A Scalable GPT-2 Inference Hardware Architecture on FPGA

Anil Yemme
Dept. of Electronics Systems Engineering
Indian Institute of Science
Bengaluru, India
anily@iisc.ac.in

Shayan Srinivasa Garani
Dept. of Electronics Systems Engineering
Indian Institute of Science
Bengaluru, India
shayangs@iisc.ac.in

2023 International Joint Conference on Neural Networks (IJCNN) | 978-1-6654-8867-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/IJCNN54540.2023.10191067

***Abstract*—Transformer-based architectures using attention mechanisms are a class of learning architectures for sequence processing tasks. These include architectures such as the generative pretrained transformer (GPT) and the bidirectional encoder representations from transformers (BERT). GPT-2 is a popular sequence learning architecture that uses transformer architecture. GPT-2 is trained on text prediction, and the network parameters obtained during this training process can be used in various other tasks like text classification and premise-hypothesis testing. Edge computing is an recent trend in which training is done on cloud or server with multiple GPUs, but inference is done on edge devices like mobile phones to reduce latency and improve privacy. This necessitates a study of GPT-2 performance and complexity to distill hardware-based architectures for their usability on edge devices. In this paper, a single layer of GPT-2 based inference architecture is implemented on Virtex-7 xc7vx485tffg1761-2 FPGA board. The inference engine has model dimensionality of 128 and latency of 1.637 ms while operating at 142.44 MHz, consuming 85.6K flip-flops and 96.8K lookup tables, achieving 1.73x speedup compared to previously reported work on transformer-based architecture. The approach proposed in this paper is scalable to models of higher dimensionality.**

***Index Terms*—GPT, transformer, neural networks, hardware architecture**

I. INTRODUCTION

Natural language processing (NLP) is a branch of artificial intelligence that deals with various language-related tasks like text prediction, language translation, question answering, text summarization etc. Historically, NLP was implemented using handwritten rules and statistical models like Markov models were used for text prediction. The availability of big datasets and growth of computation power ushered the age of neural NLP, which are primarily based on neural networks. Recurrent neural networks (RNN) and long short-term memories (LSTM) [1] belongs to a class of neural NLPs that had been widely used to learn dependencies and patterns within text sequences.

However, training speeds of both RNNs and LSTMs are limited due to their inherent sequential computations. This is better understood by unrolling a single copy of RNN a number of times with the same shared network parameters. So, processing of inputs and computation of outputs is only possible when all the previous outputs are already computed. This recurrence relation fundamentally limits concurrent computation of network outputs.

Attention mechanism is a recent neural network architecture which overcomes the limitations of RNNs and LSTMs [2]. Fig. 1 provides a block-level schematic of the attention mechanism used for text generation. It uses attention function to compute the alignment between input X_i and all its preceding inputs $\{X_1, X_2, ..., X_{i-1}\}$ to predict the output Y_i. Since inputs $\{X_j\}_{j=1}^{i}$ and outputs $\{Y_j\}_{j=1}^{i}$ are vectors, dot product is used as a measure of alignment inside the attention function [1]. For a particular input X_i, we need to process inputs $\{X_j\}_{j=1}^{i}$ in that order. Thus attention mechanism can compute all its outputs parallelly because there is no dependency between output Y_i and Y_{i-1}.

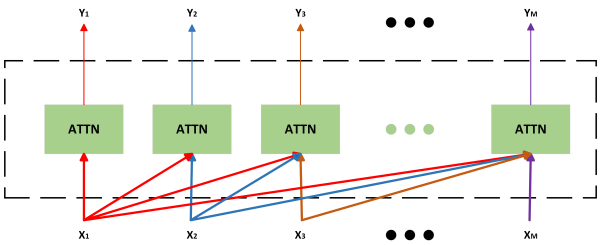

Fig. 1. Attention mechanism for text generation. Attention architecture processes all the inputs $\{X_1, X_2, X_3, \ldots, X_N\}$ simultaneously and produces outputs $\{Y_1, Y_2, Y_3, \ldots, Y_N\}$ concurrently, whereas RNNs and LSTMs compute outputs sequentially.

While previous neural NLPs used RNNs along with attention layers, transformers are a recent class of NLP engines that solely use attention mechanism, thereby achieving parallel processing during the training phase.

Bidirectional Encoder Representation from Transformers (BERT) [3] and Generative Pretrained Transformer version 3 (GPT-3) are few of the recent transformer-based architectures which have outperformed traditional neural networks like

Y. Anil was financially supported by the Ministry of Education, Govt. of India for his Masters.

[1]Attention function in transformer architecture computes alignment between X_i and X_j. However, we can also include second order relationship by computing alignment between sequentially concatenated vectors in the set $\{X_{i-1}, X_i\}$ and $\{X_{j-1}, X_j\}$.

RNNs and LSTMs in various natural language processing tasks like text generation and text classification. However, we consider GPT-2 architecture since it is the latest open-source version in GPT series [4].

GPT architecture forms the backbone for recently popular neural networks like DALL-E (image generation based on text input), ChatGPT (chatbot with near human-level sematic and contextual response) and MuseNet(music generation). Hence, analysis of GPT is relevant and necessary to optimize its performance on edge devices, which is a recent computing technique to implement inference on edge devices to reduce latency and provide enhanced privacy protection.

To the best of our knowledge, no prior hardware implementation of GPT-2 based architecture is reported, though implementations of the transformer architecture is reported in literature [5] [6]. The key components of the inference engine are generalized matrix multiplication unit (GEMM), softmax unit and add-norm unit. Though previous research [7] exists on implementation of GEMM using high-level synthesis tools, they were focused on optimizing GEMM unit in a standalone case.

