When the Edge Meets Transformers: Distributed Inference with Transformer Models

Chenghao Hu, Baochun Li
ch.hu@mail.utoronto.ca, bli@ece.toronto.edu
Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Transformer models achieved significant breakthroughs in a wide variety of applications, yet their exorbitant computation costs pose significant challenges when it comes to deploying these models for inference, especially on resource-constrained edge devices. In this paper, we introduce the concept of cross-device distributed inference to transformer models, which accelerates the speed of inference by distributing its workload among multiple edge devices. Unlike previous work designed for multi-GPU environments, the challenge of distributing inference workload on edge devices includes not only limited computation power, but also low bandwidth connections to exchange intermediate results.**

To address these challenges, we propose *Voltage*, a distributed inference system tailored for edge devices. By exploiting the inherent parallelizability of the input sequence, *Voltage* partitions the transformer inference workload based on positions to accelerate the inference speed. We also analyze the relationship between the partition settings and the computation complexity, which allows *Voltage* to adaptively select the most efficient computation scheme. To demonstrate its effectiveness and generalizability, the performance of *Voltage* has been evaluated in the context of well-known transformer models, and in a variety of experimental settings. Our results show that *Voltage* significantly outperforms tensor parallelism by reducing the communication size by 4×, thereby accelerate the inference speed by up to 32.2% compared with single device deployment.

***Index Terms*—Distributed system, distributed inference, transformer models**

I. Introduction

The emergence of the transformer architecture [1] has "transformed" the community of natural language processing (NLP), and enabled a collection of language models like BERT [2] and GPT [3] with record-breaking performance. Recent advances also introduce the transformer into a wide range of research areas including computer vision [4], which is previously dominated by convolutional neural networks. While researchers started to embrace transformers and proposed more and more complicated models to pursue the best possible performance, the accompanied computation cost became a potentially thorny problem: it not only increases the difficulty to train the model, but also poses a significant challenge to model inference, especially when these models are to be deployed on resource-constrained edge devices [5].

Compared with cloud servers, the computation capacity of edge devices is typically severely limited, making it extremely challenging to finish the inference task of deep learning models within the latency budget. Several existing research efforts were trying to deploy transformer models on the edge devices, *e.g.*, by reducing the computation cost by pruning the original model [6], or distilling large models into more compact ones [7], [8] that can be fitted into the edge devices. Though these methods managed to reduce the inference latency, their benefits may come at a price of accuracy loss since the model's architecture and weights have been changed.

To keep the models intact and guarantee the quality of inference, the new paradigm of *distributed inference* has been proposed, which distributes the workload across multiple edge devices to accelerate the inference process [9]. For example, by utilizing the partial receptive field property of convolution kernels, the input feature map of a convolution layer can be partitioned and assigned to different devices for computation. Based on such an insight, existing works [10], [11] successfully distributed and accelerated the inference of convolutional neural networks.

For transformer models, some recent works have also attempted to distribute the inference workload by borrowing parallelization techniques from model training. For instances, tensor parallelism [12], [13] and pipeline parallelism [14] have been adopted for inference applications. However, tensor parallelism incurs significant communication overhead and is therefore typically used in single-device, multi-GPU environments where intermediate results can be rapidly exchanged over high-speed internal connections.

On the other hand, pipeline parallelism builds a pipeline by assigning different layers to different devices. It requires a sufficiently large batch of input data to fully utilize the pipeline capacity. However, in edge environments like cellphones or laptops, the inference system does not have to serve many concurrent users. Inference requests typically arrive in a sporadic manner with small batch sizes, often only a single input. In these cases, the pipeline is severely underutilized.

While existing parallelisms have proven effective for enabling distributed training, we argue that they are not well-suited for distributed inference, especially within edge environments. In this paper, we design and build *Voltage*, a new system to distribute the inference workload of transformer models across resource-constrained devices. Our key insight stems from the transformer's position-wise nature, that most operations of the transformer, except for the self-attention mechanism, are applied independently to each position. *Voltage* partitions the inference workload so layer outputs at different positions can be computed in parallel across devices, thereby accelerating the overall execution speed.

For the self-attention mechanism, we observe that the computation orders significantly impact the computation complexity. One of the highlights of this work is developing techniques to rearrange self-attention calculations to generate partial output more efficiently in a distributed setting. By thoroughly analyzing the relationship between the computation complexities and the layer settings, including the partition size, input size, and feature dimensions, we develop an adaptive mechanism that selects the most efficient self-attention computation order. We formally prove this scheme chooses the optimal strategy, delivering a linear speed-up when distributing a single transformer layer.

Our extensive array of experimental evaluations conducted on well-known transformer models verifies the effectiveness of our design and analysis. Our results show that *Voltage* achieves the lowest computation complexity and the best scalability for the distributed execution. Compared with tensor parallelism, *Voltage* reduces the communication size by 4×, thereby reduces the overall inference latency of transformer models by up to 32.2% compared with single device deployment.

