Microprocessors and Microsystems 120 (2026) 105223

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

A runtime-adaptive transformer neural network accelerator on FPGAs☆

Ehsan Kabir [a],*, Jason D. Bakos [b], David Andrews [a], Miaoqing Huang [a]

[a] Department of Electrical Engineering and Computer Science, University of Arkansas, Fayetteville, AR, USA
[b] Department of Computer Science and Engineering, University of South Carolina, Columbia, SC, USA

ARTICLE INFO

Keywords:
FPGA
Transformer
Attention
Neural networks
Encoder
High-level synthesis
Natural language processing
Hardware accelerators

ABSTRACT

Transformer neural networks (TNN) excel in natural language processing (NLP), machine translation, and computer vision (CV) without relying on recurrent or convolutional layers. However, they have high computational and memory demands, particularly on resource constrained devices like FPGAs. Moreover, transformer models vary in processing time across applications, requiring custom models with specific parameters. Designing custom accelerators for each model is complex and time-intensive. Some custom accelerators exist with no runtime adaptability, and they often rely on sparse matrices to reduce latency. However, hardware designs become more challenging due to the need for application-specific sparsity patterns. This paper introduces ADAPTOR, a runtime-adaptive accelerator for dense matrix computations in transformer encoders and decoders on FPGAs. ADAPTOR enhances the utilization of processing elements and on-chip memory, enhancing parallelism and reducing latency. It incorporates efficient matrix tiling to distribute resources across FPGA platforms and is fully quantized for computational efficiency and portability. Evaluations on Xilinx Alveo U55C data center cards and embedded platforms like VC707 and ZCU102 show that our design is 1.2× and 2.87× more power efficient than the NVIDIA K80 GPU and the i7-8700K CPU respectively. Additionally, it achieves a speedup of 1.7 to 2.25× compared to some state-of-the-art FPGA-based accelerators.

1. Introduction

Transformer neural networks (TNN) have shown great performance in natural language processing (NLP) [1], machine translation [2], computer vision [3], and other fields in recent years. While recurrent neural network (RNN) [4] and long short-term memory (LSTM) [5] models run sequential computation tasks during both training and inference, transformer facilitates high levels of computation parallelism throughout both processes using an attention mechanism. Thus, TNN is becoming a potential alternative to CNN, RNN, and LSTM [6,7]. There are many transformer models, such as full transformers containing both encoder and decoder [8], BERT [9,10], ALBERT [11], structBERT [12], and others. These models contain different numbers of encoder and decoder stack [8] for different applications. A single encoder will often require a latency on the order of 100s of μS [13]. Around 38% to 64% of this time is spent in the multihead attention (MHA) mechanism depending on the number of tokens in the input sequence [14,15], and the rest of the time is spent on feed forward network (FFN). Unfortunately, general-purpose platforms like GPUs and CPUs often suffer from low computational efficiency, underutilized memory bandwidth, and substantial compilation overheads for MHA layers [16]. MHA and FFN also occupy most of the on chip storage units [17–19]. Therefore, it is essential to prioritize efficient hardware deployment on resource-constrained devices. FPGAs have gained widespread use for accelerating DNNs due to their high level of parallelism, high energy efficiency, and low latency [20,21]. Recently, some works have successfully built FPGA based custom hardware accelerators for transformers [13,18,22]. Application-specific integrated circuits (ASIC)-based accelerators also exist [23].

Lu et al. [22] accelerated the attention mechanism and feedforward network separately, but did not implement the full transformer encoder. Ye et al. [24] focused on accelerating only the attention mechanism using a reconfigurable systolic array for the transformer. Similarly, Zhang et al. [25] concentrated on accelerating the attention layer through hardware–software co-design. In contrast, ***ADAPTOR*** is developed to support the entire transformer neural network (TNN). Some other works accelerate the full transformer networks but their logic circuits go through the time-consuming synthesis steps for different models or they perform poorly on the same model with different configurations [26]. These approaches lack the generality to support

☆ This material is based upon work supported by the National Science Foundation, United States under Grant No. 1956071.
* Corresponding author.
E-mail addresses: kabir40ehsan@gmail.com, ekabir@tamut.edu (E. Kabir), jbakos@cse.sc.edu (J.D. Bakos), dandrews@uark.edu (D. Andrews), mqhuang@uark.edu (M. Huang).

https://doi.org/10.1016/j.micpro.2025.105223
Received 11 July 2025; Received in revised form 30 October 2025; Accepted 4 November 2025
Available online 17 November 2025

diverse variants, whereas ***ADAPTOR*** eliminates the need for repeated synthesis across models.

As transformer variants continue to evolve with differing parameters, designing a generic and efficient accelerator that can be customized to the structural characteristics of these variants becomes increasingly valuable. Thus, a versatile accelerator is needed to efficiently handle dense matrix computations across various TNN applications, and ***ADAPTOR*** is designed to fulfill this role. Digital signal processing (DSP) resources are capable of high-speed computation at higher frequencies. Proper utilization of them depends on the implementation method. For example, most accelerators [13,27–29] used high-level synthesis (HLS) tools, while some used hardware description language (HDL) [30–32] for design. While HLS requires less implementation time compared to HDL, writing efficient HLS code to use parallel DSPs for optimal performance is challenging [33]. To address this, ***ADAPTOR*** employs optimized HLS coding techniques. Additional challenges include storing the vast number of TNN parameters in the on-chip memories of FPGAs, which typically have a size of 5MB for low-end devices such as the ZCU104 and 35MB for high-end devices such as the Alveo U200 [34] and executing the extensive number of multiplication and accumulation (MAC) operations required by TNNs on the DSPs, with Ultrascale+ FPGAs offering approximately 9024 DSPs. Therefore, input matrices must be partitioned into tiles. However, developing an optimal partitioning scheme that aligns well with the architecture presents a significant challenge, one that has been carefully addressed in the design of ***ADAPTOR***. The data access and computation patterns differ across various blocks within the transformer, which also prevents acceleration. To overcome this, ***ADAPTOR*** assigns dedicated hardware modules to each block, enabling more effective design and optimization. The full source code[1] to reproduce the presented results or improve the design.

In summary, this work makes the following contributions:

- A novel accelerator architecture for a complete transformer that maximizes DSP and LUT utilization to enhance parallel processing and achieve low latency, supported by an analytical model for pre-execution estimates of resource use and latency.
- An efficient tiling strategy for weight matrices in both the multi-head attention layer and the feedforward neural network layer, enabling the deployment of the accelerator to any FPGA platform for most TNN models.
- A modular design approach implemented using parameterized HLS codes to accommodate varying computation and data access patterns, as well as to allow design-time modification of different TNN components.
- A runtime adaptive feature allows software-driven parameter adjustments to run different models without hardware re-synthesis.

2. Related work

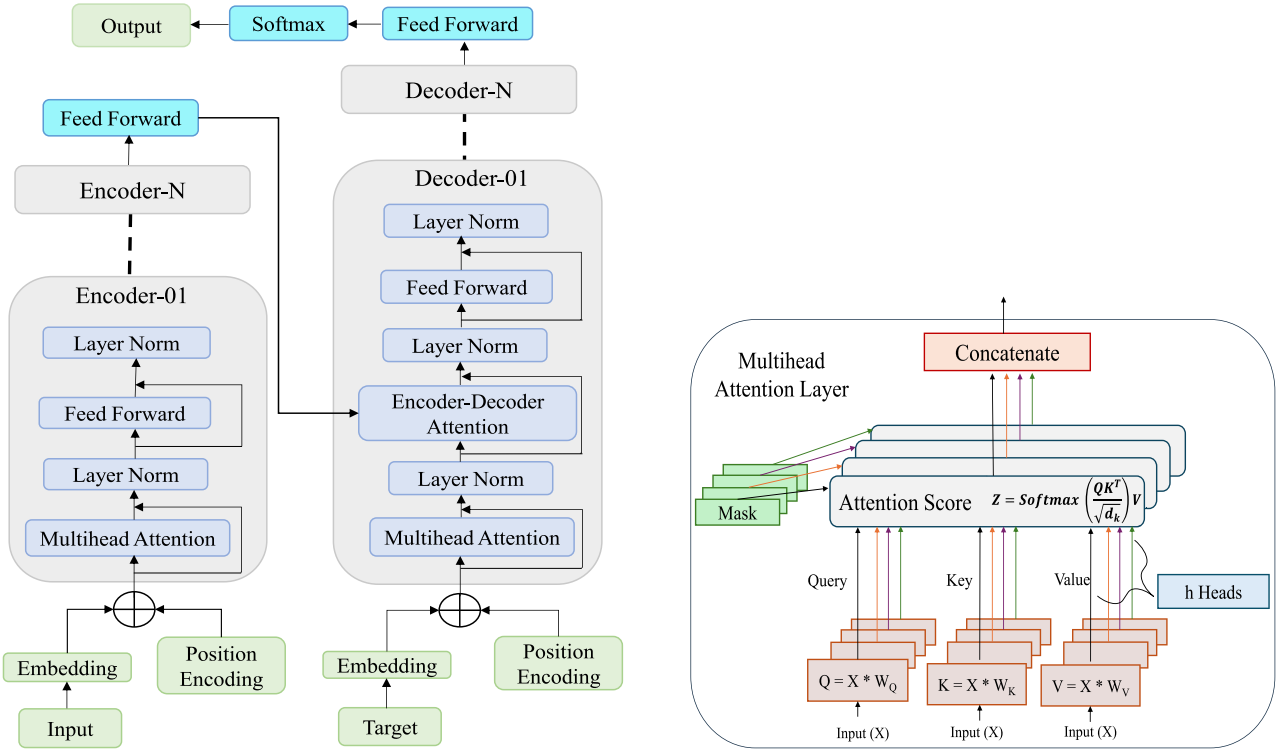
Various custom and partially adaptive FPGA accelerators have been developed for TNNs. Peng et al. [13,27] introduced a coherent sequence length–adaptive algorithm-hardware co-design for Transformer acceleration and explored column-balanced block-wise pruning. Qi et al. [19,34] proposed an acceleration framework combining balanced model compression at the algorithm level with hardware-level FPGA optimization. Chen et al. [35] developed an analytical model for evaluating spatial TNN accelerators, considering FPGA compute and memory resources, identifying optimal parallelization and buffering strategies, and providing reusable HLS kernels. Similarly, we developed an analytical model to estimate the latency and resource utilization of ***ADAPTOR*** and designed it modularly, with each module implemented as an HLS function for easy optimization and reuse. Qin et al. [36] designed a TNN accelerator with separate attention and linear kernels for long input sequences, applying tiling only to the attention layer, whereas our design applies unique tiling strategies to both attention and linear layers. Both architectures incorporate analytical models. The energy-efficient FTRANS framework [18] employed an improved block-circulant matrix method for algorithm-level sparsity, alongside a dedicated accelerator designed for this approach. Most of these architectures target specific TNNs and sparsity patterns, lacking runtime flexibility to reconfigure the computing structure for different applications. In contrast, ***ADAPTOR*** can be programmed from software for any dense TNN model. FlexRun [37] identified key NLP model components, implemented them on a state-of-the-art FPGA accelerator, performed design space exploration to determine the optimal architecture for a given NLP model, and enabled automatic reconfiguration based on the results. FET-OPU [38] presented an overlay architecture for general TNN acceleration featuring a DSP-packed Matrix Multiplication Unit (MMU) with a FIFO-based data caching mechanism. FlightLLM [16] introduced a configurable sparse DSP chain for handling diverse sparsity patterns efficiently, an always-on-chip decode scheme for improved memory bandwidth with mixed-precision support, and a length-adaptive compilation method to minimize instruction storage overhead for large language models. TRAC [39] focused on dedicated hardware generation, integrating code generation into the compilation process to create parameterized and synthesized modules for specific Transformer configurations-unlike fixed, though parameterizable, overlays. EFA-Trans [31] supports both dense and sparse computation patterns but requires hardware resynthesis to switch between them. Moreover, none of these works examined optimal tile sizes or DSP utilization for maximum parallelism as done in ***ADAPTOR***.

3. Background

3.1. Transformer architecture

There are several building blocks in transformers as shown in Fig. 1(a). An input sequence of tokens is converted into embeddings. The positional encoder enables the model to consider the order of tokens in a sequence by adding positional information to the embeddings. It generates vectors that give context according to the word's position in a sentence. Then the vectors are linearly transformed into three tensors: Q (queries), K (keys), and V (values) by multiplying the embedding matrix with three weight matrices. The encoder block handles these tensors, transforming them into a higher-level representation that encapsulates crucial information. This process ensures the proper capture of features and contextual relationships within the input sequence. The encoder architecture comprises two main sub-layers: (1) the self-attention mechanism, and (2) the position-wise feed-forward network. The self-attention mechanism enables the model to assess different segments of an input sequence simultaneously. It captures long-range relationships by measuring attention scores and utilizing multi-head projections for various input representations. Thus, it can learn complex patterns, dependencies, and relationships effectively. The position-wise feed-forward network (FFN), which is equivalent to a multilayer perceptron (MLP), applies linear transformations to every position independently in the input sequence. In this network, two linear transformations are executed. They mainly contain matrix–vector multiplication. The first linear transformation has activation functions such as the Rectified Linear Unit (ReLU) or Gaussian Error Linear Unit (GeLU) but the second one does not have these. Furthermore, each sub-layer includes a residual connection combined with layer normalization (LN). This reduces the vanishing gradient problem during training. Residual addition and LN layers are inserted after each MHA and FFN. It mainly includes the addition of matrix elements and nonlinear functions. The decoder block illustrated in Fig. 1(a) is responsible for generating the output sequence based on the encoded representations supplied by the encoder. Like the encoder, the decoder also consists of a

[1] https://github.com/Kabir-Ehsan/Transformer_on_FPGA.