In this paper, all the key components are designed and integrated using high-level synthesis tool to produce a pipelined GPT-2 inference architecture optimized for latency, throughput and resource utilization. The inference engine proposed in this paper achieves a significant reduction in latency compared to previously reported transformer-based inference engines. Since the components like GEMM, softmax unit and add-norm unit are ubiquitous in almost all the large language models (LLM) like GPT, BERT and LaMDA, the approach presented in this paper can be scaled and used to improve the performance and latency of various LLMs.

The paper is organized as follows. Section 2 provides a comprehensive description of GPT-2 architecture. Section 3 presents the details regarding the software implementation. Section 4 deals with the hardware implementation of GPT-2 inference engine and its key components along various optimizations and design tradeoffs. Section 5 provides a summary of the design implementation results. Section 6 summarizes the work and discusses the scope for future work.

II. GPT-2 ARCHITECTURE

A. GPT-2 Model

This subsection briefly describes the key components and flow of data in the GPT-2 model [4].A sequence of M input words, also referred to as input tokens $\{t_1, t_2, ..., t_M\}$ are fed as inputs to the GPT-2 model. Each of these input tokens is used as address indices to read out the corresponding word embedding vectors $\{\underline{w}_i\} \in \mathbb{R}^N$ and positional embedding vectors $\{\underline{p}_i\} \in \mathbb{R}^N$, where N is the model dimensionality.

The word embedding vector and corresponding positional embedding vector are added and fed to to the GPT-2 stack. Hence the input to the GPT-2 stack is a $M \times N$ matrix, where each row constitutes a token which is represented by a vector in $\mathbb{R}^N$ space. Fig. 2 illustrates the top-level view of GPT-2 model. The input data is processed by 12 layers of the GPT-2 stack. Each of these layers contain an attention sublayer, position-wise feedforward sublayer and 'add and normalization' sublayers. The final layer produces the output tokens, which can be interrupted as the predictions for text generation.

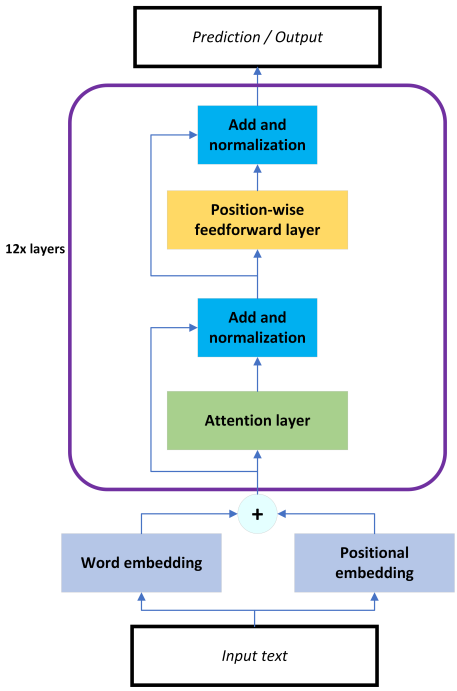

Fig. 2. GPT-2 architecture contains stack of 12 transformer decoder layers. Each layer contains an attention layer and a feedforward layer.

B. Embedding

A sequence of M input words, also referred to as input tokens $\{t_1, t_2, ..., t_M\}$ are fed as inputs to the GPT-2 model. One of the basic datasets used in NLP training is the decades old *Penn Treebank* [8] and it has vocabulary size V (number of unique words in dataset) of 10,000. Representing each word as a one-hot encoded vector would result in $x_i \in \mathbb{R}^{10000}$ space which is computationally not feasible.

Word embedding [9] is a dimensionality reduction technique that is used to represent these tokens in a lower-dimensional space $\mathbb{R}^N$ where $N << V$. It can also group words that are contextually related to each other. Because of these advantages, recent neural NLP engines use word embedding.

Ordering of the words contains contextual meaning and profoundly affects the meaning and grammatical structure of the sentence. RNNs and LSTMs processes the input tokens sequentially and hence the positional information is implicitly incorporated. However, in case of GPT-2, all the input tokens are processed concurrently. Transformer uses the scheme described in equations (1) and (2) to embed positional information, where i is the positional index, d is the dimension in which the positional embedding operates and F is set appropriately [2]. From equations (1) and (2), we observe that

the frequency of oscillations decrease as we move towards higher dimensions.

$$PE_{i,2\,d} = \sin\left(\frac{i}{F^{\frac{2\,d}{N}}}\right). \tag{1}$$

$$PE_{i,2\,d+1} = \cos\left(\frac{i}{F^{\frac{2\,d}{N}}}\right). \tag{2}$$

In the GPT-2 model, word embedding is used along with positional embedding to retain the positional information. Table I shows this process of obtaining the final text embedding. Note that the word embedding is independent of the position of the token. So, token '*the*' has the same word embedding at both positions 1 and 5. Note that $\{t_i\}_{i=1}^{V}$ represents the words from the dataset. Vectors $\{\underline{w}_i\}$, $\{\underline{p}_i\}$ and $\{\underline{x}_i\} \in \mathbb{R}^N$ vector space where N is the model dimensionality.