II. Preliminaries

A. Transformer Models

Transformer models are composed of encoders and decoders, which are referred to as transformer layers in this paper since they share similar computation patterns. There are two major components of the transformer layer: a *multi-head self-attention mechanism* and a *position-wise feed-forward network*.

Self-attention is the heart of transformer models. Let $x \in \mathbb{R}^{N\times F}$ denote the input sequence with length N and feature dimensionality F. Self-attention projects input x into three matrices $Q, K, V \in \mathbb{R}^{N\times F_H}$ with attention weights $W_Q, W_K, W_V \in \mathbb{R}^{F\times F_H}$, where F_H is the dimensions of the attention features. Then the self-attention output $\mathrm{Attn}(Q, K, V)$ can be computed by:

$$\begin{aligned} Q &= xW_Q, \\ K &= xW_K, \\ V &= xW_V. \end{aligned} \tag{1}$$

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{F_H}}\right)V.$$

Instead of performing a single self-attention function, the transformer architecture adopts a multi-head self-attention design to allow the model to attend to information from different representation spaces. Let $H > 1$ denote the number of attention heads. Then there will be H different sets of learned attention weights to be applied on the input x independently. The output of these H independent attention heads will be concatenated and projected as follows:

$$\begin{aligned} &\mathrm{MultiHead}(x) = \mathrm{Concat}\left(A^1(x), \ldots, A^H(x)\right)W_O, \\ &\text{where} \quad A^i(x) = \mathrm{Attn}(xW_Q^i, xW_K^i, xW_V^i), \end{aligned} \tag{2}$$

and $W_O \in \mathbb{R}^{HF_H\times F}$ is another weight matrix. In practice, the transformer models typically set $HF_H = F$, such that the computation cost is similar to a single head attention with full feature dimensionality [1].

After the projection, the output $\mathrm{MultiHead}(x)$ has the same shape as the input x, which allows the transformer to apply a residual link to add them together. Lastly, the result is normalized through a layer normalization [15] step like below:

$$\mathrm{LayerNorm}(\mathrm{MultiHead}(x) + x).$$

The multi-head self-attention mechanism is followed by a position-wise feed-forward network, which will be applied to the input at each position separately and identically. Typically this feed-forward network consists of two linear transformations and an activation function (e.g. RELU in [1], and GELU in [2]) in between:

$$\mathrm{FFN}(x) = \mathrm{Act}(xW_1 + b_1)W_2 + b_2.$$

The feed-forward network also adopts the residual connection and layer normalization design to generate the final output of this transformer layer. The procedures mentioned above are summarized as an end-to-end illustration in Fig. 1.

B. Tensor Parallelism

The significant amount of computation complexity of transformer models has motivated many parallel computation paradigms to facilitate the computation efficiency of transformer models, where some of them have been utilized in inference systems. For instances, DeepSpeed [16] and Parallelformer [17] adopt tensor parallelism to split weight tensors of transformer layers across devices (typically GPUs) during inference.

Tensor parallelism takes advantage of the inherent parallel nature of transformer architecture, particularly in the multi-head self attention mechanism. As depicted in Fig. 2, different attention heads are applied to the input data on each device. The attention outputs are then combined via a global All-Reduce operation to produce the final output that incorporates results from all attention heads. Similarly, the feed-forward network weights are split across devices and require another All-Reduce step to gather the outputs and produce the final inferred representation.

In other words, tensor parallelism works by distributing the computation of a large transformer model across multiple devices, at the cost of extra All-Reduce communication steps. This communication overhead is acceptable in environments like multi-GPU servers where high bandwidth connections exist between the GPUs.

However, for distributed inference at the edge, the devices are typically connected by slower connections like Wi-Fi, and Ethernet networks, the communication overhead could dominate the inference latency. If we adopted tensor parallelism to distribute inference across slower connected edge devices, the required global All-Reduce operations would introduce substantial communication and synchronization overhead. This hampers potential computational efficiency gains, especially on resource constrained edge hardware with lower compute

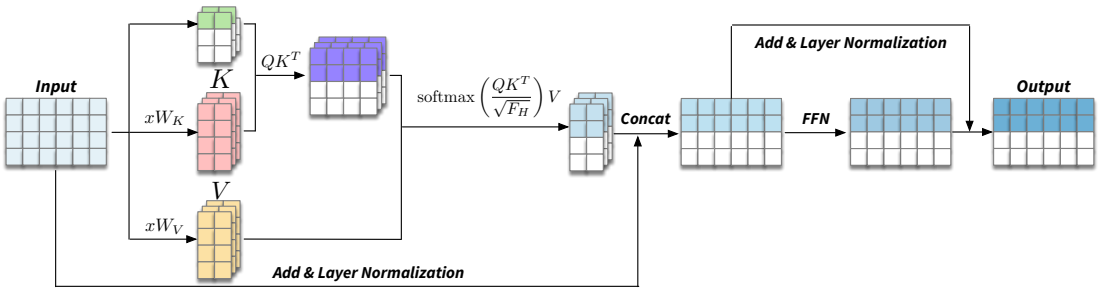


Fig. 1. The computation flow of a transformer layer and an illustration of position-wise layer partition where the elements involved in computation are colored, while irrelevant entries are left blank.

capacity. Therefore, we argue that tensor parallelism is not the optimal solution for distributed inference across networks of edge devices.