(a) Complete architecture.

(b) Multihead attention layer.

Fig. 1. Transformer neural network.

stack of N identical layers. Each layer within the decoder contains three sub-layers. They are: (1) the Masked Attention Mechanism, resembling the encoder's self-attention, and it includes a masking feature that restricts the output's dependency on known preceding outputs; and (2) an attention layer that directs its focus to the encoder's output, enabling the decoder to emphasize relevant sections of the input sequence for each output element and (3) a position-wise feed-forward network.

The self-attention mechanism in transformers allows each position in the sequence to attend to all other positions, enabling the model to consider global context easily. Each attention head is composed of three linear layers and a scaled dot-product attention function. The parameter h – or number of heads – is equal to 8 in the Transformer base model or 16 in the Transformer big model. As illustrated in Fig. 1(b), the scaled dot product attention in each head is a crucial part of the multihead attention layer. The attention weights are computed by performing the dot product of the query and key vectors and subsequently scaling it down by the square root of the dimension of the key vectors. This scaling is essential to prevent the dot products from becoming excessively large, which contributes to the stabilization of gradients during the training process. Subsequently, the scaled dot products undergo the softmax function, resulting in the computation of attention weights. These weights are then used to perform a weighted sum of the value vectors. The ultimate output is the projection of the concatenated sequences from all heads.

The output of MHA can be represented as Eqs. (1) & (2). The input sequence X is linearly mapped into Q_i, K_i, V_i matrices using weights and biases. The parameter $d_k = d_{model}/h$ is the dimension of Q_i and K_i. d_{model} is a hyperparameter called embedding dimension, and h is the number of heads.

$$Attention(Q_i, K_i, V_i) = softmax\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i \tag{1}$$

$$Q_i = X \times W_q + B_q, \quad K_i = X \times \quad W_k + B_k, \quad V_i = X \times W_v + B_v \tag{2}$$

$$FFN(X) = Layer\_Norm(X + ReLU(X \times W_1 + b_1) \times W_2 + b_2) \tag{3}$$

$$Layer\_Norm(X) = \gamma\left(\frac{X - \mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta \tag{4}$$

The FFN comprises a LN operation, residual addition, a ReLU activation, and two linear sublayers, as described in Eq. (3), where W_1, W_2 are weights and b_1, b_2 are biases. The operations for layer normalization, softmax, GELU and RELU activation functions are described in Eqs. (4), (5), (6), and (7) respectively, where X is the input vector (for a particular position in the sequence), μ is the mean of X, σ^2 is the variance of X, γ and β are learnable parameters, and ϵ is a small constant.

$$softmax(X_j) = \frac{e^{X_j}}{\sum_{i=1} e^{X_i}} \tag{5}$$

$$GELU(x) = xP(X \leq x) = x \times \frac{1}{2}[1 + erf(X/\sqrt{(2)})] \tag{6}$$

$$RELU(X) = \begin{cases} 0, & X < 0 \\ X, & X \geq 0 \end{cases} \tag{7}$$

3.2. High Level Synthesis design

High-Level Synthesis (HLS) allows designers to describe circuit functionality at a higher level of abstraction than that of hardware description language. HLS tools translate high-level code, typically written in languages like C, C++, or OpenCL, into Register-Transfer Level (RTL) code suitable for FPGA implementation. This approach offers several advantages, including faster development cycles and simplified design modifications, as designers can use familiar programming languages to describe the hardware. Moreover, HLS enables efficient design space exploration, allowing different architectures to be evaluated without extensive hardware design expertise, leading to the rapid

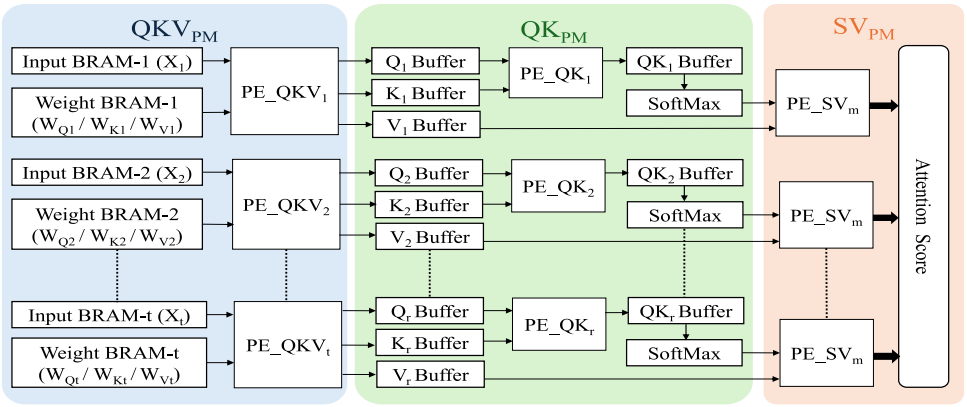


Fig. 2. Attention module of ***ADAPTOR***.

creation of optimized accelerators optimized for power, performance, and area [40]. However, HLS does come with challenges, such as ensuring that the generated RTL meets the specified constraints. The success of the synthesized hardware is largely dependent on the robustness of the HLS tools and the expertise of the designer.

4. ADAPTOR's architecture

The core of the ***ADAPTOR*** is designed in C language on Vitis high-level synthesis (HLS) 2022.2.1 tool. C simulation confirms the algorithm's correctness, while C/RTL co-simulation validates the functionality of the synthesized hardware. This section describes the HLS design technique that generates an optimized architecture utilizing most of the LUTs and DSPs in the processing modules, ensuring high parallelism of computation. There are loading units, computing modules, and activation function units in the overall architecture, which are described below. Figs. 2 and 3 represent two main computing modules of ***ADAPTOR***.

4.1. Attention module

The overall architecture designed to accelerate the attention mechanism is illustrated in Fig. 2. It consists of three principal processing modules (PMs), denoted as QKV_{PM}, QK_{PM}, and SV_{PM}, according to the specific operations they perform. Each of these modules begins operation only after the previous module has completed its computations. This strict sequential execution ensures that all data dependencies are respected and simplifies control logic. The number of module instances corresponds to the number of attention heads (h). Within each module, computation is carried out by an array of processing elements (PEs), where each PE incorporates a DSP48 unit responsible for multiplication and accumulation (MAC) operations. The organization of the PE arrays varies across modules, as their computational demands and data access patterns differ. To accommodate these differences, the modules are implemented as separate functions in high-level synthesis (HLS), thereby enabling targeted optimization of the corresponding register-transfer level (RTL) components. Parallel data access is supported by distributing input activations and weights across multiple BRAMs and LUTRAMs.

Each PE operates independently, equipped with its own local memory, control logic, and computational resources. The weight matrices associated with the generation of queries (W_q), keys (W_k), and values (W_v) are stored as two-dimensional arrays of dimension $\left(\frac{d_{model}}{h} \times TS_{MHA}\right)$, where TS_{MHA} denotes the tile size of the attention module. This tiling strategy partitions the larger weight matrices into sub-matrices, thereby facilitating efficient parallelization. The interplay between the number of heads, tiling parameters, and the HLS array partitioning directives determines how these arrays are mapped onto multiple two-port BRAMs. Since BRAM ports are limited, careful partitioning and scheduling of data transfers ensure that all operands required concurrently by the DSP units are accessible without contention. The intermediate Q, K, and V matrices, each of size $\left(SL \times \frac{d_{model}}{h}\right)$ where SL denotes the sequence length, are buffered locally to support subsequent stages of computation.

4.1.1. QKV_{PM} module

The QKV_{PM} module is responsible for generating the query, key, and value matrices. It incorporates dedicated BRAMs for the weights (W_Q, W_K, W_V) and for the input activations (X_i), which provide parallel data access to the DSP units within the processing element (PE) array. To accommodate on-chip memory constraints, the weight and input arrays are divided into subarrays using a tiling strategy, ensuring that the data can be efficiently mapped onto BRAMs or LUTRAMs. The number of times the QKV_{PM} module is invoked is determined by the tiling factor, resulting in a total of $\frac{d_{model}}{TS_{MHA}}$ iterations. At each iteration, the buffers for W_Q, W_K, W_V, and X_i are populated with distinct tiles of data, after which computation is initiated within the PEs.

During these operations, the corresponding bias terms for the Q, K, and V matrices are fetched from off-chip memory into registers in parallel with the primary computations of the QKV_{PM} module. These biases are subsequently integrated into the generated matrices, thereby completing the linear transformations. The computational flow of this module is summarized in Algorithm 9 of Appendix, where pipelining of the outer loop facilitates full unrolling of the innermost loop. This design yields an array of $\frac{dmodel}{TS_{MHA}}$ PEs, thereby maximizing throughput while maintaining an efficient mapping of resources.

4.1.2. QK_{PM} module

The QK_{PM} module carries out the matrix–matrix multiplication between the Q and K matrices. Since these matrices are relatively small in dimension, tiling is not required. The computational flow is summarized in Algorithm 11 of Appendix, where full unrolling of the innermost loop produces $\frac{dmodel}{h}$ processing elements (PEs). Within this module, the Q and K matrices are buffered to enable parallel access by the DSP units. In addition to the multiplication operations, the division specified in Eq. (1) is also performed within this module using LUT resources. To avoid excessive LUT utilization, the degree of parallelism for this operation is deliberately constrained. The output of this module is the intermediate attention weight matrix S, which is stored in either BRAMs or registers depending on availability and access requirements. These weights are subsequently passed to the non-linear softmax function, implemented in HLS using LUTs and flip-flops, to complete the attention score computation.

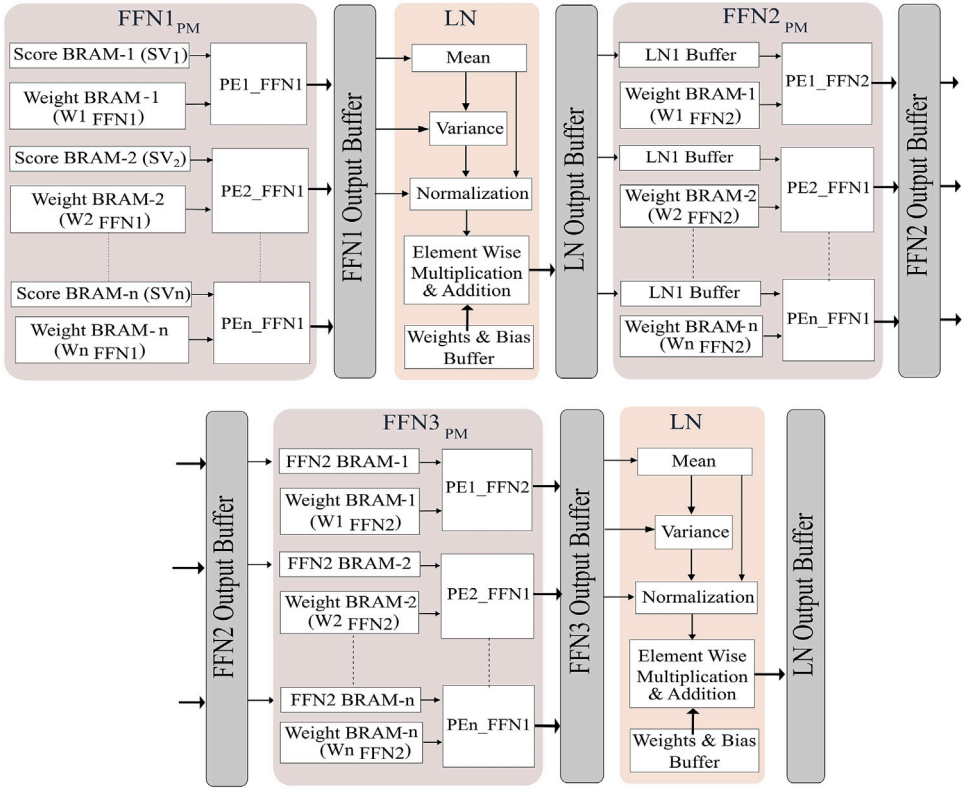


Fig. 3. Feedforward network module of ***ADAPTOR***.

4.1.3. SV_{PM} module

The normalized attention weight matrix (S), obtained from the softmax operation, is supplied to the SV_{PM} module, where it is combined with the value (V) matrix through matrix–matrix multiplication. As described in Algorithm 12 of Appendix, the innermost loop is fully unrolled, enabling SL processing elements to operate in parallel. The resulting output, referred to as the attention score, represents a weighted aggregation of the value vectors and constitutes the final contribution of the attention mechanism to the subsequent layers.

4.2. Feedforward network module

The architecture developed to accelerate the feedforward network (FFN) module is depicted in Fig. 3. Three distinct RTL modules $FFN1_{PM}$, $FFN2_{PM}$, and $FFN3_{PM}$ are implemented to support variations of the FFN across different architectural configurations. They are executed sequentially to respect the inherent data dependencies. Each module begins processing only after the preceding module has fully completed its computation. In high-level synthesis (HLS), these modules are described as separate functions, each defined by input and output arrays of different dimensions, which are subsequently mapped onto BRAMs or LUTRAMs during synthesis. Since the computational workload differs across the modules, each function is optimized independently, resulting in varying numbers of processing elements depending on the unrolling factor applied to the innermost loop. The weights of the FFN are stored in a two-dimensional array (W_o) of dimensions $\left(\frac{d_{model}}{TS_{FFN}} \times \frac{4{\times}d_{model}}{TS_{FFN}}\right)$, where TS_{FFN} denotes the tile size in FFN. This tiling strategy partitions the weight matrices into smaller blocks, facilitating parallel access and efficient memory utilization. Among the three RTL modules, both $FFN1_{PM}$ and $FFN3_{PM}$ are followed by layer normalization (LN), ensuring stabilized activations before passing results to subsequent stages.