TABLE I
TEXT EMBEDDING FOR A SIMPLE EXAMPLE

Words $\{t_i\}$	the	sun	rises	in	the	east
Word embed $\{w_i\}$	w_1	w_2	w_3	w_4	w_1	w_5
Pos embed $\{p_i\}$	p_1	p_2	p_3	p_4	p_5	p_6
Text embed $\{x_i\}$	w_1 $+p_1$	w_2 $+p_2$	w_3 $+p_3$	w_4 $+p_4$	w_1 $+p_5$	w_5 $+p_6$

C. Attention

GPT-2 is based on attention mechanism. Attention mechanism uses three different representations of its input vectors $\{x_1, x_2, ..., x_M\}$ internally to compute its outputs $\{y_1, y_2, ..., y_M\}$. The three different representations, namely query, key and value are obtained by linear transformations of input vectors. Equations (3), (4), and (5) illustrate the process of obtaining query ($\underline{q}_i$), key ($\underline{k}_i$), and value ($\underline{v}_i$) vectors. Note that the vectors $\{\underline{x}_i, \underline{q}_i, \underline{k}_i, \underline{v}_i\} \in \mathbb{R}^N$ and weight matrices for these linear transforms $\{\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v\} \in \mathbb{R}^{N\times N}$.

$$\underline{q}_i = \mathbf{W}_q \underline{x}_i. \tag{3}$$

$$\underline{k}_i = \mathbf{W}_k \underline{x}_i. \tag{4}$$

$$\underline{v}_i = \mathbf{W}_v \underline{x}_i. \tag{5}$$

In the attention mechanism, the representation of tokens must be influenced by other input tokens in the sequence that have similar embedding. Each token must attend to every other previous token in the sequence and compute the similarity score based on a compatibility function (cf). Scaled dot product function (6) is a used as compatibility function in the transformer model. This function takes two inputs, namely, query ($\underline{q}_i$) and key ($\underline{k}_j$), and returns the compatibility score $cs_{i,j}$.

$$cs_{i,j} = \mathrm{cf}(\underline{q}_i, \underline{k}_j) = \frac{<\underline{q}_i, \underline{k}_j>}{\sqrt{N}}. \tag{6}$$

The final representation $\underline{y}_i$ for each input token $\underline{x}_i$ is obtained by performing a weighted summations of value $\underline{v}_i$ vector. Compatibility scores $cs_{i,j}$ are used as weights while performing the weighted summation. Since the compatibility scores $s_{i,j}$ are not normalized, softmax operation (7) is performed to normalize these scores.

$$s_{i,j} = \mathrm{softmax}\left(\mathrm{cf}(\underline{q}_i, \underline{k}_j)\right) = \frac{e^{cs_{i,j}}}{\sum_{j=1}^{i} e^{cs_{i,j}}}. \tag{7}$$

Masked attention layer used in GPT-2 obeys the causality principle i.e. only the previous tokens can affect the representation of the present token, whereas in BERT, future tokens can also affect the representation. Therefore, masked attention layer takes input vector $\underline{x}_i$, computes query, key, and values (linear transformation of input token) and predicts the next token i.e. output token $\underline{y}_i$. Equation (8) describes the formulation of scaled dot-product attention function .

$$\underline{y}_i = \mathrm{attn}(\underline{x}_i) = \sum_{j=1}^{j=i} s_{i,j}\underline{v}_i. \tag{8}$$

Note that the above process must be repeated for all the input vectors $\{x_i\}$ $\forall i \in [1, M]$. Stacking each of the input token vectors as columns of a matrix offers an efficient way to compute attentions for all the input tokens because matrix multiplication also involves vector dot product. Equations (9), (10) and (11) illustrates this computation for **X**, **Q** and **Y**. Matrices **K**, and **V** follow similar procedure.

$$\mathbf{X}_{M\times N} = \begin{bmatrix} | & | & | & | & | & | \\ \underline{x}_1 & \underline{x}_2 & \cdot & \cdot & \cdot & \underline{x}_M \\ | & | & | & | & | & | \end{bmatrix}^T \tag{9}$$

$$\mathbf{Q}_{M\times N} = \begin{bmatrix} | & | & | & | & | & | \\ \underline{q}_1 & \underline{q}_2 & \cdot & \cdot & \cdot & \underline{q}_M \\ | & | & | & | & | & | \end{bmatrix}^T \tag{10}$$

$$\mathbf{Y}_{M\times N} = \begin{bmatrix} | & | & | & | & | & | \\ \underline{y}_1 & \underline{y}_2 & \cdot & \cdot & \cdot & \underline{y}_M \\ | & | & | & | & | & | \end{bmatrix}^T \tag{11}$$

Since matrices **X**, **Q**, **K**, and **V** are generated by row wise stacking of M vectors belonging to $\mathbb{R}^N$ vector spaces, all of them have matrix sizes of $M \times N$, and the softmax function operates independently on each row of a matrix. Equation (12) summarizes the entire computations using matrix formulation.

$$\mathbf{Y} = \mathrm{attn}(\mathbf{X}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\,\mathbf{K}^T}{\sqrt{N}}\right)\mathbf{V}. \tag{12}$$

D. Position-wise feedforward layer

A fully connected multi-layer preceptor (MLP) is one of the most used network architectures in various machine learning tasks. GPT-2 has feedforward layers in each of its decoder stacks. It is named as 'Position-wise feedforward layer' because the same network (along with its weights and biases) is used for all the input tokens.