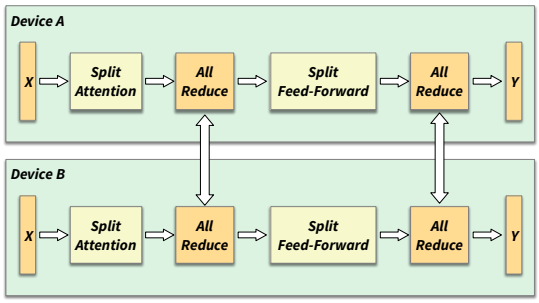


Fig. 2. An illustration of tensor parallelism for a single transformer layer.

III. Position-Wise Layer Partition

To alleviate the communication overhead of tensor parallelism and fully exploit the computation power of multiple devices to reduce inference latency, we introduce a novel position-wise layer partition method for transformer models. The key inspiration comes from the position-wise nature of the transformer. For instances, most of the operations inside the transformer layer, like the feed-forward network and layer normalization, are applied independently to each position. This motivates a position-wise partitioning of the layer, where each device computes outputs for a subset of positions.

As an example, assume we intend to compute the first two rows of the output in Fig. 1, representing the output at the first and second positions. We can still follow the standard computation procedure, but only some parts of the weight matrices are needed. The elements involved in the computation are highlighted with different colors, we can see that most matrix entries are unused, as they are irrelevant for producing the desired outputs.

Formally, let x_p denote the input partition for the selected positions. To obtain the desired output partitions, we simply replace the full query matrix Q with the sub-matrix $Q_p = x_p W_Q$ during attention. That is, we modify Eq. (2) to:

$$A_p^i(x) = \mathrm{Attn}(x_p W_Q^i, xW_K^i, xW_V^i).$$

Concatenating the output partitions from different heads then gives the multi-head self-attention outputs for those positions. As remaining transformer operations are position-wise, this output partition can flow through them to obtain the final layer output at desired positions. In this way, we can partition the workload of a single transformer layer into pieces and assign them to different devices for computation.

In tensor parallelism, the model weights are partitioned and processed by different devices, such that the computation on each device has no overlapping, *i.e.*, the workload is perfectly partitioned among devices. But in our method, though we only wish to obtain a part of the output, we still need to compute the whole K, V matrices (fully colored in Fig. 1) no matter how small the partition is, which means the computation is redundantly executed on each device. To avoid such a situation, we can optimize the computation process of self-attention to partition the workload more efficiently, which will be discussed in the next section.

IV. Revisiting Self-Attention

In this section, we measure the computation complexity of the naive computation method to understand why it fails to partition the self-attention function efficiently. Then we show that the computation process can be optimized according to the input and layer settings. Based on this observation, we present a partitioned transformer layer which achieves linear acceleration with respect to the partition numbers.

A. Computation Complexity Analysis

Since the computation cost of the multi-head self-attention mechanism is exactly the sum of the cost of every attention head, we focus on the analysis of a single self-attention function for simplicity. Follow the notation in Section II, let $x_p \in \mathbb{R}^{P \times F}$ denote the input partition corresponding to the positions of output partition $A_p(x)$. According to Eq. (1), the naive computation method to compute $A_p(x)$ can be condensed into the following equation:

$$A_p(x) = \mathrm{softmax}\left(\frac{(x_p W_Q)(xW_K)^T}{\sqrt{F_H}}\right)(xW_V) \qquad (3)$$

where the parentheses indicate the order of the computation, *i.e.*, compute Q, K, V matrices in advance.

The matrix multiplication is the main operation of the self-attention, therefore we use the number of the floating operations, denoted as $\Gamma(\cdot)$, to measure the computation complexity. For example, with $x \in \mathbb{R}^{N\times F}$, and $W_Q \in \mathbb{R}^{F\times F_H}$, the computation complexity of computing $Q = xW_Q$ can be measured by

$$\Gamma(xW_Q) = N \times F \times F_H = NFF_H.$$

Follow this rule, we give the computation complexity of Eq. (3) by the following theorem.