4.2.1. $FFN1_{PM}$ module

The $FFN1_{PM}$ module performs the initial linear transformation on the attention scores, serving as the first stage of the feedforward network. To accommodate on-chip memory constraints, the arrays used by the processing elements (PEs) are tiled along both dimensions. Consequently, the module is invoked $TS_{FFN} \times TS_{FFN}$ times to complete the transformation. As outlined in Algorithm 14 of Appendix, pipelining of the second loop enables full unrolling of the innermost loop (line 7), producing TS_{FFN} PEs in total. This corresponds to $\frac{d_{model}}{\text{No. of Tiles\_}FFN}$ parallel computational units.

4.2.2. $FFN2_{PM}$ module

Building upon the normalized outputs of $FFN1_{PM}$, the $FFN2_{PM}$ module performs the second linear transformation, expanding the intermediate representation. Similar to $FFN1_{PM}$, arrays are tiled along both dimensions, though this module requires $4 \times TS_{FFN} \times TS_{FFN}$ accesses due to the increased dimensionality of the operation. The computational flow is summarized in Algorithm 15 of Appendix, where pipelining again enables full unrolling of the innermost loop (line 7). This results in TS_{FFN} PEs, corresponding to $\frac{d_{model}}{\text{No. of Tiles\_}FFN}$ units of parallelism, consistent with the structural design of the first module.

4.2.3. $FFN3_{PM}$ module

The $FFN3_{PM}$ module applies the final linear transformation to the normalized outputs of $FFN2_{PM}$, projecting them back to the original model dimension. As with the preceding modules, arrays are tiled along both dimensions, requiring $4 \times TS_{FFN} \times TS_{FFN}$ iterations to complete the computation. Algorithm 10 of Appendix describes the workflow, where pipelining and full loop unrolling (line 7) yield $4 \times TS_{FFN}$ PEs. This corresponds to $\frac{4{\times}d_{model}}{\text{No. of Tiles\_}FFN}$, reflecting the higher dimensionality of this stage.

4.3. Load weights unit

Three dedicated *Load_Weights* units are employed to manage the transfer of parameters from external memory to on-chip buffers. The first unit supplies weights to the weight memories of the attention heads (Fig. 2), while the second serves the feedforward network (Fig. 3). A third unit is responsible for loading the weights associated with the layer normalization modules. This separation ensures that weight data can be delivered efficiently to each functional block in accordance with its computational demands. For the attention module, weights are represented in HLS as two-dimensional arrays of dimension $\left(\frac{d_{model}}{h} \times TS_{MHA}\right)$, where TS_{MHA} denotes the tile size applied to partition the larger matrices into sub-matrices. After synthesis, these arrays are mapped onto dual-port BRAMs or LUTRAMs, and are populated iteratively with tile-specific data transferred from external memory at each iteration. In the feedforward network, the weight arrays are defined with dimensions $(TS_{FFN} \times 4 \times TS_{FFN})$. Here, TS_{FFN} corresponds to the tiling parameter, defined as $\frac{\text{Embedding Dimension}}{\text{No. of Tiles\_}FFN}$, while $4 \times TS_{FFN}$ equals $\frac{\text{Hidden Dimension}}{\text{No. of Tiles\_}FFN}$. These weights are therefore partitioned along both row and column dimensions, requiring iterative loading across tiles. As in the attention module, they are synthesized as dual-port BRAMs or LUTRAMs. The weights for layer normalization are comparatively simple, represented as one-dimensional arrays of length d_{model}. As no tiling is required in this case, the entire weight set is transferred in a single step and subsequently synthesized into dual-port BRAMs or LUTRAMs. The complete procedure for weight loading is outlined in Algorithm 1 of Appendix.

4.4. Load inputs unit

Input data are transferred from external memory into dedicated input BRAMs, which are implemented in HLS as dual-port, two-dimensional arrays of size $(SL \times d_{model})$, where SL denotes the sequence length. These BRAMs are reused across encoder and decoder layers, allowing each layer to access the outputs of the previous layer as inputs for subsequent computations. Three distinct *Load_inputs* units manage the data movement to accommodate differences in computation, tiling, and array dimensions across modules. The first unit populates the intermediate input BRAMs of each attention head (Fig. 2) using Algorithm 2 of Appendix. These BRAMs are represented as two-dimensional arrays of size $(SL \times TS_{MHA})$, where tiling is applied along the column dimension. Consequently, data are loaded iteratively $\frac{d_{model}}{TS_{MHA}}$ times to supply all columns to the processing elements.

The second unit transfers data to the Score BRAMs of the $FFN1_{PM}$ module (Fig. 3), defined as two-dimensional arrays of size $(SL \times TS_{FFN})$, using Algorithm 3. The third unit supplies data to the LN1 buffers of the $FFN2_{PM}$ module (Fig. 3), represented as arrays of size $(SL \times 4 \times TS_{FFN})$ and loaded according to Algorithm 4 of Appendix. The separation of load units ensures efficient handling of the differing computational demands, tile sizes, and array shapes across the attention and feedforward network modules.

4.5. Load biases unit

Bias parameters are stored in registers due to their relatively small size, enabling low-latency access during computation. Three dedicated *Load_bias* units manage the transfer of biases from external memory to the corresponding registers. The first unit supplies biases to the registers of each attention head in accordance with Algorithm 5. The same procedure, as described in Algorithm 6 of Appendix, is used to load biases for the feedforward network and the layer normalization modules. In HLS, biases are represented as one-dimensional arrays, and the application of a complete array partition pragma maps these arrays directly to registers. Since tiling is unnecessary for these small vectors, each array is loaded in a single transfer, providing all bias values simultaneously.

4.6. Activation unit

The activation functions employed within the transformer architecture are implemented at this stage. Commonly used functions include ReLU, GeLU, and softmax, each defined according to its mathematical formulation. After synthesis, these functions are realized using LUTs to support efficient hardware computation. While the implementations of ReLU and GeLU are straightforward, the softmax function involves more complex operations; therefore, only its implementation is detailed in Algorithm 7 of Appendix.

4.7. Layer Normalization unit

The Layer Normalization (LN) unit computes the mean and variance of the outputs from both the attention and feedforward network layers, following Eq. (4). These statistics are then used to normalize the outputs, which are subsequently scaled by learned weights and shifted by biases in an element-wise manner. Notably, the outputs of $FFN1_{PM}$ and $FFN3_{PM}$ are processed through the LN unit prior to subsequent stages, as illustrated in Fig. 3. The corresponding HLS implementation is provided in Algorithm 8 of Appendix.

4.8. Bias add unit

Three dedicated *Bias_add* units are employed to incorporate biases into the query (Q), key (K), and value (V) matrices, as well as into the outputs of the three feedforward network modules. Separate units are necessary because the corresponding HLS functions operate on arrays of differing dimensions, producing outputs with distinct shapes. One of the units associated with the feedforward networks additionally integrates the ReLU activation function. The operations performed by these units are detailed in Algorithms 16, 17, and 13 of Appendix.

5. System design and optimizations

The overall system design for deploying ***ADAPTOR*** across multiple FPGA platforms is presented in Fig. 4. Experiments were conducted on three representative devices: the VC707 (Virtex-7 xc7vx485tffg1761-2), ZCU102 (Zynq UltraScale+ xczu9eg-ffvb1156-2-e MPSoC), and the Alveo U55C (UltraScale+ xcu55c-fsvh2892-2L-e). While the VC707 and ZCU102 boards integrate on-board DDR3 DRAM, the Alveo U55C employs high-bandwidth memory (HBM), offering significantly greater throughput for memory-intensive workloads.

Design parameters such as the number of attention heads, embedding dimension, hidden dimension, sequence length, and the number of encoder and decoder layers can be reconfigured at runtime, up to their maximum supported values, via a MicroBlaze softcore processor using the AXI4-Lite interface. These parameters are stored in a set of registers within ***ADAPTOR*** to specify the topology of the TNN during runtime from the software. The registers are described in Table 1, along with the corresponding parameters they store. The system architecture was implemented using the Vivado 2022.1.2 design suite. The accelerator itself is encapsulated in a custom IP block generated from high-level synthesis (HLS) and integrated into the larger system. All toolflows from Xilinx-AMD were executed on a host workstation equipped with an Intel(R) Xeon(R) Gold 6130 CPU (2.10 GHz, 32 cores) and 192 GB of RAM. Data movement between the accelerator and external memory is managed through AXI4 master interfaces [41]. Depending on workload demand, the accelerator fetches inputs and weights directly from off-chip HBM or DRAM when instructed by the accelerator controller. Control signals are delivered from the processor to the accelerator via an AXI-Lite slave interface. Additionally, the processor facilitates data transfers between external memory and on-chip BRAMs, while also issuing configuration and synchronization signals to the accelerator.

The boards were connected to the host system through either a USB–JTAG interface or PCIe 3.0 × 4 link. Although the system includes a

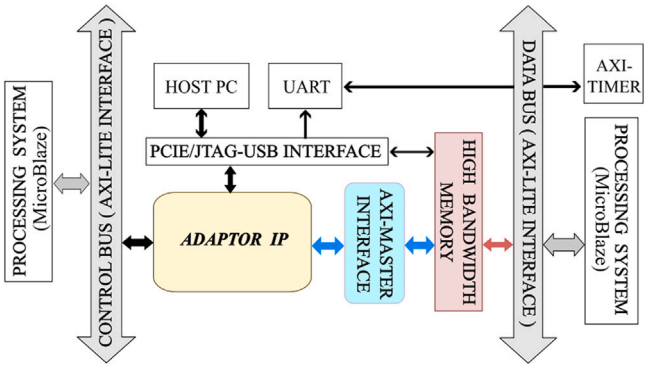


Fig. 4. Complete system design.

Table 1
Configuration registers of the *ADAPTOR*.

Register name	Description/Stored parameter
`Sequence`	Sequence length of inputs
`Heads`	Number of attention heads
`Layers_enc`	Number of encoders
`Layers_dec`	Number of decoders
`Embeddings`	Dimension of the embedding layer
`Hidden`	Dimension of the intermediate layers
`Out`	Number of outputs

DMA/Bridge Subsystem for PCIe IP [42], PCIe-based communication was not utilized in this work. Performance measurements were conducted using the AXI-TIMER [43], which recorded end-to-end latency between the initiation and completion signals of the custom IP. Final results were communicated back to the host via the UARTLite interface [44], with outputs displayed on the terminal connected through the JTAG interface [45].

The software interface illustrated in Fig. 5 is designed to communicate the programmable parameters mentioned above to the accelerator. To support this, TNN models are trained using the PyTorch framework, with the trained models stored as *'.pth'* files. In our experiments, we utilized publicly available pre-trained models from Hugging Face [46], trained on a Tesla V100 GPU. The software stack processes these files through a Python interpreter, which extracts the relevant parameter values. While these values vary across applications, the accelerator itself does not require re-synthesis for each case. Instead, only a subset of variables within the software must be reassigned to reflect the new configuration. The software, implemented in C++ using the Xilinx SDK and executed on the embedded processor, is summarized in Algorithm 18 of Appendix. Based on the extracted parameters, the processor generates the necessary instructions and control signals to configure the accelerator, thereby enabling selective activation of different hardware components.

The software control overhead varies depending on the execution environment and task complexity. Loading a PyTorch model in Jupyter Notebook typically takes less than 2 s for small models and 5 to 20 s for large transformer models. Running a Python script, which primarily generates the C code executed in Vitis IDE, generally completes within a few seconds, depending on script complexity and hardware resources. In Vitis IDE, compiling and running the generated C program on the ARM processor (with the bitstream already programmed) requires about 5 to 30 s for compilation and 1 to 5 s to download and execute the ELF file; rerunning a prebuilt ELF takes less than 2 s before execution begins. The FPGA hardware kernels, already synthesized in the bitstream, execute directly with negligible control overhead, and their performance has been reported and compared in the results section.

Transformer models are inherently large, leading to significant demands on both on-chip memory and computational resources. To address these challenges, we adopt a tiling strategy that enables efficient utilization of available hardware resources while maintaining manageable compilation times. Tiling facilitates the effective partitioning of arrays by the HLS tool, which in turn allows loop pipelining and unrolling to reduce computational latency. The proposed tiling approach for multi-head attention (MHA) is illustrated in Fig. 6(a).

In the attention module, the weight matrices are partitioned into tiles, enabling partial data loading from off-chip memory into BRAMs. Tiling is applied along the second dimension (the columns of the matrix), since the first dimension (the rows) is already reduced by the number of attention heads. Consequently, the weight matrices are loaded $\frac{d_{model}}{TS_{MHA}}$ times. Similarly, the input buffers for each attention head are defined as two-dimensional arrays of size $(SL \times TS_{MHA})$, and tiling is applied along the column dimension. The buffers are replenished $\frac{d_{model}}{TS_{MHA}}$ times, with one tile being processed at each iteration. During each iteration, the PEs compute results on the loaded tile, store intermediate results in buffers, and accumulate these with the outputs from previous iterations. The final output is thus obtained as the cumulative sum across all tiles. The feedforward networks (FFNs) following the attention layer are the most computationally demanding components of the encoder. Their weight matrices are represented as two-dimensional arrays of size $(TS_{FFN}) \times (4 \times TS_{FFN})$ and are tiled along both dimensions (rows and columns). Iterative loading is performed using two nested loops, one for each tiling dimension. As a result, the first FFN module is reused $(\frac{d_{model}}{TS_{FFN}})^2$ times, since both loops iterate $\frac{d_{model}}{TS_{FFN}}$ times. The second and third FFN modules are reused $(\frac{4\times(d_{model})^2}{(TS_{FFN})^2})$ times due to their larger dimensions. The tiling strategy for the FFN is summarized in Fig. 6(b), where intermediate results are first accumulated along the columns and subsequently along the rows to produce the final outputs across all tiles.