Equations (13), (14) and (15) describe the feedforward network used in the GPT-2 model. It involves two linear

transformations along with a nonlinear activation function in between them. Transformer architecture uses rectified linear unit (ReLU) non-linear function. Though the input $\mathbf{X}$ and output $\mathbf{Y}$ dimensionality of feedforward layer remain $\mathbb{R}^N$, the hidden layer Y_1 dimensionality is $\mathbb{R}^{N1}$. Note that $\mathbf{W}_1 \in \mathbb{R}^{N\times N_1}, \underline{b}_1 \in \mathbb{R}^{N_1}, \mathbf{W}_2 \in \mathbb{R}^{N_1\times N}$ and $\underline{b}_2 \in \mathbb{R}^N$.

$$\mathbf{V}_1 = \mathbf{X}^T\mathbf{W}_1 + \underline{b}_1. \tag{13}$$

$$\mathbf{Y}_1 = \max(0, \mathbf{V}_1). \tag{14}$$

$$\mathbf{Y} = \text{ffn}(\mathbf{X}) = \mathbf{Y}_1^T\mathbf{W}_2 + \underline{b}_2. \tag{15}$$

Note that the GPT-2 stack has 12 layers. Let $\{X_l\}_{l=1}^L$ and $\{Y_l\}_{l=1}^L$ represent the input and output of the layer $l \in [1, L = 12]$. The output of the attention layer post 'add and layer normalization' is fed as input to the position-wise feedforward layer. This data processing continues sequentially over all the layers in the GPT-2 stack. As the output of each layer is fed as input to the next layer, $X_l = Y_{l-1}$ for $i \in [2, 12]$. The output of final layer Y_{12} is used to compute the cross-entropy loss L. Using backpropagation, loss L is propagated back to adjust each layer's weights and biases.

E. Residual connection and layer normalization

Vanishing gradient problem is frequently encountered in deep neural networks with multiple layers. In GPT-2 architecture, there are two kinds of layers: attention layer and position-wise feedforward layer. Consider a general layer which operates a function F on its input X to produce output Y. Hence in case of GPT-2, function F can either be attention mechanism or the position-wise feedforward network.

Consider the case in which the local gradient $\frac{\delta F(X)}{\delta X}$ of the layer is zero. Given $\frac{\delta L}{\delta Y}$, $\frac{\delta L}{\delta X}$ is computed, where L is the loss value at the final layer. Since the local gradient is zero, $\frac{\delta L}{\delta X}$ would also be zero. This is because of the multiplicative nature as described in (16).

$$\begin{aligned} Y &= F(X) \\ \frac{\delta L}{\delta X} &= \frac{\delta F(X)}{\delta X}\frac{\delta L}{\delta Y} \\ \frac{\delta F(X)}{\delta X} = 0 &\implies \frac{\delta L}{\delta X} = 0 \end{aligned} \tag{16}$$

However, the vanishing gradient problem is resolved if a residual connection is employed [10]. Residual connection adds the input and output of each sublayer to produce the final output. The incoming gradient bypasses the layer with zero local gradients and can efficiently propagate down streams described in (17). Fig. 3 illustrate the advantage of employing residual connection.

$$\begin{aligned} Y &= F(X) + X \\ \frac{\delta L}{\delta X} &= \frac{\delta F(X)}{\delta X}\frac{\delta L}{\delta Y} + 1 \\ \frac{\delta F(X)}{\delta X} = 0 &\implies \frac{\delta L}{\delta X} = \frac{\delta L}{\delta Y} \end{aligned} \tag{17}$$

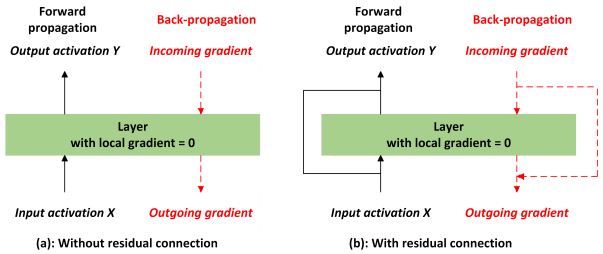

Fig. 3. Illustration of residual connection. In (a), absence of residual connection results in zero gradient for all the layers downstream. In (b), presence of residual connection help to bypass the layer with zero local gradient and propagate gradient to layers downstream.

Layer normalization [11] is used to normalize the output of each layer so that they have zero mean and unit standard deviation. Performing layer normalization helps to counteract exploding activation problems. Since GPT-2 processes all the input tokens $\{x_1, x_2, ..., x_M\}$ where x_i belongs to $\mathbb{R}^N$, the input to the layer normalization is matrix of size $M \times N$. The mean and standard deviation of the entire matrix is computed, and normalization is performed.

The resultant of layer normalization $\mathbf{Y}$ has zero mean and unit variance. This resultant is used as input for next sublayer in the GPT-2 model. For example, layer normalization is performed after the attention layer and the resultant $\mathbf{Y}$ is used as input for the feedforward layer.

Therefore the presence of residual connections and layer normalization after each sublayer in the GPT-2 stack solves the vanishing and exploding gradient problems, respectively.

III. Software Implementation

GPT-2 is implemented on Python3.9 programming language using pytorch library. Since the aim of the paper is to develop hardware architecture for inference engine, the HuggingFace trained model is used to initialize the GPT-2 model weights and biases. Then, the GPT-2 model is trained on the WikiText-2 dataset [12] to test the interconnections between all the sublayers.