Theorem 1. *Given input $x \in \mathbb{R}^{N\times F}$ and input partition $x_p \in \mathbb{R}^{P\times F}$ where $P = \frac{N}{K}$, attention weights $W_Q, W_K, W_V \in \mathbb{R}^{F\times F_H}$. The computation complexity of Eq.* (3) *is*

$$\begin{aligned}\Gamma(\text{Eqn.3}) &= \frac{NFF_H + 2N^2F_H}{K} + 2NFF_H + O\left(\frac{N}{K}F_H\right)\\ &= O\left(\frac{1}{K}\right) + 2NFF_H. \end{aligned} \tag{4}$$

Proof. The computation of Eq. (3) can be separated into the following steps:

1) Compute Q_p, K, V
2) Compute Q_pK^T
3) Compute $\text{softmax}(\frac{Q_pK^T}{\sqrt{F_H}})$
4) Compute $\text{softmax}(\frac{Q_pK^T}{\sqrt{F_H}})V$

And the computation complexity of Eq. (3) will be the sum of the above steps. For step 1, the computation complexity is

$$\begin{aligned}\Gamma(\text{step1}) &= \Gamma(x_pW_Q) + \Gamma(xW_K) + \Gamma(xW_V)\\ &= PFF_H + NFF_H + NFF_H\end{aligned}$$

since $x_P \in \mathbb{R}^{P\times F}, x \in \mathbb{R}^{N\times F}$, and $W_Q, W_K, W_V \in \mathrm{R}^{F\times F_H}$.

Then we have $Q_p \in \mathbb{R}^{P\times F_H}$ and $K \in \mathbb{R}^{N\times F_H}$, which gives the complexity of step 2

$$\Gamma(\text{step2}) = PF_HN.$$

Step 3 includes a position-wise division applied on $Q_pK^T \in \mathbb{R}^{P\times N}$ and a softmax function. The computation complexities for both of them are linear with the number of elements. Therefore the computation complexity of step 3 can be given by

$$\Gamma(\text{step3}) = O(PN).$$

For the last step, we have $\text{softmax}(\frac{Q_pK^T}{\sqrt{F_H}}) \in \mathbb{R}^{P\times N}$, and $V \in \mathbb{R}^{N\times F_H}$. Therefore

$$\Gamma(\text{step3}) = PNF_H.$$

Combining all the complexities of these four steps, we have

$$\Gamma(Eq.\ (3)) = PFF_H + 2PNF_H + 2NFF_H + O(PN).$$

Replacing P by $\frac{N}{K}$ yields the result and finishes the proof. □

Due to the existence of the constant term $2NFF_H$, the naive computation method fails to achieve linear acceleration, which means no matter how small the partition is or how many available devices we have, the time spent on computing K, V matrices remains the same and possibly becomes the bottleneck. Thereby the naive computation method cannot efficiently distribute the workload of the self-attention function.

B. Computation Order Matters

By observing Eq. (3), we have a simple question: do we really have to compute K, V in advance? For example, if we define $S \in \mathbb{R}^{P\times N}$ by

$$S \triangleq \text{softmax}\left(\frac{(x_pW_Q)(xW_K)^T}{\sqrt{F_H}}\right), \tag{5}$$

then we have the following two equivalent ways to compute $A_p(x)$ with different computation complexity:

$$\begin{aligned}\Gamma\left(S(xW_v)\right) &= PNF_H + NFF_H,\\ \Gamma\left((Sx)W_v\right) &= PNF + PFF_H.\end{aligned} \tag{6}$$

By changing the order of the matrix multiplication, we don't have to compute V in advance and leave W_V until the last, which could be better in computation complexity when certain conditions apply.

Similarly, it's unnecessary to compute $K = xW_k$ in advance to obtain Q_pK^T. We can expand the term by

$$Q_pK^T = (x_pW_Q)(xW_K)^T = x_pW_QW_K^Tx. \tag{7}$$

The above calculation involves four matrices and can be computed in 5 different orders, where computing Q, K in advance is only one of them. Combined with Eq. (5), there will be 10 different orders in total to obtain $A_p(x)$. We can easily enumerate all the possibilities or use dynamic programming to find the optimal way to multiply these matrices given different settings. But this adds extra computation overhead to each layer which is not acceptable in a possibly real-time system where the quality of service is measured by the latency.

Fortunately, in the context of transformer models, the feature size and the shape of the attention weights are fixed for a given layer, which left the input and partition size the only two variables. We introduce the following theorem to reveal the relationship between the optimal computation order and the input settings.

Theorem 2. *Given input $x \in \mathbb{R}^{N\times F}$, input partition $x_p \in \mathbb{R}^{P\times F}$, attention weights $W_Q, W_K, W_V \in \mathbb{R}^{F\times F_H}$, and $F = HF_H$ where H is the number of attention heads,*

1) when $\frac{1}{P} - \frac{1}{N} \le \frac{F-F_H}{FF_H}$, Eq. (3) *achieves the lowest computation complexity;*
2) when $\frac{1}{P} - \frac{1}{N} > \frac{F-F_H}{FF_H}$, the following equation achieves the lowest computation complexity

$$A_p(x) = \left(\text{softmax}\left(\frac{((x_pW_Q)W_K^T)x^T}{\sqrt{F_H}}\right)x\right)W_V. \tag{8}$$

Proof. To compute $A_p(x)$, we have to go through the following two steps sequentially.