The tile size must be fixed before synthesis, since changing it would require re-synthesizing the hardware. Fig. 7(a) and (b) show how different choices of TS_{MHA} and TS_{FFN} affect both system frequency (MHz) and latency (normalized to the minimum value). In these experiments, the number of tiles in MHA $(\frac{d_{model}}{TS_{MHA}})$ was varied between 6 and 48, while the FFN tile count $(\frac{d_{model}}{TS_{FFN}})$ ranged from 2 to 6. The results highlight that using 24 tiles for MHA together with 6 tiles for FFN yields the best overall performance, reaching the highest frequency of 200 MHz and the lowest latency.

6. Theoretical model

The primary parameters influencing both resource utilization and performance in ***ADAPTOR*** include the tile size, or equivalently the number of tiles, in the attention module and the feedforward network, as well as the number of attention heads, sequence length, embedding dimension, hidden dimension, and the number of encoder and decoder layers, assuming a fixed bit width. The utilization of DSPs is largely determined by the degree of parallelism in multiplication operations, with the highest demand observed in the QKV_{PM}, QK_{PM}, SV_{PM}, and FFN modules. In contrast, BRAM utilization depends on the number of arrays required for intermediate data storage, the synthesis modes assigned to these memories, and the partitioning strategies specified through HLS pragmas.

To guide design-space exploration, we developed an analytical model that captures the relationship between these architectural parameters and the resulting latency and resource consumption. This model enables designers to predict performance and utilization outcomes, thereby facilitating informed parameter selection prior to full hardware synthesis.

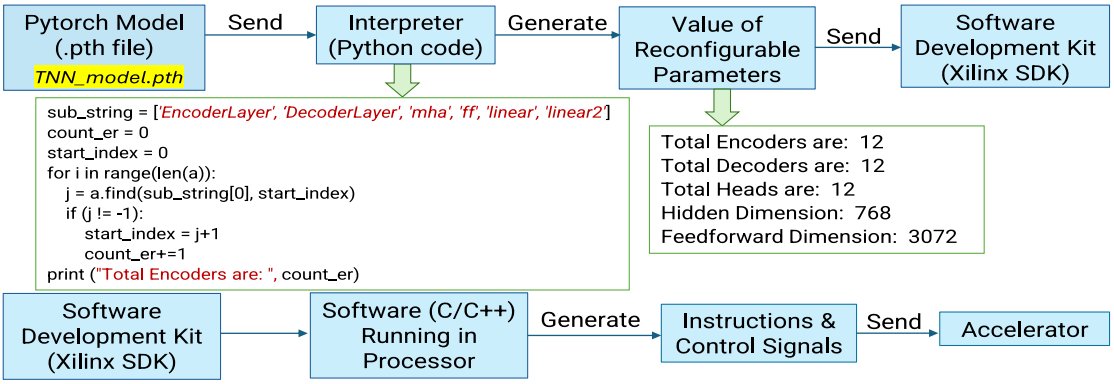


Fig. 5. Programming procedures with software.

6.1. Model for DSP utilization

Eq. (8) gives an estimate for DSP consumption. It was derived from all the loops described in the functions that generate RTL modules for QKV_{PM}, QK_{PM}, SV_{PM}, and FFN.

$$No.\ of\ DSPs = 3 \times h \times \frac{d_{model}}{Tile\ no.\ MHA} + h \times \left(\frac{d_{model}}{h} + SL\right) + 6 \times \frac{d_{model}}{Tile\ no.\ FFN} + d_{model} \tag{8}$$

The design follows a modular approach, with each module implemented as a function containing loops. The latency of a module depends on the time taken by its loops, which is affected by loop pipelining and unrolling directives. For nested loops, the second-to-last loop is pipelined, while the innermost loop is fully unrolled. The outermost loop is not modified with pragmas to avoid overly complex pipeline depth and high resource usage. The latency of a pipelined loop (PLL) can be calculated using Eq. (9). If a pipelined loop is inside another loop, the total latency (TL) is given by Eq. (10). Here, the loop trip count (TC) is the number of iterations, and the initiation interval (II) is the time between the start of two consecutive iterations. Pipeline depth is the time needed to complete one iteration, depending on the sequential and parallel operations within it. Different modules can have different pipeline depths (PD). Latency is measured in clock cycles (cc).

$$Pipelined\_Loop\_Latency = Pipeline\_Depth + Initiation\_Interval \times (Trip\_Count - 1) \tag{9}$$

$$Total\_Latency = Pipelined\_Loop\_Latency \times Outer\_Loop\_Trip\_Count \tag{10}$$

Eqs. (9) & (10) are generalized equations for measuring latency, the variables of which differ for different modules of ***ADAPTOR*** as shown in the following equations (see [47]).

6.2. Latency model for attention module

$$LI = [(d_{model} - 1) \times 1 + PD\_L] \times SL \tag{11}$$

$$LBA = (\frac{d_{model}}{h} - 1) \times 1 + PD\_L \tag{12}$$

$$LWA = [(\frac{d_{model}}{h} - 1) \times 1 + PD\_L] \times SL \tag{13}$$

$$LIA = [(\frac{d\_model}{Tile\ no.\ MHA} - 1) \times 1 + PD\_L] \times SL \tag{14}$$

where, PD_L is $Pipeline\_Depth\_Load$ that includes the time required to establish communication with HBM using AXI master interface (7 cc), read address location (1 cc), load (1 cc), and store (1 cc) data from and to that address, and convert floating point data to fixed point (3 cc) for tasks such as loading all inputs (LI), as well as loading inputs (LIA), biases (LBA) and weights (LWA) for each attention head. $Pipeline\_Depth\_MHA$ (PD_MHA) equals $(\frac{d_{model}}{TS_{MHA}})$ plus the time required to load, multiply (2 cc), add (1 cc), and store for computing self-attention (SA) in QKV_{PM} module (Eq. (15)). $Pipeline\_Depth\_Bias\_Add$ (PD_BA) includes latency associated with loading, adding, and storing operations in bias addition (BA) tasks (Eq. (16)). $Pipeline\_Depth\_Score$ (PD_S) equals $(\frac{d_{model}}{h})$, the time required to compute the score (S) in QK_{PM} module (Eq. (17)). $Pipeline\_Depth\_SV$ (PD_SV) equals $Sequence\_Length$ in the computation of SV within the SV_{PM} module (Eq. (18)). Eq. (19) estimates time for softmax (SM) calculation, which includes exponentiation (4 cc) and division (14 cc). It starts after the QK_{PM} module is finished.

$$SA = [(\frac{d\_model}{h} - 1) \times 1 + PD\_MHA] \times SL \tag{15}$$

$$BA = [(\frac{d\_model}{h} - 1) \times 1 + PD\_BA] \times SL \tag{16}$$

$$Score(S) = [(SL - 1) \times 1 + PD\_S] \times SL \tag{17}$$

$$SV = [(\frac{d\_model}{h} - 1) \times 1 + PD\_SV] \times SL \tag{18}$$

$$SM = [(SL - 1) \times 1 + Load + Store] \times SL + [(SL - 1) \times 1 + Load + Store + add + exponentiation] \times SL + [(SL - 1) \times 2 + Load + Store + divide] \times SL \tag{19}$$

6.3. Latency model for FFN1 module

$$LIF1 = [(\frac{d\_model}{Tile\ no.\ FFN} - 1) \times 1 + PD\_LFFN1] \times SL \tag{20}$$

$$LWF1 = [(\frac{d\_model}{Tile\ no.\ FFN} - 1) \times 1 + PD\_L] \times \frac{d\_model}{Tile\ no.\ FFN} \tag{21}$$

$$LBF1 = (d\_model - 1) \times 1 + PD\_L \tag{22}$$

$$FFN1 = [(\frac{d\_model}{Tile\ no.\ FFN} - 1) \times 1 + PD\_FFN1] \times SL \tag{23}$$

$$BAF1 = [(d\_model - 1) \times 1 + PD\_BA] \times SL \tag{24}$$

where, $Load\_Inputs\_FFN1$ (LIF1) unit loads tiled outputs from the attention module into the input buffer of the FFN1 module. $Load\_Weights\_FFN1$ (LWF1) unit loads partial weights from off-chip memory to the weight buffer of the FFN1 module according to TS_{FFN}. $Pipeline\_Depth\_FFN1$ (PD_FFN1) equals $(\frac{d_{model}}{Tile\ no.\ FFN})$ plus the time required to perform load, add, and store operations in the FFN1 module. $Pipeline\_Depth\_Load\_FFN1$ (PD_LFFN1) is the time required to load, add, and store in the loading units. FFN1 is the computation time of $FFN1_{PM}$ module. $Load\_Biases\_FFN$ (LBF1)

loads biases to registers from off-chip memory while $FFN1_{PM}$ operates. $Bias\_Addition\_FFN1$ (BAF1) adds biases to the outputs of $FFN1_{PM}$.

6.4. Model for BRAM utilization

Eq. (25) gives an estimate for BRAM consumption. It was derived from all the arrays declared in HLS with true dual-port BRAM pragmas.

$$\begin{aligned} No.\ of\ BRAMs = & \frac{10 \times SL \times d_{model} \times Bit\_w}{BRAM\_w \times BRAM\_d} + SL \\ & \times max\left(0.5, \frac{SL \times Bit\_w}{BRAM\_w \times BRAM\_d}\right) \\ & + max\left(0.5, \frac{SL \times d_{model} \times Bit\_w}{BRAM\_w \times BRAM\_d}\right) \\ & + \frac{h \times SL \times d_{model} \times Bit\_w}{BRAM\_w \times BRAM\_d} \\ & + max\left(0.5, \frac{d_{model} \times Bit\_w}{BRAM\_w \times BRAM\_d}\right) \\ & + \frac{SL \times Tile\ no.\ MHA \times Bit\_w}{BRAM\_w \times BRAM\_d} \\ & + Tile\ no.\ MHA \times h \times max \\ & \times \left(0.5, \frac{SL \times Bit\_w}{BRAM\_w \times BRAM\_d}\right) \\ & + \frac{8 \times d_{model}^2 \times Bit\_w}{Tile\ no.\ FFN \times BRAM\_w \times BRAM\_d} \\ & + Tile\ no.\ MHA \times h \\ & \times max\left(0.5, \frac{d_{model} \times Bit\_w}{BRAM\_w \times BRAM\_d}\right) \\ & + \frac{d_{model}}{Tile\ no.\ FFN} \times max\,(0.5, \\ & \frac{SL \times Bit\_w}{BRAM\_w \times BRAM\_d}\Big) + 4 \times d_{model} \\ & \times max\left(0.5, \frac{SL \times Bit\_w}{BRAM\_w \times BRAM\_d}\right) \end{aligned} \tag{25}$$

Here, BRAM_d is the depth of BRAMs, which indicates the number of storage locations (or entries) within a BRAM block. Each location holds a fixed number of bits, defined by the width of BRAM (BRAM_w), and both parameters can vary depending on the platform. Bit_w is the bit precision of the data being stored. BRAM_w = 36 and BRAM_d = 1024 for most FPGAs. Each term in the equation corresponds to an array declared in the HLS code. For instance, the first term represents the number of BRAMs synthesized for 10 arrays of size SL× d_{model}. The max function in the second term accounts for cases where an array may not fully utilize the 18 kb width of a synthesized BRAM, but at least one 18 kb BRAM will still be allocated. The factor 0.5 is used because the total BRAM count is calculated based on 36 kb BRAMs.

6.5. Latency model for LN module

$$LWN = (d\_model - 1) \times 1 + PD\_L \tag{26}$$

$$LBN = (d\_model - 1) \times 1 + PD\_L \tag{27}$$

$$RC = [(d\_model - 1) \times 1 + PD\_BA] \times SL \tag{28}$$

$$\begin{aligned} Layer\ Norm = & [(d\_model - 1) \times 2 + Load + Add + Store] \\ & \times SL + [(d\_model - 1) \times 2 + Load + multiply \\ & + add + store] \times SL + [(d\_model - 1) \times 1 + Load \\ & + Square + multiply + add + Store \\ & + divide + float\_to\_fixed\_conversion] \times SL \\ & + [(d\_model - 1) \times 1 + Load + add \\ & + Store] \times SL \end{aligned} \tag{29}$$

where, $Load\_Weights\_LN$ (LWN) unit loads weights from off-chip memory to the weight buffer of the LN module. $Load\_Biases\_LN$ (LBN) loads biases to registers from off-chip memory. RC represents the operations of the residual connection in LN module. $float\_to\_fixed\_conversion$ in the LN module takes 3 cc.