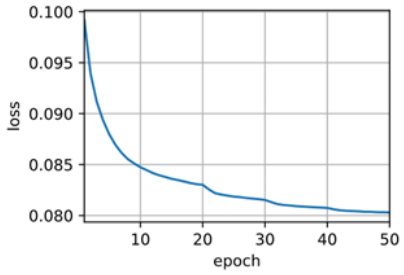

Fig. 4. The plot of cross-entropy loss function during the training of the GPT-2 model. Loss decreases abruptly at epoch 20 because the learning rate is decreased by a factor of 2 after epoch 20.

The masked cross-entropy (CE) function is used to compute loss, and network parameters are updated using Adam optimizer. The learning rate at the start of training is set to

0.005 and remains the same for 20 epochs, after which it is decreased by a factor of 2 for every ten epochs. Fig. 4 shows the plot of cross-entropy loss function.

Fig. 5 shows the inference phase of the GPT-2 model. An input token sequence is fed to the GPT-2 model and the model successively generates 50 additional tokens.

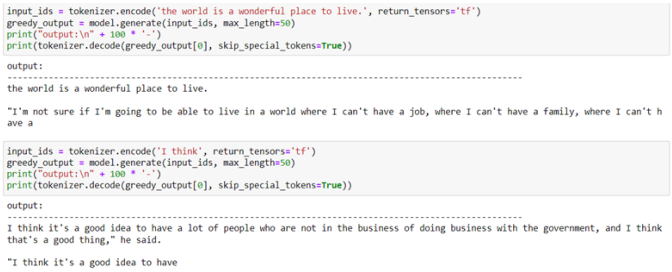

Fig. 5. Inference phase of the GPT-2 model. GPT-2 model generates 50 additional tokens based on the input sequence of tokens fed into the model.

IV. Hardware implementation

A. High level synthesis

High-level synthesis (HLS) is a recent paradigm in digital design which works at a higher abstraction level than HDL languages. In HLS, the designer can code an algorithm in C/C++ language, and the HLS tool synthesizes a digital circuit for this algorithm. HLS tools support programming languages like C/C++ rather than other high-level programming languages like python because C/C++ uses low-level constructs easily mapped to hardware.

Key required for HLS synthesis is that the hardware implementation must be clearly defined in C/C++ language. For example, factorial using recursive function call is not supported in HLS whereas 'For loops' which fixed number of iterations is supported. Compile directives pragmas allows designer to achieve tradeoffs between latency and resource utilization.

B. General matrix multiplication unit

In the GPT-2 model, the attention and feedforward layers require multiple matrix multiplication operations. Hence, creating an optimized matrix multiplication unit, also called a general matrix multiplication (GEMM) unit, is crucial for achieving an efficient hardware implementation of the GPT-2 model.

Algorithm 1 shows the naïve implementation of the GEMM unit. In this implementation, only one multiple-and-accumulate (MAC) unit is used. Hence the latency is in the order $O(N^3)$ because the GEMM unit requires N^3 MAC operations. We can direct the compiler to use additional MAC units to decrease the latency. However, in this case, additional MAC units will only result in a marginal decrease in the latency. This is because these additional MAC units cannot be used as the innermost loop has a loop-carried dependency. Loop carried dependency means that the variable 'acc' value at a given iteration k = K depends upon all the previous iterations.

Algorithm 1 Naïve implementation of GEMM unit

Ensure: $\mathbf{C}_{N\times N} = \mathbf{A}_{N\times N}\ \mathbf{B}_{N\times N}$
for i = 1 to N **do**
  for j = 1 to N **do**
    acc ← 0
    for k = 1 to N **do**
      {//Parallelism cannot achieve due to loop carried dependency}
      acc += $\mathbf{A}_{i,k} \cdot \mathbf{B}_{k,j}$
    end for
    $\mathbf{C}_{i,j}$ ← acc
  end for
end for

Algorithm 2 Optimized implementation of GEMM unit

Ensure: $\mathbf{C}_{N\times N} = \mathbf{A}_{N\times N}\ \mathbf{B}_{N\times N}$
for i = 1 to N **do**
  for k = 1 to N **do**
    {//to minimize read operations}
    tempA ← $\mathbf{A}_{i,k}$
    acc ← 0
    {//Interchange loop to achieve parallelism}
    for j = 1 to N **do**
      #pragma HLS pipeline
      if k is 0 **then**
        accPrv ← 0
      else
        accPrv ← acc[j]
      end if
      {//No loop carried dependency}
      acc[j] = accPrv + tempA · $\mathbf{B}_{k,j}$
    end for
  end for
  for j = 1 to N **do**
    $\mathbf{C}_{i,j}$ ← acc[j]
  end for
end for

One of the key drawbacks in the naïve implementation is that loop carried dependency of the accumulator variable. However, using multiple accumulators requires loop interchange, as illustrated in Algorithm 2. Algorithm 2 instructs the HLS compiler to use the 'PIPELINE' directive on the innermost loop. Hence, the HLS compiler uses multiple MAC units. In this implementation [7], using multiple MAC units decreases the latency because there are multiple instances of accumulated variable acc. That is, each iteration j = J has its own MAC unit. Also, it can be observed from Algorithm 2 that the number of read operations of the memory unit containing input A is reduced because the same value of input A is used multiple times.

Table II shows the comparison between naïve and optimized implementation of the GEMM unit obtained from Xilinx HLS tool. We observe 25x decrease in the latency of the optimized GEMM unit compared to the naïve GEMM unit. The drastic

reduction is latency is achieved because inner-most loop 'J' has no recurrence and thus it's computation can be pipelined. We note that the optimized GEMM unit uses 128 DSP units, which is consistent with our analysis that each iteration j = J has its own MAC unit. Note that the MAC unit is reported as a DSP unit by the Xilinx HLS tool.