1) Compute $S = \text{softmax}\left(\frac{x_pW_QW_K^Tx^T}{\sqrt{F_H}}\right)$
2) Compute $A_P(x) = SxW_V$

We start with the second step, which only involves three matrices. There are only two different orders to compute SxW_V: $(Sx)W_V$ and $S(xW_V)$. According to Eq. (6), we can solve the inequality $\Gamma(S(xW_V)) > \Gamma((Sx)W_V)$ to determine the better one:

$$\begin{aligned}
&\Gamma((Sx)W_V) > \Gamma(S(xW_V)) \\
\Rightarrow &PNF_H + NFF_H > PNF + PFF_H \\
\Rightarrow &(N-P)FF_H > PN(F-F_H) \\
\Rightarrow &\frac{N-P}{PN} > \frac{F-F_H}{FF_H} \\
\Rightarrow &\frac{1}{P} - \frac{1}{N} > \frac{F-F_H}{FF_H}.
\end{aligned} \tag{9}$$

Therefore when inequality Eq. (9) holds, the computation complexity of $S(xW_V)$ is greater than $(Sx)W_V$, and vice versa.

Next we focus on the first step to find the best order to compute $x_pW_QW_K^Tx^T$. There are five different orders in total to do the computation, which are $((x_pW_Q)W_K^T)x^T$, $(x_pW_Q)(W_K^Tx^T)$, $(x_p(W_QW_K^T))x^T$, $x_p((W_QW_K^T)x^T)$ and $x_p((W_Q(W_K^Tx^T))$. Notice that $(W_QW_K^T)$ can be computed in advance since they are fixed attention weights. Their respective computation complexities can be computed by

$$\Gamma\left(((x_pW_Q)W_K^T)x^T\right) = 2PFF_H + PFN, \tag{10}$$
$$\Gamma\left((x_pW_Q)(W_K^Tx^T)\right) = PFF_H + NFF_H + PNF_H, \tag{11}$$
$$\Gamma\left((x_p(W_QW_K^T))x^T\right) = PF^2 + PFN, \tag{12}$$
$$\Gamma\left(x_p((W_QW_K^T)x^T)\right) = NF^2 + PFN, \tag{13}$$
$$\Gamma\left(x_p((W_Q(W_K^Tx^T))\right) = 2NFF_H + PNF_H. \tag{14}$$

Firstly, we can eliminate the last candidate $x_p((W_Q(W_K^Tx^T))$. Comparing Eq. (11) and Eq. (14), we have $\Gamma\left((x_pW_Q)(W_K^Tx^T)\right) < \Gamma\left(x_p((W_Q(W_K^Tx^T))\right)$ always holds since $P < N$.

Next we eliminate the third and fourth candidates. In multi-head self-attention, we have $H \geq 2$ and $HF_H = F$. Therefore, the following inequality always holds

$$\begin{aligned}
\Gamma\left((x_p(W_QW_K^T))x^T\right) &= PF^2 + PFN \\
&= HPFF_H + PFN \\
&\geq 2PFF_H + PFN \\
&\geq \Gamma\left(((x_pW_Q)W_K^T)x^T\right),
\end{aligned}$$

which means the computation complexity of $((x_pW_Q)W_K^T)x^T$ can never be worse than $x_p(W_QW_K^T))x^T$. Similarly we can eliminate $x_p((W_QW_K^T)x^T)$ when compared with $x_p(W_QW_K^T))x^T$.

Now we can see that within the context of a multi-head self-attention mechanism. There are only two available candidate computation order to achieve the lowest computation complexity. Solving the following inequality:

$$\begin{aligned}
&\Gamma\left((x_pW_Q)(W_K^Tx^T)\right) > \Gamma\left(((x_pW_Q)W_K^T)x^T\right) \\
\Rightarrow &PFF_H + NFF_H + PNF_H > 2PFF_H + PFN \\
\Rightarrow &NFF_H + PNF_H > PFF_H + PFN \\
\Rightarrow &\frac{1}{P} - \frac{1}{N} > \frac{F-F_H}{FF_H}
\end{aligned}$$

which means when the above condition holds $((x_pW_Q)W_K^T)x^T$ is better than $(x_pW_Q)(W_K^Tx^T)$ regarding the computation complexity. Notice that the condition is the same as Eq. (9). Therefore when Eq. (9) holds, the following equation achieves the lowest computation complexity to obtain $A_p(x)$:

$$A_p(x) = \left(\text{softmax}\left(\frac{((x_pW_Q)W_K^T)x^T}{\sqrt{F_H}}\right)x\right)W_V$$

Otherwise

$$A_p(x) = \text{softmax}\left(\frac{(x_pW_Q)(xW_K)^T}{\sqrt{F_H}}\right)(xW_V)$$

is the best computation order. The proof is finished.