6.6. Latency model for FFN2 module

$$LIF2 = [(\frac{d_{model}}{Tile\ no.\ FFN} - 1) \times 1 + PD\_LFFN2] \times SL \tag{30}$$

$$LWF2 = [(\frac{d_{model}}{Tile\ no.\ FFN} - 1) \times 1 + PD\_L] \times \frac{d_{model}}{Tile\ no.\ FFN} \tag{31}$$

$$LBF2 = (d_{model} - 1) \times 1 + PD\_L \tag{32}$$

$$FFN2 = [(\frac{4 \times d_{model}}{Tile\ no.\ FFN} - 1) \times 1 + PD\_FFN2] \times SL \tag{33}$$

$$BAF2 = [(4 \times d_{model} - 1) \times 1 + PD\_BA] \times SL \tag{34}$$

where, $Load\_Inputs\_FFN2$ (LIF2) unit loads tiled outputs from the FFN1 module into the input buffer of the FFN2 module. $Load\_Weights\_FFN2$ (LWF2) unit loads partial weights from off-chip memory to the weight buffer of the FFN2 module according to TS_{FFN}. $Pipeline\_Depth\_FFN2$ equals ($\frac{d_{model}}{Tile\ no.\ FFN}$) plus the time required to perform load, add, and store operations in the FFN2 module. $Pipeline\_Depth\_Load\_FFN2$ (PD_LFFN2) is the time required to load, add, and store in the loading units. FFN2 is the computation time of $FFN2_{PM}$ module. $Load\_Biases\_FFN2$ (LBF2) loads biases to registers from off-chip memory while $FFN2_{PM}$ operates. $Bias\_Addition\_FFN2$ (BAF2) adds biases to the outputs of $FFN2_{PM}$.

6.7. Latency model for FFN3 module

$$LIF3 = [(\frac{4 \times d_{model}}{Tile\ no.\ FFN} - 1) \times 1 + PD\_LFFN3] \times SL \tag{35}$$

$$LWF3 = [(\frac{4 \times d_{model}}{Tile\ no.\ FFN} - 1) \times 1 + PD\_L] \times \frac{d_{model}}{Tile\ no.\ FFN} \tag{36}$$

$$LBF3 = (d_{model} - 1) \times 1 + PD\_L \tag{37}$$

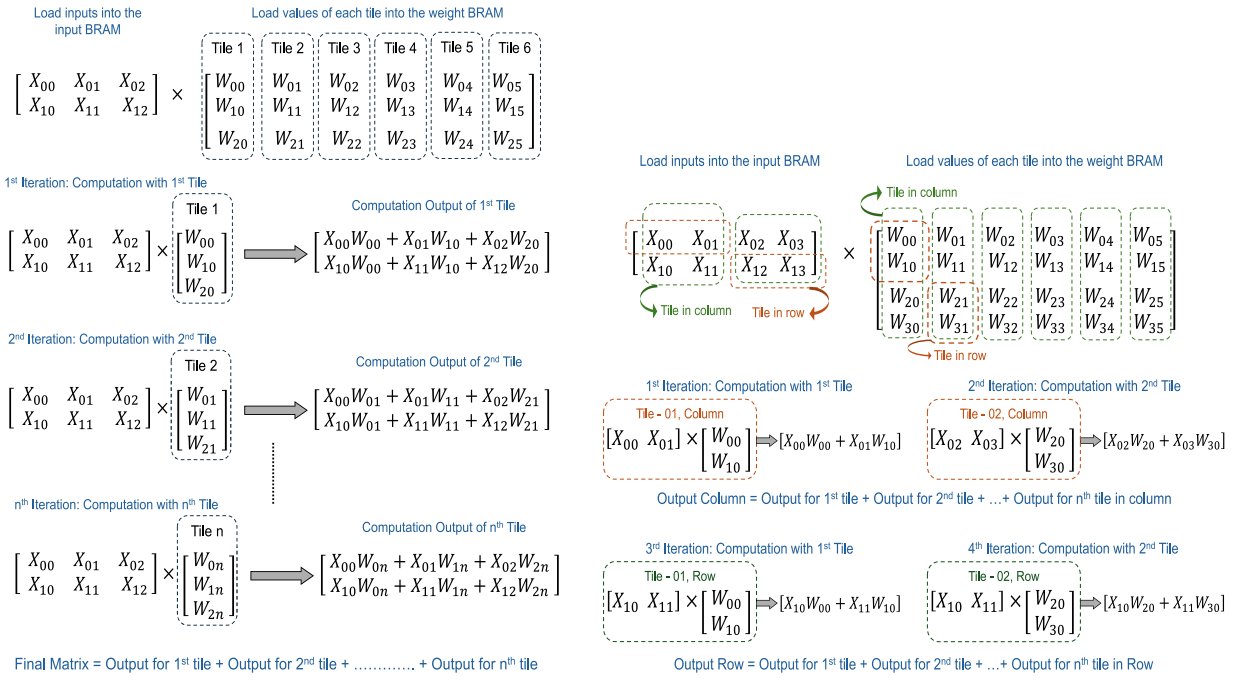
$$FFN3 = [(\frac{d_{model}}{Tile\ no.\ FFN} - 1) \times 1 + PD\_FFN3] \times SL \tag{38}$$

$$BAF3 = [(d_{model} - 1) \times 1 + PD\_BA] \times SL \tag{39}$$

where, $Load\_Inputs\_FFN3$ (LIF3) unit loads tiled outputs from the FFN2 module into the input buffer of the FFN3 module. $Load\_Weights\_FFN3$ (LWF3) unit loads partial weights from off-chip memory to the weight buffer of the FFN3 module according to TS_{FFN}. $Pipeline\_Depth\_FFN3$ equals ($\frac{4 \times d_{model}}{Tile\ no.\ FFN}$) plus the time required to perform load, add, and store operations in the FFN3 module. $Pipeline\_Depth\_Load\_FFN3$ is the time required to load, add, and store in the loading units. FFN3 is the computation time of $FFN3_{PM}$ module. $Load\_Biases\_FFN3$ (LBF3) loads biases to registers from off-chip memory while $FFN3_{PM}$ operates. $Bias\_Addition\_FFN3$ (BAF3) adds biases to the outputs of $FFN3_{PM}$.

7. Evaluation and results

ADAPTOR supports software-level programmability, allowing modification of key design parameters at runtime. These parameters include the embedding dimension (d_{model}), number of attention heads (h), number of encoder layers (N), and sequence length (SL). Initially, these parameters were configured with fixed values of 768, 12, 12, and 64, respectively, based on a BERT variant [10], which is a widely used



(a) Tiling technique in MHA. (b) Tiling technique in FFN.

Fig. 6. Tiling technique.

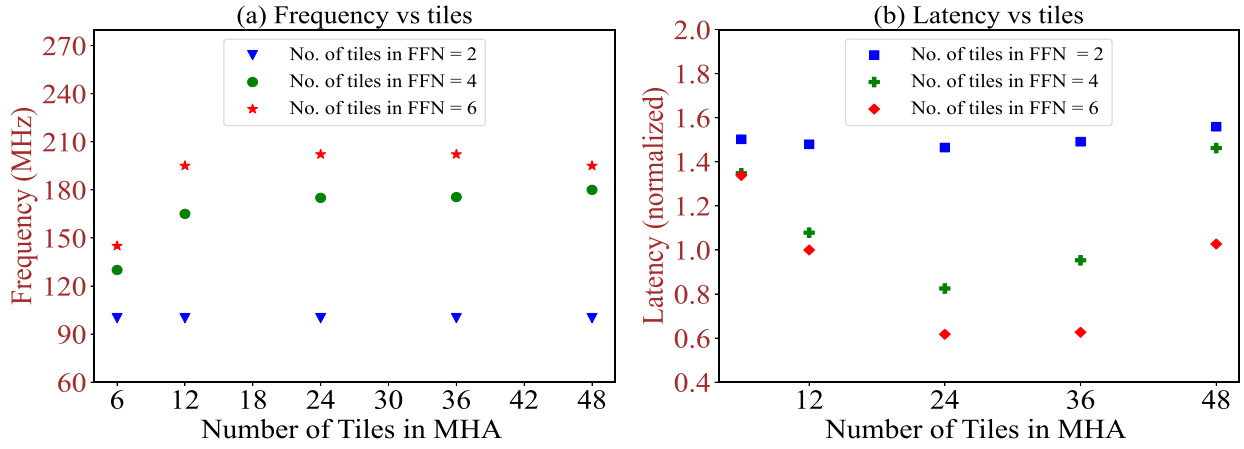
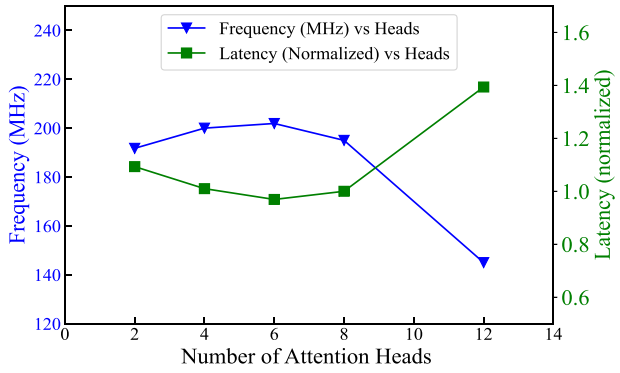


Fig. 7. Choosing the optimum tile size.

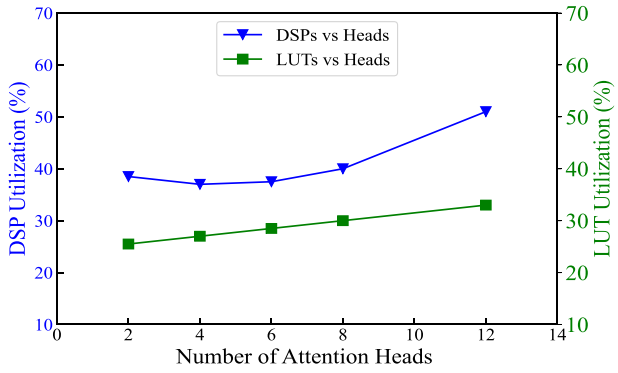
transformer model for natural language processing, and the available FPGA resources. In contrast, the tile sizes are fixed at synthesis and cannot be modified at runtime. Consequently, synthesis was performed with fixed tile sizes of $TS_{MHA} = 64$ and $TS_{FFN} = 128$. This design approach is a key contribution that gives ***ADAPTOR*** the ability to retain a single, resource-constrained synthesis configuration while enabling runtime configurability of core transformer parameters so that it can support diverse transformer neural network models without requiring re-synthesis.

Fig. 8(a) presents the effect of varying the number of attention heads on system frequency and normalized latency, where latency accounts for computation time assuming overlap with data loading. While increasing the number of attention heads generally improves parallelism and reduces latency, the system frequency decreases beyond a certain threshold, leading to higher latency. Optimal performance is observed with 6–10 attention heads. Fig. 8(b) illustrates the corresponding increase in DSP and LUT utilization, showing that higher resource usage contributes to reduced system frequency. These results provide a quantitative analysis of the trade-off between parallelism and hardware timing constraints, identify an optimal design point for FPGA-based transformers, and characterize the impact of resource utilization on latency and frequency. Furthermore, the evaluation methodology accounts for overlapped data loading and computation, offering a realistic performance assessment for hardware accelerators.

Fig. 9 illustrates the effect of varying tile sizes (TS_{MHA}, TS_{FFN}) on the utilization of DSP, LUT, and BRAM. Since processing modules rely on DSPs for multiplication–accumulation (MAC) operations, DSPs represent the most widely used resource and can reach saturation before BRAMs, rendering accelerator computation bound. Increasing the tile sizes for both the attention and feedforward modules results in higher DSP utilization, which enables greater parallelism and reduces latency until the system frequency begins to decline. This analysis is the characterization of resource–performance trade-offs, showing that tile size selection directly determines when the accelerator becomes computation-bound or frequency-limited, thus guiding optimal design choices for FPGA-based transformer implementations.



(a) Variation of performance with number of attention heads.



(b) Variation of resources with number of attention heads.

Fig. 8. Performance and resource utilization vs. attention heads.

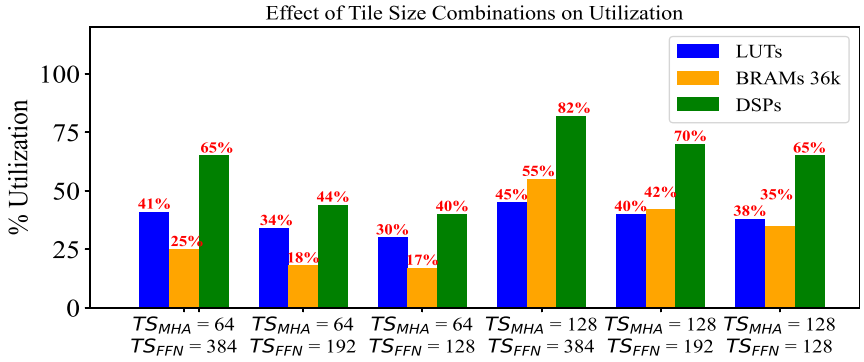


Fig. 9. Utilization vs. tile size.

Fig. 10 compares the power consumption (in watts) and power efficiency (throughput per watt, GOPS/W) for various models across different CPUs, GPUs, and our FPGA accelerator. Data for different models and platforms were obtained from cited literature, and we used them to compare the performance of ***ADAPTOR*** on the U55C platform for the same models. Since ***ADAPTOR*** is synthesized only once, and power is measured using Vivado's power estimation tool post-synthesis, the total dynamic power consumption remains constant for all models. The JETSON TX2 GPU [18] achieves the highest power efficiency for the BERT model, mainly due to the sparse architecture of the algorithm, and also has the lowest overall power consumption. The RTX K5000 GPU [48] is 1.5× more power efficient than ***ADAPTOR*** for the BERT model, due to compression techniques, but consumes 10× more power. The i7-8700K CPU is the least power-efficient for BERT [48]. ***ADAPTOR*** is 1.2× and 2.87× more power efficient than the NVIDIA K80 GPU and i7-8700K CPU, respectively, when running BERT, according to FQ-BERT [49]. A custom encoder with four encoding layers was run on an i5-4460 CPU and an RTX 3060 GPU [31], both of which were 5.1× and 1.63× less power efficient than ***ADAPTOR*** while also being more power-hungry. Fang et al. [50] executed a shallow transformer on an i9-9900X CPU, JETSON NANO GPU, RTX 2080, and RTX 3090 GPUs. Although the JETSON NANO GPU consumed 1.56× less power than ***ADAPTOR***, the other devices used 14–30× more power. However, ***ADAPTOR*** is 3.7×, 1.28×, 4.4×, and 1.67× more power efficient than all of them.

