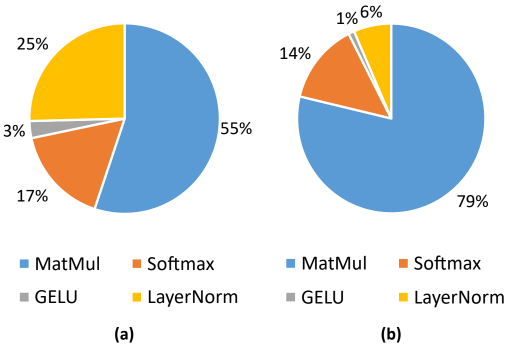


Fig. 18. **(a)** Area and **(b)** power breakdown of our *SwiftTron* architecture.

key features of a hardware architecture for Transformers that push its efficiency to the upper boundary. Not only the hardware device on which it is implemented is important, but also the bit-width plays a key role in determining its energy efficiency. This is because even simple operators like adders and multipliers are relatively lightweight when implemented using integer arithmetic and low bit-width, like for INT8. On the other hand, their floating-point implementation incurs a significant complexity overhead.

Table III summarizes the comparison between our *SwiftTron* architecture and the related works, considering these important features for a hardware architecture for Transformers. From the table, it is clear that *our work complies with all the requirements, while all the related works have at least one missing feature*. Some works [8] [11] [21] implement their design on GPUs, other works [6] [7] [24] accelerate only part of a complete Transformer, and other works do not use efficient computations for their nonlinear functions. The design proposed in [31] uses integer computations, but its complexity is high due to the presence of FFT transforms. The architectures presented in [29] [30] use LUTs for computing some nonlinear operations, while the works in [8] [9] performs the nonlinear computations using FP16 or FP32 arithmetics.

V. CONCLUSION

In this paper, we present *SwiftTron*, a specialized accelerator for Transformers that executes all the operations, including the nonlinear operations, in integer arithmetic. The correct computations between integers are achieved through a specialized quantization scheme that accounts for diverse scaling factors. Dedicated designs implement approximated versions of the nonlinear units, like Softmax, GELU, and Layer Normalization. The *SwiftTron* architecture synthesized using the ASIC design flow shows efficient area, power, and performance while complying with all the desired features for a Transformer accelerator. Our design and thorough analyses pave the way for future developments of efficient Transformer architectures.

ACKNOWLEDGMENT

This work has been supported in part by the Doctoral College Resilient Embedded Systems, which is run jointly by the TU Wien's Faculty of Informatics and the UAS Technikum Wien. This work was also supported in parts by the NYUAD's Research Enhancement Fund (REF) Award on "eDLAuto: An Automated Framework for Energy-Efficient Embedded Deep Learning in Autonomous Systems", and by the NYUAD Center for Artificial Intelligence and Robotics (CAIR), funded by Tamkeen under the NYUAD Research Institute Award CG010.

REFERENCES

[1] M. Shafique, T. Theocharides, C. Bouganis, M. A. Hanif, F. Khalid, R. Hafiz, and S. Rehman, "An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the iot era," in *2018 Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pp. 827–832, IEEE, 2018.

[2] A. Marchisio, M. A. Hanif, F. Khalid, G. Plastiras, C. Kyrkou, T. Theocharides, and M. Shafique, "Deep learning for edge computing: Current trends, cross-layer optimizations, and open research challenges," in *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 553–559, IEEE, 2019.

[3] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, vol. 12, no. 7, p. 113, 2020.

[4] M. Shafique, A. Marchisio, R. V. W. Putra, and M. A. Hanif, "Towards energy-efficient and secure edge AI: A cross-layer framework ICCAD special session paper," in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*, pp. 1–9, IEEE, 2021.

[5] S. Dave, A. Marchisio, M. A. Hanif, A. Guesmi, A. Shrivastava, I. Alouani, and M. Shafique, "Special session: Towards an agile design methodology for efficient, reliable, and secure ML systems," in *40th IEEE VLSI Test Symposium, VTS 2022, San Diego, CA, USA, April 25-27, 2022*, pp. 1–14, IEEE, 2022.