□

According to the theorem above, there are only two candidates among all the possible computation orders. Notice that there is a deceptive method to optimize the computation of Eq. (7). Since attention weights W_Q, W_K are constants during the inference, it seems we can compute $W_QW_K^T$ in advance such that there will be only three matrices involved in Eq. (7). This actually works for the single-head attention mechanism, but when it comes to multi-head attention, $W_QW_K^T \in \mathbb{R}^{F\times F}$ is much bigger than $W_Q, W_K \in \mathbb{R}^{F\times F_H}$, which eventually increases the computation complexity.

Also, Theorem 2 indicates that when the model is deployed on a single device, *i.e.* $P = N$, the original computation flow is already the most efficient one. The opportunity that we can change the computation orders to reduce the computation complexity exclusively exists when we try to partition the self-attention function. With different input sizes and partition sizes, Theorem 2 guarantees that we can always choose the most efficient way to do the computation.

C. Partitioned Transformer Layer

Based on our previous analysis, we can formally introduce our partitioned transformer layer. The layer takes the whole input sequence x and a range of the desired output partition p, which can be specified by the positions, as the input, and generates the corresponding output partition of this transformer layer. Based on the input and layer settings, the algorithm automatically chooses the most efficient way and applies it to each head of the self-attention function[1]. Then the attention output can be directly fed into the subsequent position-wise

[1]The multi-head attention can be implemented through tensor multiplications instead of iterating each head, but the computation complexities are the same.

Algorithm 1 Partitioned Transformer Layer.

Require: $x \in \mathbb{R}^{N \times F}$, desired partitions p
Ensure: Transformer layer output partition $T_p(x)$
$P \leftarrow$ partition length
for each head of attention $i = 1, \ldots, H$ **do**
  if $\frac{1}{P} - \frac{1}{N} > \frac{F - F_H}{FF_H}$ **then**
    Compute $A_p^{(i)}(x)$ with Eq. (8)
  else
    Compute $A_p^{(i)}(x)$ with Eq. (3)
  end if
end for
$R \leftarrow \mathrm{Concat}(A_p^{(1)}(x), \ldots, A_p^{(H)}(x))W_O$
$Y \leftarrow \mathrm{LayerNorm}(R + x_p)$
$T_p(x) \leftarrow \mathrm{LayerNorm}(Y + \mathrm{FFN}(Y))$
Return $T_p(x)$

feed-forward network and layer normalization to generate the desired output partition. The step-by-step procedures are illustrated in Algorithm 1.

Theorem 2 guarantees that the algorithm can always choose the most efficient way to do the computation for a specific input and layer settings. We further introduce the following theorem to show its scalability.

Theorem 3. *Given input $x \in \mathbb{R}^{N \times F}$ and input partition $x_p \in \mathbb{R}^{P \times F}$, if $P = \frac{N}{K}$, we have $\Gamma(\text{Algorithm 1}) = O(\frac{1}{K})$.*

Proof. Except for the self-attention part, the remain operations of Algorithm 1 are position-wise, therefore the computation complexity is naturally $O(P) = O(\frac{N}{K}) = O(\frac{1}{K})$. We only have to show that the self-attention part of Algorithm 1 is $O(\frac{1}{K})$.

Replace P by $\frac{N}{K}$, Eq. (9) becomes

$$\begin{aligned} &\frac{1}{P} - \frac{1}{N} > \frac{F - F_H}{FF_H} \\ \Rightarrow &\frac{K-1}{N} > \frac{F - F_H}{FF_H} \\ \Rightarrow &K > \frac{F - F_H}{FF_H} N + 1. \end{aligned}$$

Therefore when $K > \frac{F-F_H}{FF_H} N + 1$, Algorithm 1 uses the following equation to compute the self-attention function:

$$A_p(x) = \left(\mathrm{softmax} \left(\frac{((x_p W_Q) W_K^T) x^T}{\sqrt{F_H}} \right) x \right) W_V.$$

The computation complexity can be computed through the same procedure as the proof of Theorem 1, which is

$$3PFF_H + 2PNF + O(PN).$$

Replace P by $\frac{N}{K}$ yields the result and finishes the proof. □

Compared with the naive computation method, our proposed method gets rid of the constant term in Eq. (3) therefore doesn't have the bottleneck. Theorem 3 indicates that Algorithm 1 achieves linear acceleration with respect to the number of partitions for a single transformer layer, which means it can be perfectly scaled to more devices in practice.

V. *Voltage* Design

A. System overview

Now we have presented how to efficiently partition the computation workload of a single transformer layer in a position-wise manner, in this section, we introduce the design of *Voltage*, a distributed inference system that expands our method to the entire transformer model.