Fig. 11 illustrates that ***ADAPTOR*** can be deployed on any platform, regardless of the size of the TNN model or available resources, by adjusting the TS_{MHA} and TS_{FFN} parameters in HLS during design time. The figure presents results for a custom TNN encoder with an embedding dimension of 200, 3 attention heads, 2 encoder layers, and a sequence length of 64. On the Alveo U55C, the tile sizes can be maximized (TS_{MHA} = 200, TS_{FFN} = 200) due to the abundance of resources, resulting in lower latency. For the ZCU102 board, the tile sizes were reduced to 25 and 50 respectively, to fit the model within its resource constraints, nearly consuming 100% of the DSPs and LUTs and increasing the latency. On the VC707 board, TS_{MHA} and TS_{FFN} were set to 50 each, as it has slightly more resources than the ZCU102. However, latency increased as fewer DSPs were utilized, and LUT consumption almost reached its limit.

Fig. 12 presents the roofline model of ***ADAPTOR***, highlighting its peak performance and memory bandwidth limits. The *Memory Bound* (blue dashed line) indicates the maximum achievable performance based on the memory bandwidth, which is 103,000 GB/s. Data points to the left of this line are constrained by memory bandwidth. The *Compute Bound* (red line) represents the peak performance determined by the FPGA's computational resources, capped at 53 GOP/s. Points below this line indicate underutilization of computational resources. All data points (green, yellow, and purple) fall within the compute and memory-bound regions, meaning none fully utilize the accelerator's available resources. The yellow square, representing the BERT model with TS_{MHA} = 64 and TS_{FFN} = 192, achieves the highest performance, being closest to the compute bound. In contrast, the purple star, corresponding to the shallow transformer model with TS_{MHA} = 64 and TS_{FFN} = 128, exhibits the highest operational intensity but the lowest performance.

Eq. (40) below is used to calculate memory bandwidth (BW), where no. of BRAMs = 340, BRAM's width = 36 KB, no. of LUTRAMs = 129 101, LUTRAM's Width = 32 KB. Our previous work [51] calculated latency and throughput data (53 GOP/s).

$$\begin{aligned} Memory\ Bandwidth = (&No.\ of\ BRAMs \times BRAM's\ Width \\ &+ No.\ of\ LUTRAMs \\ &\times LUTRAM's\ Width) \times Frequency \end{aligned} \tag{40}$$

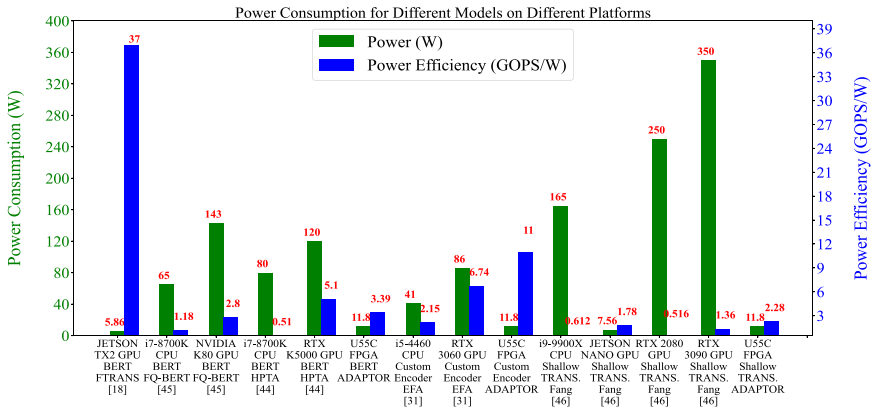


Fig. 10. Cross platform comparison of power consumption.

Table 2
Comparison with FPGA accelerators.

Accelerator	DSP	LUT	GOPS	Power (W)	(GOPS/DSP) ×1000	(GOPS/LUT) ×1000	GOPS/ Power	Method	Sparsity
Network #1	Shallow transformer								
Qi et al. [19]	3572 (52%)	485k (41%)	14	–	3.92	0.03	–		80%
Qi et al. [34]	5040 (74%)	908k (76%)	12	–	2.38	0.013	–	HLS	86%
ADAPTOR	3612 (40%)	391k (30%)	27	11.8	7.47	0.069	2.28		0%
Network #2	Custom transformer encoder								
Qi et al. [34]	4145 (60%)	937k (79%)	75.94	–	18	0.08	–	HLS	0%
ADAPTOR	3612 (40%)	391k (30%)	132	11.8	37	0.34	11		
Network #3	BERT								
Tzanos et al. [52]	5861 (85%)	910k (77%)	65.7	–	11.2	0.07	–	–	0%
TRAC [39]	1379 (80%)	126k (55%)	128	–	93	1.01	–	–	–
ADAPTOR	3612 (40%)	391k (30%)	40	11.8	11	0.10	3.39	HLS	0%

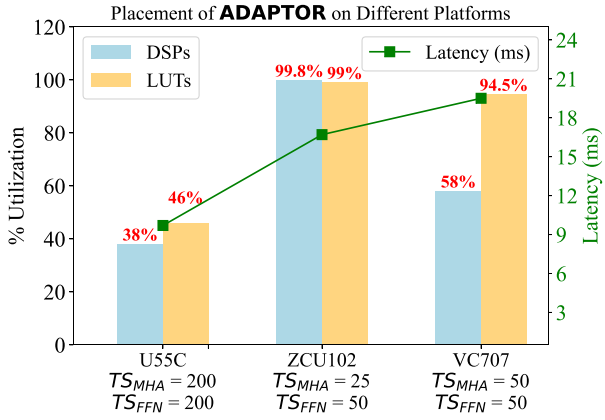


Fig. 11. Testing portability feature.

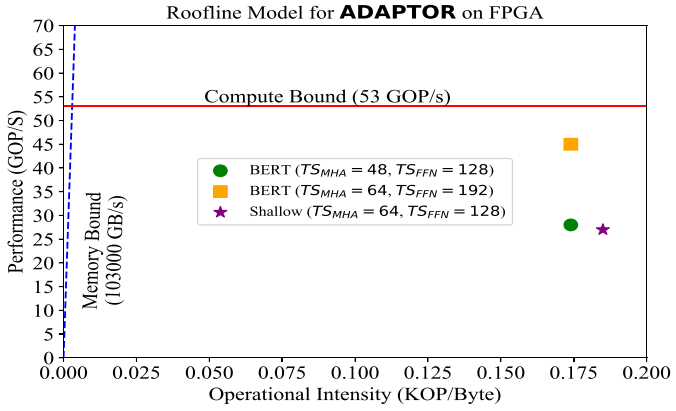


Fig. 12. Peak performance and peak memory bandwidth.

Table 2 compares the performance of our accelerator, ***ADAPTOR***, with other FPGA-based accelerators. Each of these accelerators is optimized for specific TNN models, with some designed for sparse computations. TRAC [39] is the only one that automatically generates accelerator code based on the target FPGA and TNN architecture. Since ***ADAPTOR*** was synthesized once with fixed hardware resources and bit width, and implemented on a dense model without sparsity, we evaluated throughput (GOPS), power consumption, normalized throughput (GOPS per DSP or GOPS per LUT), and power efficiency (GOPS per watt) for a fair comparison. ***ADAPTOR*** achieved 1.9× and 2.25× higher GOPS compared to the accelerators by Qi et al. in [19,34], respectively, for a shallow transformer. Its normalized throughput was also higher, indicating more efficient DSP and LUT usage without relying on pruning, whereas Qi et al. employed block balanced pruning and block row storage. Qi et al.'s four-layer transformer encoder [34] was 1.7× slower and 2× less resource-efficient than ***ADAPTOR*** even with hierarchical pruning. TRAC [39] consumed fewer DSPs and LUTs but reported 3.2× higher GOPS and 8.4× higher GOPS/DSP. None of these accelerators incorporated tiling or partitioning schemes to support large models such as BERT, which our design explicitly addresses. Tzanos et al. [52] applied tiling and used more resources, achieving 1.6× higher speed with GOPS/DSP comparable to ***ADAPTOR***.

Although ***ADAPTOR*** utilizes over 3000 DSPs at 200 MHz, the measured throughput is significantly lower than the theoretical peak of 1200 GOPS. This underutilization arises from several factors: (i)

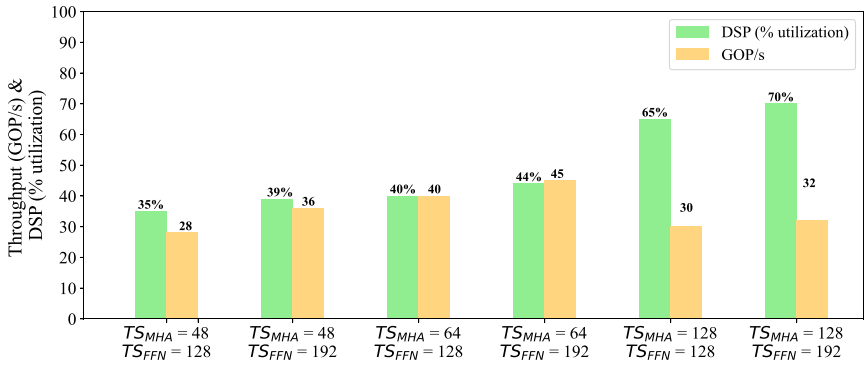


Fig. 13. Effect of DSP on GOP/s for various tile size combinations.

Table 3
Validation of experimental and analytical results.

Method	Sequence length	Embedding dimension	Number of heads	Tile size MHA	Tile size FFN	DSPs	BRAMs 18k	Frequency (MHz)	Latency (ms)		
									Attention module (SA)	Load weights unit (LWA)	FFN module (FFN1)
Analytical	64	768	8	64	128	3784	2375	200	0.052	0.037	0.082
Experimental						3612	2246		0.053	0.038	0.084
Analytical	128	768	8	64	128	3784	2375		0.103	0.037	0.165
Experimental						3612	2246		0.106	0.038	0.168
Analytical	64	512	8	64	128	3784	2375		0.042	0.025	0.055
Experimental						3612	2246		0.043	0.026	0.056
Analytical	64	768	8	128	192	6272	2955	135	0.11	0.1	0.18
Experimental						6317	1693		0.11	0.1	0.23

sequential execution of certain submodules (e.g., QKV_{PM}, QK_{PM}, SV_{PM}, $FFN1_{PM}$, $FFN2_{PM}$, $FFN3_{PM}$, softmax, layer normalization etc.) that prevents simultaneous activation of all functional modules, and (ii) control dependencies that limit pipelining across nested loops. As a result, a portion of the available DSPs remain idle at different stages of execution, reducing overall efficiency. Fig. 13 illustrates how GOPS scales with DSP consumption as the tile sizes of the MHA and FFN layers increase. While larger tiles increase DSP utilization and improve throughput, the system frequency drops beyond certain tile sizes (Fig. 7), leading to diminishing returns and even a reduction of GOPS to 30 and 32 for 65% and 70% DSP utilization, respectively. This analysis highlights the fundamental trade-off between resource utilization, frequency, and achievable throughput in FPGA-based transformer accelerators.

Table 3 presents a comparison between the experimental results of ***ADAPTOR*** and the theoretical predictions derived in Section 6. For clarity, only a subset of design configurations is reported, focusing on the computation time of the attention and feedforward modules as well as the loading time of the attention module. Latency is primarily influenced by parameters such as sequence length, embedding dimension, and number of attention heads. The measured latency closely aligned with the theoretical estimates, with an average deviation of only 1.8%. Resource utilization remained stable across configurations with fixed tile sizes, whereas variations in tile size led to corresponding changes in both analytical and experimental values. The deviations were relatively small for DSPs (0.71–4.7%), but larger for BRAMs (5.7%–74%), particularly at larger tile sizes. The latter discrepancy arises because LUTRAMs were increasingly used in place of BRAMs to sustain higher operating frequency, thereby reducing the accuracy of BRAM utilization estimates.

8. Conclusion

In this article, we present a runtime-adaptive FPGA-based accelerator for the encoder and decoder layers of transformer neural networks (TNN), designed using a high-level synthesis (HLS) tool. The architecture leverages FPGA parallelism as well as the inherent parallel nature of TNNs. We demonstrated its deployment on various FPGA platforms, including Alveo U55C, VC707, and ZCU102, highlighting how resources like DSPs and LUTs can be effectively utilized to maximize parallelism and minimize latency in HLS designs. The accelerator is software-programmable, enabling adaptability to different topologies without requiring new code generation or re-synthesis. We implemented an efficient tiling technique and data-loading method for weight matrices, ensuring portability and resource-efficient execution across different TNN models. Experimental results indicate that our design outperforms certain CPUs and GPUs in terms of dynamic power consumption and power efficiency, despite no algorithmic optimizations. Moreover, it achieved a 1.7 to 2.25× speedup over leading FPGA-based accelerators. An analytical model was also developed to validate the experimental findings.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This material is based upon work supported by the National Science Foundation, United States under Grant No. 1956071.