[6] J. Park, H. Yoon, D. Ahn, J. Choi, and J. Kim, "OPTIMUS: optimized matrix multiplication structure for transformer neural network accelerator," in *Proceedings of Machine Learning and Systems 2020 (MLSys)*, 2020.

[7] T. J. Ham, S. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J. Park, S. Lee, K. Park, J. W. Lee, and D. Jeong, "A^3: Accelerating attention mechanisms in neural networks with approximation," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 328–341, 2020.

[8] Y. Lin, Y. Li, T. Liu, T. Xiao, T. Liu, and J. Zhu, "Towards fully 8-bit integer inference for the transformer model," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 3759–3765, 2020.

[9] "Nvidia h100 tensor core gpu architecture whitepaper," https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper.

[10] G. Marcus, P. Hinojosa, A. Avila, and J. Nolazco-FIores, "A fully synthesizable single-precision, floating-point adder/substractor and multiplier in vhdl for general and educational use," in *Proceedings of the Fifth IEEE International Caracas Conference on Devices, Circuits and Systems, 2004.*, vol. 1, pp. 319–323, 2004.

[11] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, "I-BERT: integer-only BERT quantization," in *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pp. 5506–5518, 2021.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30 (NeurIPS)*, pp. 5998–6008, 2017.

[13] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 807–814, 2010.

[14] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with gaussian error linear units," *CoRR*, vol. abs/1606.08415, 2016.

[15] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and L. Kaiser, "Universal transformers," in *7th International Conference on Learning Representations (ICLR)*, 2019.

[16] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 4171–4186, 2019.

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2020.

[18] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *CoRR*, vol. abs/2006.04768, 2020.

[19] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 9992–10002, IEEE, 2021.

[20] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *CoRR*, vol. abs/2101.03961, 2021.

[21] Z. Li and Q. Gu, "I-vit: Integer-only quantization for efficient vision transformer inference," *CoRR*, vol. abs/2207.01405, 2022.

[22] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations (ICLR)*, 2021.

[23] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead," *IEEE Access*, vol. 8, pp. 225134–225180, 2020.

[24] S. Lu, M. Wang, S. Liang, J. Lin, and Z. Wang, "Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer," in *33rd IEEE International System-on-Chip Conference (SoCC)*, pp. 84–89, IEEE, 2020.

[25] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "HAT: hardware-aware transformers for efficient natural language processing," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 7675–7688, 2020.

[26] H. Wang *et al.*, *Efficient algorithms and hardware for natural language processing*. PhD thesis, Massachusetts Institute of Technology, 2020.

[27] A. Bhandare, V. Sripathi, D. Karkada, V. Menon, S. Choi, K. Datta, and V. Saletore, "Efficient 8-bit quantization of transformer neural machine language translation model," *CoRR*, vol. abs/1906.00532, 2019.

[28] G. Prato, E. Charlaix, and M. Rezagholizadeh, "Fully quantized transformer for machine translation," in *Findings of the Association for Computational Linguistics (EMNLP)*, pp. 1–14, 2020.

[29] X. Yang and T. Su, "Efa-trans: An efficient and flexible acceleration architecture for transformers," *Electronics*, vol. 11, no. 21, 2022.

[30] Z. Liu, G. Li, and J. Cheng, "Hardware acceleration of fully quantized BERT for efficient natural language processing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 513–516, IEEE, 2021.

[31] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, and C. Ding, "FTRANS: energy-efficient acceleration of transformers using FPGA," in *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 175–180, ACM, 2020.

[32] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. W. Mahoney, and K. Keutzer, "HAWQ-V3: dyadic neural network quantization," in *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pp. 11875–11886, 2021.

[33] C. P. Richard Crandall, *Prime Numbers: A Computational Perspective*, vol. 182. Springer New York, NY, 2006.

[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019.

[35] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *7th International Conference on Learning Representations (ICLR)*, 2019.

[36] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, "Training data-efficient image transformers & distillation through attention," in *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pp. 10347–10357, 2021.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25 (NeurIPS)*, pp. 1106–1114, 2012.

[38] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020.