Voltage considers the transformer model as a stack of transformer layers where the output of one layer feeds directly into the next. Despite transformer layers, the model also contains some layers for pre- and post-processing, such as an embedding layer that converts language tokens into embeddings, and a classifier that generates predictions. Therefore, as illustrated in Fig. 3, users will submit their inference requests to a terminal device which will perform pre-processing on the user input and then distributes the input features to all other computing devices where transformer layers are located, thereby initiates the distributed inference process.

Let $T(x)$ denote the output for transformer layer, $T(x)$ can be parallel obtained at different devices with Algorithm 1, but the subsequent layer still requires the entire $T(x)$ as input since both Eq. (3) and Eq. (8) involve all elements in x. This means we have to synchronize the output partitions among all the devices through an All-Gather operation to obtain $T(x)$ before entering the next layer.

As a result, the computation workload of each transformer layer is partitioned and executed at different devices while the layer computations are interleaved by data synchronizations to get the input data ready. This process repeats until the last transformer layer, and then the terminal device will collect the computation results of transformer layers and deliver the inference results to users.

The above process is formally described in Algorithm 2, *Voltage* takes the inference data x and a partition scheme as the input, where the partition scheme indicates how the workload is distributed among the devices. When the inference request comes, *Voltage* will distribute the input data x to all devices, then for each transformer layer, the device computes the assigned output partition according to Algorithm 1, and then synchronize the output through an All-Gather communication primitive. By the end of this layer, all the devices should be able to assemble the full output of the layer, which becomes the input for the next layer, and starts a new round of computation.

B. Partition Scheme

Voltage partitions the inference workload along the sequence dimension, which means the partition scheme should dictate which positions each device is supposed to compute. However, the input sequences of transformer models typically have different sizes. Therefore, we cannot give a fixed partition size for each device. Instead, we express the partition scheme as a vector of ratios.

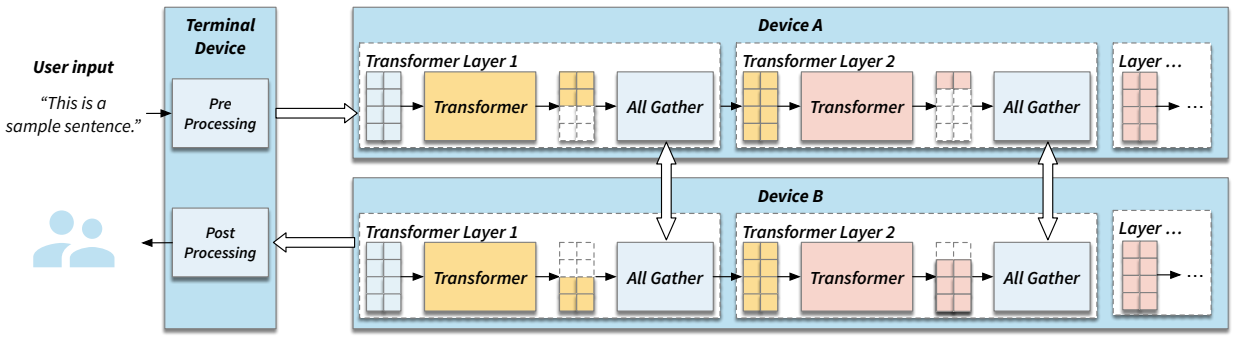


Fig. 3. Overview of *Voltage*. The user will submit inference requests and gather results from a terminal device, which performs pre-processing before distributing inputs to other devices for transformer computation, and post-processes the transformer outputs to generate predictions.

Assume there are K computing devices, then a partition scheme P can be represented as:

$$P = [p_1, p_2, \cdots, p_K]$$

where p_i should satisfy the following two conditions:

$$0 \leq p_i \leq 1, \quad \forall i = 1, 2, \cdots, K$$
$$\sum_{i=1}^{K} p_i = 1.$$

In this way, given an input sequence with length N, p_i indicates the positions that device i will be computing during the inference falls in between $N\sum_{j=0}^{i-1} p_j$ and $N\sum_{j=0}^{i} p_j$. And the second condition guarantees all positions are covered during the inference. Since the output of each transformer layer T and the input are bijective, the above conditions also imply

$$T_{p_i}(x) \cap T_{p_j}(x) = \emptyset, \quad \forall i \neq j,$$
$$\cup_{i=1,\ldots,k} T_{p_i}(x) = T(x),$$

which means there are no overlapping areas between different devices, and a full-position output of the layer can be rebuilt from the output partitions.

Notice that we enforce that all transformer layers share the same partition ratio at this point for the sake of simplicity. But each transformer layer has all the input data ready after data synchronization, which means it is totally able to compute any other positions other than the assigned ones. As illustrated in Fig. 3, the positions that device A and B computes at layer 1 and layer 2 are different. This implies that *Voltage* is flexible enough to dynamically adjusting partition schemes for each layer during the runtime without any penalty, and we will leave it for the future discussion.