TABLE II
LATENCY AND RESOURCE UTILIZATION COMPARISON BETWEEN GEMM UNITS PERFORMING 128×128 MATRIX MULTIPLICATION.

Metric	Naïve GEMM	Optimized GEMM
Latency (clock cyles)	2147591	83210
BRAM	74	194
DSP	1	128
FlipFlop	517	13707
Lookup tables	1093	15490
Clock period (ns)	6.046	5.853

This drastic reduction in latency provided us with enough confidence to pursue HLS path for hardware architecture of GPT-2 stack. The estimated clock frequency of the optimized GEMM unit is higher than the Naïve GEMM unit. This might be due to the reduced number of memory accesses in the optimized GEMM unit.

C. Softmax unit and Add-Norm unit

Equation (7) provides the formulation for softmax computations. This formulation works well in software implementation because python software uses floating-point representation during exponential operation. Hence, it is necessary to use fixed-point representation for exponential operation.

However, due to the intrinsic nature of the exponential operation, overflow is a common issue faced while implementing exponential operation using fixed-point representation. LogSumExp is a common technique employed to overcome this issue [13]. Equation (18) shows that computation of maximum input value is required before performing the exponential operation. The overflow scenario does not arise during computation since the input to exponential operation is always an integer less than or equal to zero.

$$\mathrm{y_i} = \frac{\mathrm{e}^{\mathrm{x_i}-\mathrm{x_{max}}}}{\sum_{\mathrm{j}=1}^{\mathrm{M}} \mathrm{e}^{\mathrm{x_j}-\mathrm{x_{max}}}} \;\; \forall \;\; \mathrm{i} \in [1, \mathrm{M}] \;\; \text{and} \;\; \mathrm{x_{max}} = \max_{\mathrm{i}}(\mathrm{x_i}). \tag{18}$$

Note that the output of softmax computation is the probabilistic value whose range is limited to [0,1]. Since fixed-point representation is used, it is necessary to scale the output of softmax computation. Otherwise, these output values will be either 0 or 1 and thus would not be able to fully utilize the available range of -128 to +127 provided by 8-bit representation. Therefore, the scale factor of 127 is used during the computation of the softmax function. Algorithm 3 describes the key points of the C++ code used during the HLS synthesis of the softmax unit.

Another fundamental block in the GPT-2 model is the Add-Norm unit. This unit involves the residual connection and layer normalization sub-unit. In software, the layer normalization sub-unit scales the data to have zero mean and unit variance. However, in hardware implementation, it is necessary to scale the output of the layer normalization unit to utilize the available range of -128 to +127 provided by 8-bit representation. The scaling factor of 32 rather than 127 is utilized for HLS synthesis because a small yet considerable number of layer normalization outputs lie outside the range of unit standard deviation (σ) during software implementation. However, a minuscule number of layer normalization outputs lie outside the range of 4(σ). Hence, scaling factor of 32 is chosen.

Algorithm 3 Softmax unit operating row-wise on the input matrix

Ensure: $\mathbf{smA}_{N\times N}$ = softmax($\mathbf{A}_{N\times N}$)
  scale_factor ← 128
  for i = 1 to N **do**
    row_max[i] ← $\mathbf{A}_{i,0}$
    row_sum[i] ← 0
  end for
  {//Compute Row maximum value}
  for j = 2 to N **do**
    {//Interchange loop to achieve parallelism}
    for i = 1 to N **do**
      if row_max[i] < $\mathbf{A}_{i,j}$ **then**
        row_max[i] ← $\mathbf{A}_{i,j}$
      end if
    end for
  end for
  for j = 1 to N **do**
    {//Interchange loop to achieve parallelism}
    for i = 1 to N **do**
      {//Use LogSumExp trick to avoid overflow}
      $\mathbf{expA}_{i,j}$ ← exp($\mathbf{A}_{i,j}$ - row_max[i])
      row_sum[i] += $\mathbf{expA}_{i,j}$
    end for
  end for
  for j = 1 to N **do**
    for i = 1 to N **do**
      {//Scale the resultant to utilize full range of 8bit}
      $\mathbf{smA}_{i,j} \leftarrow \frac{\mathbf{expA}_{i,j} \times scale\_factor}{row\_sum[i]}$
    end for
  end for

D. Hardware architecture of GPT-2 model

The entire GPT-2 architecture is coded in synthesizable C++ code utilizing the following functions: GEMM unit, softmax unit, and Add-Norm unit. GPT-2 architecture demands significant memory and processing units. This is because of multiple linear layers in each sublayer, requiring $O(MN^2)$ multiply-accumulate operations where N is the model dimensionality and M is the input sequence length. Due to limited DSP units available on FPGA and limited memory bandwidth to access the weights and biases stored in BRAM, model parameters are reduced as follows:

- Datatype: 8-bit fixed point representation
- Model Dimensionality = 128
- Input sequence length (Batch size) = 64
- Feedforward network's hidden layer dimensionality = 64
- Number of layers = 1
- Number of heads in attention layer = 1

The reason for choosing a single layer is that all the layers are identical, and analysis obtained for a single layer can be scaled to multiple layers as required.

The resultant is a streaming architecture in which each module in the Fig 6 is optimized individually and connected in a pipelined structure for decreased latency and increased throughput. A single decoder layer requires 8 GEMM units, as described in Table III. Similarly, a single decoder layer also requires a softmax unit and two Add-Norm units, as described in Table IV.