Appendix. Supplementary materials

Algorithm 1 Load Weights for MHA

for $(i = 1; i <= \frac{Embedding\ Dimension}{Number\ of\ Heads}; i = i + 1)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= Tiles\_in\_MHA; j = j + 1)$ **do**
    #pragma HLS pipeline II = 1
    $W_Q[i][j] \leftarrow weights\_Q[index]$;
    $W_K[i][j] \leftarrow weights\_K[index]$;
    $W_V[i][j] \leftarrow weights\_V[index]$;
    $index \leftarrow index + 1$;
  end for
end for

Algorithm 2 Load Inputs for MHA

for $(i = 1; i <= sequence\_length; i = i + 1)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= Tiles\_in\_MHA; j = j + 1)$ **do**
    #pragma HLS pipeline II = 1
    $X_1[i][j] \leftarrow input\_token[index]$;
    $X_2[i][j] \leftarrow input\_token[index]$;
    .....;
    $X_N[i][j] \leftarrow input\_token[index]$;
    $index \leftarrow index + 1$;
  end for
end for

Algorithm 3 Load Inputs for FFN1

for $(i = 1; i <= sequence\_length; i = i + 1$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j < TS_{FFN}; j = j + 1))$ **do**
    #pragma HLS pipeline II = 1
    $k \leftarrow (index) * (factor)$;
    $X_1[i][j] \leftarrow outputs\_MHA[i][k + j]$;
    $X_2[i][j] \leftarrow outputs\_MHA[i][k + j]$;
    .....;
    $X_N[i][j] \leftarrow outputs\_MHA[i][k + j]$;
    $index \leftarrow index + 1$;
  end for
end for

Algorithm 4 Load Inputs for FFN2 & FFN3

for $(i = 1; i <= sequence\_length; i = i + 1$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j < TS_{FFN}; j = j + 1))$ **do**
    #pragma HLS pipeline II = 1
    $k \leftarrow (index) * (factor)$;
    $X_1[i][j] \leftarrow outputs\_FFN2[i][k + j]$;
    $X_2[i][j] \leftarrow outputs\_FFN2[i][k + j]$;
    .....;
    $X_N[i][j] \leftarrow outputs\_FFN2[i][k + j]$;
    $index \leftarrow index + 1$;
  end for
end for

Algorithm 5 Load Biases for MHA

for $(i = 1; i <= \frac{Embedding\ Dimension}{Number\ of\ Heads}; i = i + 1)$ **do**
  #pragma HLS pipeline II = 1
  $b_q[i] \leftarrow bias\_Q[index]$;
  $b_k[i] \leftarrow bias\_K[index]$;
  $b_v[i] \leftarrow bias\_V[index]$;
  $index \leftarrow index + 1$;
end for

Algorithm 6 Load Biases for FFN & Layer Norm.

for $(i = 1; i <= Embedding\ Dimension; i = i + 1)$ **do**
  #pragma HLS pipeline II = 1
  $b_{FFN}[i] \leftarrow bias\_port[index]$;
  $index \leftarrow index + 1$;
end for

Algorithm 9 Q, K, V Calculation

for $(i = 1; i <= Sequence\ Length; i = i + 1)$ **do**
  #pragma HLS pipeline off
  $S_q \leftarrow 0$
  $S_k \leftarrow 0$
  $S_v \leftarrow 0$
  for $(k = 1; k <= \frac{d_{model}}{h}; k + +)$ **do**
    #pragma HLS pipeline II = 1
    for $(j = 1; j <= \frac{d_{model}}{TS_{MHA}}; j + +)$ **do**
      $S_q \leftarrow S_q + X[i][j] \times W_Q[k][j]$;
      $S_k \leftarrow S_k + X[i][j] \times W_K[k][j]$;
      $S_v \leftarrow S_v + X[i][j] \times W_V[k][j]$;
    end for
    $Q[i][k] \leftarrow Q[i][k] + S_q$;
    $K[i][k] \leftarrow K[i][k] + S_k$;
    $V[i][k] \leftarrow V[i][k] + S_v$;
  end for
end for

Algorithm 10 FFN3 Calculation

for $(i = 1; i <= Sequence\ Length; i = i + 1)$ **do**
  #pragma HLS pipeline off
  $m \leftarrow index \times \frac{Embedding\ Dimension}{Tiles\ in\ FFN}$
  for $(j = 1; j <= \frac{d_{model}}{Tiles\ in\ FFN}; j + +)$ **do**
    #pragma HLS pipeline II = 1
    $sum \leftarrow 0$
    for $(k = 1; k <= \frac{4 \times d_{model}}{Tiles\ in\ FFN}; k + +)$ **do**
      $sum \leftarrow sum + inputs[i][k] \times weights[k][j]$;
    end for
    $output[i][m] \leftarrow output[i][j] + sum$;
    $m \leftarrow m + 1$;
  end for
end for

Algorithm 7 Softmax

Max Value:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= SL; j++)$ **do**
    #pragma HLS pipeline II = 1
    if $x[i][j] > maxValue$ **then**
      $maxValue \leftarrow x[i][j]$
    end if
  end for
end for

Exponential:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= SL; j++)$ **do**
    #pragma HLS pipeline II = 1
    $x[i][j] \leftarrow exp(x[i][j] - maxValue)$
    $sum \leftarrow sum + x[i][j]$
  end for
end for

Normalization:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= SL; j++)$ **do**
    #pragma HLS pipeline II = 1
    $x[i][j] \leftarrow \frac{x[i][j]}{sum}$
  end for
end for

Algorithm 8 Layer Normalization

Mean:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= d_{model}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $m[i] \leftarrow m[i] + inputs[i][j]$
  end for
  $m[i] \leftarrow m[i]/Embedding\_Dimension;$
end for

Variance:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= d_{model}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $v[i] \leftarrow v[i] + (inputs[i][j] - m[i])^2$
  end for
  $m[i] \leftarrow m[i]/Embedding\_Dimension;$
end for

Normalization:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= d_{model}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $norm_{out}[i][j] \leftarrow \frac{(inputs[i][j] - m[i])}{\sqrt{v[i] + \epsilon}}$
  end for
end for

Final Output:
for $(i = 1; i <= SL; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= d_{model}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $outputs[i][j] \leftarrow gamma[j] \times norm_{out}[i][j] + beta[j];$
  end for
end for

Algorithm 11 $Q \times K^T$ Calculation

for $(i = 1; i <= SL; i = i + 1)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= SL; j = j + 1)$ **do**
    #pragma HLS pipeline II = 1
    $S \leftarrow 0$
    for $(k = 1; k <= \frac{d_{model}}{h}; k++)$ **do**
      $S \leftarrow S + Q[i][k] \times K[j][k];$
    end for
    $s[i][j] \leftarrow S/Embedding\_Dimension;$
  end for
end for

Algorithm 12 $S \times V$ Calculation

for $(i = 1; i <= SL; i = i + 1)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= \frac{d_{model}}{h}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $vv \leftarrow 0$
    for $(k = 1; k <= SL; k = k + 1)$ **do**
      $vv \leftarrow vv + S[i][k] \times V[k][j];$
    end for
    $SV[i][j] \leftarrow vv;$
  end for
end for

Algorithm 13 Bias add unit 3

for $(i = 1; i <= Sequence\ Length; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j <= Hidden\ Dimension; j++)$ **do**
    #pragma HLS pipeline II = 1
    $FF_{out}[i][j] \leftarrow FF_{out}[i][j] + bias_{FFN}[j];$
    $FF_{out}[i][j] \leftarrow relu(FF_{out}[i][j]);$
  end for
end for

Algorithm 14 FFN1 Calculation

for $(i = 1; i <= Sequence\ Length; i = i + 1)$ **do**
  #pragma HLS pipeline off
  $m \leftarrow index \times \frac{Embedding\ Dimension}{Tiles\ in\ FFN}$
  for $(j = 1; j <= \frac{d_{model}}{Tiles\ in\ FFN}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $sum \leftarrow 0$
    for $(k = 1; K <= \frac{d_{model}}{Tiles\ in\ FFN}; k++)$ **do**
      $sum \leftarrow sum + inputs[i][k] \times weights[k][j];$
    end for
    $output[i][m] \leftarrow output[i][j] + sum;$
    $m \leftarrow m + 1;$
  end for
end for

Algorithm 15 FFN2 Calculation

for $(i = 1; i <= Sequence\ Length; i++)$ **do**
  #pragma HLS pipeline off
  $m \leftarrow index \times \frac{Hidden\ Dimension}{Tiles\ in\ FFN}$
  for $(j = 1; j <= \frac{4 \times d_{model}}{Tiles\ in\ FFN}; j++)$ **do**
    #pragma HLS pipeline II = 1
    $sum \leftarrow 0$
    for $(k = 1; k <= \frac{d_{model}}{Tiles\ in\ FFN}; k++)$ **do**
      $sum \leftarrow sum + inputs[i][k] \times weights[k][j];$
    end for
    $output[i][m] \leftarrow output[i][j] + sum;$
    $m \leftarrow m + 1;$
  end for
end for

Algorithm 16 Bias add unit 1

for $(i = 1; i <= Sequence\ Length; i = i + 1)$ **do**
  #pragma HLS pipeline off
  for $(k = 1; k <= \frac{d_{model}}{h}; k++)$ **do**
    #pragma HLS pipeline II = 1
    $Q[i][k] \leftarrow Q[i][k] + bias_q[k];$
    $K[i][k] \leftarrow K[i][k] + bias_k[k];$
    $V[i][k] \leftarrow V[i][k] + bias_v[k];$
  end for
end for

Algorithm 17 Bias add unit 2

for $(i = 1; i < Sequence\ Length; i++)$ **do**
  #pragma HLS pipeline off
  for $(j = 1; j < Embedding\ Dimension; j++)$ **do**
    #pragma HLS pipeline II = 1
    $FF_{out}[i][j] \leftarrow FF_{out}[i][j] + bias_{FFN}[j];$
  end for
end for

Algorithm 18 Software Program

Assign the accelerator and other devices with IDs and base addresses
Initialize and configure the accelerator and other devices
Write to the registers of the configurable parameters: `Sequence`, `Heads`, `Layers_enc`, `Layers_dec`, `Embeddings`, `Hidden`, `Out`
for `i` from 0 to `no._of_inputs` **do** ▷ Iterate based on the number of tiles and layers
  Load input axi master interface buffers with data ▷ Same tasks for all input interfaces
end for
for `i` from 0 to `no._of_weights` **do** ▷ Iterate based on the number of tiles and layers
  Load weight axi master interface buffers with data ▷ Same tasks for all weight interfaces
end for
for `i` from 0 to `no._of_biases` **do** ▷ Iterate based on the number of tiles and layers
  Load bias axi master interface buffers with data ▷ Same tasks for all bias interfaces
end for
Write to control register to start the accelerator
Write to control register to start the timer
Record Start time
while accelerator is not done **do**
  Read status register until the accelerator has finished
end while
Record End time
Compute $Execution\_time \leftarrow End\_time - Start\_time;$

Data availability

I have shared the link of my code in the manuscript.

References

[1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al., Language models are unsupervised multitask learners, OpenAI Blog 1 (8) (2019) 9.

[2] K. Song, K. Wang, H. Yu, Y. Zhang, Z. Huang, W. Luo, X. Duan, M. Zhang, Alignment-enhanced transformer for constraining NMT with pre-specified translations, in: AAAI Conference on Artificial Intelligence, 2020, [Online]. Available: https://api.semanticscholar.org/CorpusID:213842037.

[3] T. Wang, L. Gong, C. Wang, Y. Yang, Y. Gao, X. Zhou, H. Chen, ViA: A novel vision-transformer accelerator based on FPGA, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (11) (2022) 4088–4099, [Online]. Available: https://ieeexplore.ieee.org/document/9925700/.

[4] K. Cho, B. van Merriënboer, D. Bahdanau, Y. Bengio, On the properties of neural machine translation: Encoder–decoder approaches, in: D. Wu, M. Carpuat, X. Carreras, E.M. Vecchi (Eds.), Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Association for Computational Linguistics, Doha, Qatar, 2014, pp. 103–111, [Online]. Available: https://aclanthology.org/W14-4012.

[5] S. Hochreiter, J. Schmidhuber, Long short-term memory, Neural Comput. 9 (8) (1997) 1735–1780, [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[6] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, N. Houlsby, An image is worth 16x16 words: Transformers for image recognition at scale, 2020, arXiv abs/2010.11929. [Online]. Available: https://api.semanticscholar.org/CorpusID:225039882.

[7] J.-B. Cordonnier, A. Loukas, M. Jaggi, On the relationship between self-attention and convolutional layers, in: International Conference on Learning Representations, 2020, [Online]. Available: https://openreview.net/forum?id=HJlnC1rKPB.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, Adv. Neural Inf. Process. Syst. 30 (2017).

[9] J.J. Lin, R. Nogueira, A. Yates, Pretrained transformers for text ranking: BERT and beyond, in: Proceedings of the 14th ACM International Conference on Web Search and Data Mining, 2020, [Online]. Available: https://api.semanticscholar.org/CorpusID:222310837.

[10] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, 2018, arXiv preprint arXiv:1810.04805.

[11] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, R. Soricut, Albert: A lite bert for self-supervised learning of language representations, 2019, arXiv preprint arXiv:1909.11942.

[12] W. Wang, B. Bi, M. Yan, C. Wu, Z. Bao, J. Xia, L. Peng, L. Si, Structbert: Incorporating language structures into pre-training for deep language understanding, 2019, arXiv preprint arXiv:1908.04577.

[13] H. Peng, S. Huang, S. Chen, B. Li, T. Geng, A. Li, W. Jiang, W. Wen, J. Bi, H. Liu, C. Ding, A length adaptive algorithm-hardware co-design of transformer on FPGA through sparse attention and dynamic pipelining, in: Proceedings of the 59th ACM/IEEE Design Automation Conference, ACM, San Francisco California, 2022, pp. 1135–1140, [Online]. Available: https://dl.acm.org/doi/10.1145/3489517.3530585.

[14] T.J. Ham, Y. Lee, S.H. Seo, S. Kim, H. Choi, S.J. Jung, J.W. Lee, ELSA: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks, in: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA, 2021, pp. 692–705, ISSN: 2575-713X.

[15] P. Rajpurkar, R. Jia, P. Liang, Know what you don't know: Unanswerable questions for SQuAD, in: I. Gurevych, Y. Miyao (Eds.), Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), Association for Computational Linguistics, Melbourne, Australia, 2018, pp. 784–789, [Online]. Available: https://aclanthology.org/P18-2124.

[16] S. Zeng, J. Liu, G. Dai, X. Yang, T. Fu, H. Wang, W. Ma, H. Sun, S. Li, Z. Huang, Y. Dai, J. Li, Z. Wang, R. Zhang, K. Wen, X. Ning, Y. Wang, FlightLLM: Efficient large language model inference with a complete mapping flow on FPGAs, in: Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, New York, USA, 2024, [Online]. Available: https://doi.org/10.1145/3626202.3637562.

[17] P. Ganesh, Y. Chen, X. Lou, M.A. Khan, Y. Yang, H. Sajjad, P. Nakov, D. Chen, M. Winslett, Compressing large-scale transformer-based models: A case study on bert, Trans. Assoc. Comput. Linguist. 9 (2021) 1061–1080.

[18] B. Li, S. Pandey, H. Fang, Y. Lyv, J. Li, J. Chen, M. Xie, L. Wan, H. Liu, C. Ding, FTRANS: energy-efficient acceleration of transformers using FPGA, in: Proceedings of the ACM/IEEE international symposium on low power electronics and design, ACM, Boston Massachusetts, 2020, pp. 175–180, [Online]. Available: https://dl.acm.org/doi/10.1145/3370748.3406567.

[19] P. Qi, Y. Song, H. Peng, S. Huang, Q. Zhuge, E.H.-M. Sha, Accommodating transformer onto FPGA: Coupling the balanced model compression and FPGA-implementation optimization, in: Proceedings of the 2021 on Great Lakes Symposium on VLSI, ACM, Virtual Event USA, 2021, pp. 163–168, [Online]. Available: https://dl.acm.org/doi/10.1145/3453688.3461739.

[20] K. Guo, S. Zeng, J. Yu, Y. Wang, H. Yang, [DL] A survey of FPGA-based neural network inference accelerators, ACM Trans. Reconfigurable Technol. Syst. 12 (1) (2019) [Online]. Available: https://doi.org/10.1145/3289185.

[21] M. Rognlien, Z. Que, J.G.F. Coutinho, W. Luk, Hardware-aware optimizations for deep learning inference on edge devices, in: L. Gan, Y. Wang, W. Xue, T. Chau (Eds.), Applied Reconfigurable Computing. Architectures, Tools, and Applications, in: Lecture Notes in Computer Science, vol. 13569, Springer Nature Switzerland, Cham, 2022, pp. 118–133, [Online]. Available: https://link.springer.com/10.1007/978-3-031-19983-7_9.

[22] S. Lu, M. Wang, S. Liang, J. Lin, Z. Wang, Hardware accelerator for multi-head attention and position-wise feed-forward in the transformer, in: 2020 IEEE 33rd International System-on-Chip Conference, SOCC, IEEE, Las Vegas, NV, USA, 2020, pp. 84–89, [Online]. Available: https://ieeexplore.ieee.org/document/9524802/.

[23] T.J. Ham, S. Jung, S. Kim, Y.H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J.W. Lee, D.-K. Jeong, A3: Accelerating attention mechanisms in neural networks with approximation, in: 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2020, pp. 328–341, [Online]. Available: https://api.semanticscholar.org/CorpusID:211296403.

[24] W. Ye, X. Zhou, J. Zhou, C. Chen, K. Li, Accelerating attention mechanism on FPGAs based on efficient reconfigurable systolic array, ACM Trans. Embed. Comput. Syst. 22 (6) (2023) 1–22, [Online]. Available: https://dl.acm.org/doi/10.1145/3549937.

[25] X. Zhang, Y. Wu, P. Zhou, X. Tang, J. Hu, Algorithm-hardware Co-design of attention mechanism on FPGA devices, ACM Trans. Embed. Comput. Syst. 20 (5s) (2021) 1–24, [Online]. Available: https://dl.acm.org/doi/10.1145/3477002.

[26] S. Hur, S. Na, D. Kwon, J. Kim, A. Boutros, E. Nurvitadhi, J. Kim, A fast and flexible FPGA-based accelerator for natural language processing neural networks, ACM Trans. Arch. Code Optim. 20 (1) (2023) [Online]. Available: https://doi.org/10.1145/3564606.

[27] H. Peng, S. Huang, T. Geng, A. Li, W. Jiang, H. Liu, S. Wang, C. Ding, Accelerating transformer-based deep learning models on FPGAs using column balanced block pruning, in: 2021 22nd International Symposium on Quality Electronic Design, ISQED, IEEE, Santa Clara, CA, USA, 2021, pp. 142–148, [Online]. Available: https://ieeexplore.ieee.org/document/9424344/.

[28] Z. Jiang, D. Yin, E.E. Khoda, V. Loncar, E. Govorkova, E. Moreno, P. Harris, S. Hauck, S.-C. Hsu, Ultra fast transformers on FPGAs for particle physics experiments.

[29] F. Wojcicki, Z. Que, A.D. Tapper, W. Luk, Accelerating transformer neural networks on FPGAs for high energy physics experiments, in: 2022 International Conference on Field-Programmable Technology, ICFPT, IEEE, Hong Kong, 2022, pp. 1–8, [Online]. Available: https://ieeexplore.ieee.org/document/9974463/.

[30] Y. Chen, T. Li, X. Chen, Z. Cai, T. Su, High-frequency systolic array-based transformer accelerator on field programmable gate arrays, Electronics 12 (4) (2023) 822, [Online]. Available: https://www.mdpi.com/2079-9292/12/4/822. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

[31] X. Yang, T. Su, EFA-trans: Anefficient and flexible acceleration architecture for transformers, Electronics 11 (21) (2022) 3550, [Online]. Available: https://www.mdpi.com/2079-9292/11/21/3550.

[32] Y. Bai, F. University, LTrans-OPU: A low-latency FPGA-based overlay processor for transformer networks.

[33] E. Kabir, D. Coble, J.N. Satme, A.R. Downey, J.D. Bakos, D. Andrews, M. Huang, Accelerating LSTM-based high-rate dynamic system models, in: 2023 33rd International Conference on Field-Programmable Logic and Applications, FPL, 2023, pp. 327–332.

[34] P. Qi, E.H.-M. Sha, Q. Zhuge, H. Peng, S. Huang, Z. Kong, Y. Song, B. Li, Accelerating framework of transformer by hardware design and model compression co-optimization, in: 2021 IEEE/ACM International Conference on Computer Aided Design, ICCAD, IEEE, Munich, Germany, 2021, pp. 1–9, [Online]. Available: https://ieeexplore.ieee.org/document/9643586/.

[35] H. Chen, J. Zhang, Y. Du, S. Xiang, Z. Yue, N. Zhang, Y. Cai, Z. Zhang, Understanding the potential of FPGA-based spatial acceleration for large language model inference, ACM Trans. Reconfigurable Technol. Syst. 18 (1) (2025) 1–29, [Online]. Available: https://dl.acm.org/doi/10.1145/3656177.

[36] Y. Qin, W. Lou, C. Wang, L. Gong, X. Zhou, Enhancing long sequence input processing in FPGA-based transformer accelerators through attention fusion, in: Proceedings of the Great Lakes Symposium on VLSI 2024, ACM, Clearwater FL USA, 2024, pp. 599–603, [Online]. Available: https://dl.acm.org/doi/10.1145/3649476.3658810.

[37] S. Hur, S. Na, D. Kwon, J. Kim, A. Boutros, E. Nurvitadhi, J. Kim, A fast and flexible FPGA-based accelerator for natural language processing neural networks, ACM Trans. Archit. Code Optim. 20 (1) (2023) 1–24, [Online]. Available: https://dl.acm.org/doi/10.1145/3564606.

[38] Y. Bai, H. Zhou, K. Zhao, H. Wang, J. Chen, J. Yu, K. Wang, FET-OPU: A flexible and efficient FPGA-based overlay processor for transformer networks, in: 2023 IEEE/ACM International Conference on Computer Aided Design, ICCAD, IEEE, San Francisco, CA, USA, 2023, pp. 1–9, [Online]. Available: https://ieeexplore.ieee.org/document/10323752/.

[39] P. Plagwitz, F. Hannig, J. Teich, TRAC: Compilation-based design of transformer accelerators for FPGAs, in: 2022 32nd International Conference on Field-Programmable Logic and Applications, FPL, IEEE, Belfast, United Kingdom, 2022, pp. 17–23, [Online]. Available: https://ieeexplore.ieee.org/document/10035242/.

[40] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, D. Chen, ScaleHLS: A new scalable high-level synthesis framework on multi-level intermediate representation, 2021, arXiv:2107.11673. [Online]. Available: https://arxiv.org/abs/2107.11673.

[41] AMD Technical Information Portal — docs.amd.com, https://docs.amd.com/r/en-US/ug1399-vitis-hls/AXI4-Master-Interface.

[42] Introduction • DMA/Bridge subsystem for PCI express product guide (PG195) • reader • documentation portal. [Online]. Available: https://docs.xilinx.com/r/en-US/pg195-pcie-dma.

[43] AMD Technical Information Portal — docs.amd.com, https://docs.amd.com/v/u/en-US/axi_timer_ds764.

[44] AMD Technical Information Portal — docs.amd.com, https://docs.amd.com/v/u/en-US/axi_uartlite_ds741.

[45] Programmers — digilent.com, https://digilent.com/shop/fpga-boards/programmers/.

[46] BERT — huggingface.co, https://huggingface.co/docs/transformers/en/model_doc/bert.

[47] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, B. He, Performance modeling and directives optimization for high-level synthesis on FPGA, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 39 (7) (2020) 1428–1441, [Online]. Available: https://ieeexplore.ieee.org/document/8695879/.

[48] Y. Han, T. University, HPTA: A high performance transformer accelerator based on FPGA.

[49] Z. Liu, G. Li, J. Cheng, Hardware acceleration of fully quantized BERT for efficient natural language processing, in: 2021 Design, Automation & Test in Europe Conference & Exhibition, DATE, IEEE, Grenoble, France, 2021, pp. 513–516, [Online]. Available: https://ieeexplore.ieee.org/document/9474043/.

[50] C. Fang, A. Zhou, Z. Wang, An algorithm-hardware co-optimized framework for accelerating N:M sparse transformers, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 30 (11) (2022) 1573–1586, [Online]. Available: http://arxiv.org/abs/2208.06118. arXiv:2208.06118 [cs].

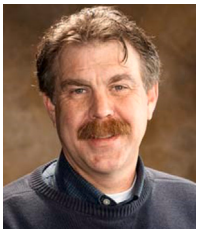
[51] E. Kabir, J.D. Bakos, D. Andrews, M. Huang, ProTEA: Programmable transformer encoder acceleration on FPGA, in: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2024, pp. 521–530.

[52] G. Tzanos, C. Kachris, D. Soudris, Hardware acceleration of transformer networks using FPGAs, in: 2022 Panhellenic Conference on Electronics & Telecommunications, PACET, IEEE, Tripolis, Greece, 2022, pp. 1–5, [Online]. Available: https://ieeexplore.ieee.org/document/9976354/.

Dr. Ehsan Kabir has been working as a Lecturer of the Computer Engineering program at Texas A & M University Texarkana since August 2025. He has recently completed his Ph.D. in Computer Engineering at the University of Arkansas, Fayetteville, where his research focused on developing machine learning accelerators using FPGA technology. He received his BSc. in Electrical & Electronics Engineering in 2016. Over the past five years, through a combination of research, coursework, and hands-on projects, he has built strong expertise in FPGAs, embedded systems, machine learning, RTL and HLS design, and hardware–software co-design. He is now seeking opportunities to apply these skills in innovative and impactful ways. His research interests are reconfigurable computing, embedded systems, and machine learning.

Dr. Jason D. Bakos is a Professor of Computer Science and Engineering at the University of South Carolina, Columbia. His research focuses on high-performance domain-specific architectures, including those based on reconfigurable, graphical, many-core, digital signal, automata, and neuromorphic processor technology. He is currently serving as associate editor for ACM Transactions on Reconfigurable Technology and Systems (TRETS). Dr. Bakos received his Ph.D. in Computer Science from the University of Pittsburgh in 2005 and his B.S. in Computer Science from Youngstown State University in 1999.

Dr. Miaoqing Huang is an associate professor in the Department of Electrical Engineering and Computer Science at the University of Arkansas. He received his BS degree in electronics and information systems from Fudan University, China, in 1998, and the PhD degree in computer engineering from The George Washington University, in 2009. His research interests are Heterogeneous many-core architecture, Hardware-oriented security, high-performance computing, and hardware design.



Dr. David Andrews is a professor of Electrical Engineering & Computer Science at the University of Arkansas, Fayetteville. He joined there as the Mullins Endowed Chair of Computer Engineering in 2008. His research interests are in the general area of embedded systems architectures, parallel and distributed real-time systems, and reconfigurable computing. He received his Ph.D. from Syracuse University in 1992.