TABLE III
INSTANCES OF GEMM UNITS IN THE GPT-2 MODEL.

Function usage	Input 1 matrix	Input 2 matrix	Output matrix
ATTN - query	$\mathbf{X}_{64\times128}$	$\mathbf{Wq}_{128\times128}$	$\mathbf{Q}_{64\times128}$
ATTN - key	$\mathbf{X}_{64\times128}$	$\mathbf{Wk}_{128\times128}$	$\mathbf{K}_{64\times128}$
ATTN- value	$\mathbf{X}_{64\times128}$	$\mathbf{Wv}_{128\times128}$	$\mathbf{V}_{64\times128}$
Softmax score	$\mathbf{Q}_{64\times128}$	$\mathbf{K}^T_{128\times64}$	$\mathbf{Score}_{64\times64}$
Weighted keys	$\mathbf{Score}_{64\times64}$	$\mathbf{V}_{64\times128}$	$\mathbf{S}_{64\times128}$
Linear layer	$\mathbf{S}_{64\times128}$	$\mathbf{Wl}_{128\times128}$	$\mathbf{C}_{64\times128}$
FFN - sublayer 1	$\mathbf{N}_{64\times128}$	$\mathbf{W1}_{128\times64}$	$\mathbf{V}_{64\times64}$
FFN - sublayer 2	$\mathbf{V2}_{64\times128}$	$\mathbf{W2}_{64\times128}$	$\mathbf{V}_{64\times128}$

TABLE IV
INSTANCES OF ADD-NORM UNITS IN THE GPT-2 MODEL.

Function usage	Input 1 matrix	Input 2 matrix	Output matrix
Add-Norm ATTN	$\mathbf{X}_{64\times128}$	$\mathbf{C}_{64\times128}$	$\mathbf{N}_{64\times128}$
Add-Norm FFN	$\mathbf{N}_{64\times128}$	$\mathbf{V3}_{64\times128}$	$\mathbf{Y}_{64\times128}$

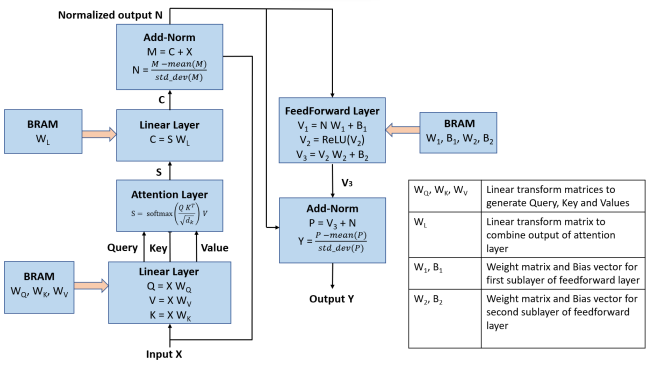

Fig. 6. Implementation of decoder layer of the GPT-2. Computation of query(**Q**), key(**K**) and value(**V**) is performed by linear layer. Attention layer computes the weighted sum of values to generate scores matrix S, Residual connection and layer normalization is performed after each sublayer namely, attention layer and feedforward layer.

V. DESIGN IMPLEMENTATION RESULTS

Key components of the GPT-2 model: GEMM unit, softmax unit, and add-norm unit, have been synthesized and tested separately on Xilinx Vitis HLS 2022.1 tool. Then top-level GPT-2 is synthesized using multiple instances of the GEMM unit, softmax unit, and add-norm unit. The synthesized design is targeted for the Virtex-7 VC707 evaluation board. Virtex-7 VC707 evaluation board is used as the target for synthesized design because it possesses sufficient resources to accommodate a single-layer of GPT-2 and also because of minimum overhead required to port the synthesized design from Xilinx HLS tool to Virtex-7 VC707 board.

The input to the RTL model is a sequence of token word embeddings. The synthesized GPT-2 model processes these word-embedding vectors by adding positional-embedding vectors to them. A single layer of transformer decoder layer is present in the GPT-2 model. Attention computation, followed by feedforward computations, is performed, and the GPT-2 model outputs a sequence of output word-embedding vectors.

A. Test setup and C-RTL cosimulation

For testing the GPT-2 model, C-RTL co-simulation is performed using Xilinx Vitis HLS 2022.1 tool. In the initial stage, the GPT-2 model functionality is verified in C++. Then Verilog RTL is synthesized for the GPT-2 model. The resultant Verilog GPT-2 model is tested using C-RTL co-simulation. In C-RTL co-simulation, the Xilinx Vitis HLS 2022.1 tool automates the generation of RTL inputs based on the C++ testbench, passes these inputs to the synthesized GPT-2 model, and collects the output values. These output values are compared to the golden output obtained in the software. C-RTL co-simulation is used to validate if the synthesized GPT-2 model works as expected. The successful C-RTL co-simulation also provides the estimated latency of the inference engine.

B. Latency and resource utilization

Key resource metrics from successful C-RTL co-simulation for various blocks of GPT-2 are summarized in Table V. The maximum operating frequency of the synthesized model is 142.44 MHz.

TABLE V
LATENCY AND RESOURCE UTILIZATION OF THE PROPOSED GPT-2 INFERENCE ENGINE FROM HLS TOOL

Module	Latency (cycles)	BRAM	DSP	FF	LUT
ATTN layer	75773	646	960	64267	70986
Add-Norm unit	25206	19	6	3485	4644
FFN sublayer 1	16897	2	0	4222	3805
FFN sublayer 2	21121	3	128	13547	16426
Entire GPT model	**163705**	**782**	**1094**	**85672**	**96855**
Total available resource on Virtex-7	NA	1030	2800	607200	303600

For benchmarking purposes in terms of resource utilization, we consider [5] as reference which use transformer for next word prediction. Since transformer has both encoder and

decoder modules, only the decoder's resource utilization is taken into consideration for comparison as the decoder is used in the inference.

Table VI shows the comparison of resource utilization. The GPT-2 architecture synthesized in this paper contains a single layer of the GPT-2 model of model dimensionality 128 and batch size 64. However, the transformer-decoder layer presented in paper [5] has hidden layer dimension 200 and batch size 32. Since the model dimensionality is different in both the cases, resource utilization is scaled for comparison purpose. As the latency of the GPT-2 models is of the order $O(M \cdot N^2)$ where M is input sequence length (batch size) and N is model dimensionality, scaling factor of $\left(\frac{64 \cdot 128^2}{32 \cdot 200^2}\right) = 0.8192$ is chosen.

TABLE VI
COMPARISON OF RESOURCE UTILIZATION AND LATENCY.

Metric	Single layer GPT-2	Single decoder layer [5]	Scaled decoder layer
Frequency	142.45 MHz	NA	NA
Latency	1.637 ms	3.456 ms	2.831 ms
DSP	1094	1318	1079
FlipFlop	85.6k	98.8k	80.93k
LUT	96.8k	83.3k	68.23k

Table VI shows that the GPT-2 architecture proposed in this paper achieved 1.73x speedup i.e. 57.8 % reduction in latency compared to previously report architecture [5]. This speedup is achieved at the cost of slight increase in resource utilization. This is a good trade-off because we are only using 39.07% DSPs, 14.1% FFs and 31.9% LUTs available on Virtex-7 VC707 board. As noted in the previous section, the latency of the GPT-2 model is of the order $O(M \cdot N^2)$, hence the input sequence length would have a quadratic effect on the model's latency and performance.

One of the limitations of the proposed method is the scope for detailed analysis of the data movement across the various layers of GPT-2 stacks. In the present work, we have implemented a single layer of GPT-2 stack. However, recent and more powerful versions of GPT-2 and its variants have multiple layers. [4] illustrates the architecture, where an increase in the number of layers increases the zero-shot results on many datasets. More detailed analysis of data movement across different layers and various optimizations like use of dedicated memory controller would help to mitigate memory bottleneck. This would also decrease the power consumption and improve energy efficiency. In real-world applications, the inference engine is used as an accelerator to offload the work of CPU and typically, the data consumed by the accelerator is stored in DRAM, external to the accelerator. Hence, use of dedicated memory controllers to fetch the data from DRAM and stage the data processing to allow maximum data reuse across various layers could be used to resolve bandwidth limitations.

VI. CONCLUSION

Generative pretrained transformer (GPT) is a recent NLP neural network based on the attention mechanism. Due to the ability of GPT-2 to concurrently process all its inputs, its training speed has far exceeded traditional neural networks like RNNs and LSTMs. Hardware implementation of the GPT-2 model is achieved using high-level synthesis. Generalized matrix multiplication (GEMM) unit, softmax unit, and Add-Norm unit are the critical components of the GPT-2 model. These components are individually designed and optimized for latency and resource utilization. Model dimensionality is reduced due to memory bandwidth limitations. The inference engine proposed in this paper achieves a significant reduction in latency compared to previously reported transformer-based inference engines. Model dimensionality is the crucial factor that affects that scalability due to its quadratic relationship to the computation cost. For ASIC implementation, these memory bandwidth issues can be resolved using a dedicated memory controller instead of BRAM.

REFERENCES

[1] Sepp Hochreiter and Jürgen Schmidhuber,“ Long short-term memory. Neural computation,” 9(8): pp 1735–1780, 1997.

[2] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez A, N, Kaiser Ł, and Polosukhin I, “Attention is all you need,”Advances in Neural Information Processing Systems, pp. 6000–6010, 2017.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova,“BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,”arXiv preprint arXiv:1810.04805, 2018.

[4] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever, “Language models are unsupervised multitask learners,”, 2019. [Online]. Available: https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf. [Accessed: 29-Jan-2023].

[5] Peng et al,“Accelerating transformer-based deep learning models on fpgas using column balanced block pruning,” 22nd International Symposium on Quality Electronic Design (ISQED), pp. 142–148, 2021.

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

[7] Kastner R, J. Matai, and S. Neuendorffer,“Parallel Programming for FPGAs,”ArXiv e-prints, May 2018.

[8] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz,“Building a large annotated corpus of English: The Penn Treebank,”Computational Linguistics, 19(2):313–330, 1993.

[9] Tomas Mikolov, Kai Chen, Greg Corrado and Jeffrey Dean,“Efficient Estimation of Word Representations in Vector Space,”arXiv preprint arXiv:1301.3781, 2013

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun,“Deep residual learning for image recognition,”Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778, 2016.

[11] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton,“ Layer normalization,” arXiv preprint arXiv:1607.06450, 2016.

[12] Merity S, Xiong C, Bradbury J, and Socher R,“Pointer sentinel mixture models,” 5th International Conference on Learning Representations ICLR, pp. 770–778, 2016.

[13] B. Yuan, “Efficient hardware architecture of softmax layer in deep neural network,” 2016 29th IEEE International System-on-Chip Conference (SOCC), Seattle, WA, USA, 2016, pp. 323-326, doi: 10.1109/SOCC.2016.7905501.