C. Comparison with Existing Parallelisms

Unlike existing works that borrow parallel computation mechanisms from model training phase, the position-wise partitioning of *Voltage* is tailor-designed to improve the inference latency in the edge environment. We outline the difference and advantage of *Voltage* compared with existing parallelism as follows:

Algorithm 2 Distributed Transformer Inference.

Require: Partition scheme $P = [p_1, p_2, \ldots, p_K]$

User submit inference request to terminal device.
Pre-processing request to obtain input features x
Distribute x to all devices
for each transformer layer T **do**
  for each device $k = 1, 2, \ldots, K$ **do**
    Compute $T_{p_k}(x)$ with Algorithm 1
    if T is the last transformer layer **then**
      Send $T_{p_k}(x)$ to terminal device
    else
      Synchronize $T_{p_k}(x)$ with other devices
    end if
    Assemble output partitions to obtain $T(x)$
    $x \leftarrow T(x)$
  end for
end for
Terminal device collects output from other devices
Return inference result to user

Data parallelism and pipeline parallelism, two common model parallelism techniques, are not well-suited for low-latency inference serving in edge environments. Data parallelism requires dividing a large batch of input data into smaller microbatches that are computed in parallel across devices. Similarly, pipeline parallelism partitions the model layer-wise across devices, so that each device starts computing the next microbatch while handing intermediate results to the next device. Both techniques therefore rely on having a sufficiently large batch size to fully utilize the pipeline or parallel devices. However, in edge environments, inference requests typically arrive in a discrete, sporadic manner with a batch size of 1. These techniques aim to improve throughput given sufficient input samples rather than optimizing the latency of individual requests. As edge environments are latency-sensitive with variable and often small batch sizes, data and pipeline parallelism struggle to effectively distribute the inference workload. In contrast, *Voltage* manages to efficiently distribute the inference workload among multiple devices even though the batch size

is only one.

Unlike data and pipeline parallelism, tensor parallelism is able to partition the computation workload across devices even with a batch size of 1. Tensor parallelism splits the weight matrices of each matrix multiplication operation among multiple devices. Therefore, the partitioning method of tensor parallelism is independent of batch size. Though the batch size may be small or even 1 in edge environments, tensor parallelism can still distribute the computations by splitting individual weight matrices across available devices.

However, tensor parallelism incurs high communication costs that can negate performance gains from parallelism. As shown in Fig. 2, tensor parallelism requires two All-Reduce operations per layer, incurring substantial communication overhead. According to [13], given an input length N, device number K and feature size F, the total per-device communication volume per layer is $4(K-1)NF/K$ with tensor parallelism. In contrast, *Voltage* only requires a single All-Gather operation for data synchronization between layers. Consequently, *Voltage*'s communication size is just $(K-1)NF/K$ per layer, only 1/4 that of tensor parallelism.

The reason *Voltage* achieves substantially lower communication overhead compared to tensor parallelism is because *Voltage* is designed solely for efficient inference, while tensor parallelism targets both training and inference. During backward propagation in training, tensor parallelism performs a transposed synchronization of gradients similar to what it does for activations in the forward pass. This adds another $4(K-1)NF/K$ of communication per layer for the backward pass. In contrast, since *Voltage* replicates the full model weights on each device, the weights get updated with gradients from different input position slices at different devices. To complete the backward pass, the updated model weights must be synchronized across all devices after the entire batch gradient computation. By focusing only on optimization for inference, *Voltage* sacrifices communication efficiency of the backward pass, which will never happen, to attain minimal communication overhead during the forward inference phase.

VI. Evaluation

A. Experiment Settings

Environment. We use PyTorch 1.9 CPU version to implement *Voltage* and the transformer models since most of the edge devices don't have a sophisticated accelerator. The system is then deployed on six virtual machines hosted on Compute Canada, where each of them has one virtual CPU and 7.6 GB of physical memory. To simulate the edge environment, we limit the network bandwidth to 500Mbps by default.

Transformer Models. We adopt three well-known transformer models from Huggingface for evaluation, including `BERT-Large-Uncased`, `ViT` and `GPT2`. `BERT` and `GPT2` will be deployed to handle text classification task, and `ViT` will be used for image classification.

During the inference, another device in the same network will take the role of a terminal device, which generates the corresponding input data for the model, *e.g.,* a random string with 200 words for `BERT` and `GPT2`, and a 224×224 image for `ViT`. The input features of transformer layers will be broadcast to all devices, and the latency is measured by the time between the device broadcasting the request data and receiving the results. Following other distributed inference works, we set the batch size to 1.

Baseline. *Voltage* aims to reduce the inference latency by exploiting the computation capacity of multiple devices. Therefore, one of our baselines is deploying the transformer model on a single device. Besides single device deployment, we also compared *Voltage* with the tensor parallelism we introduced before. But we skip the comparison with pipeline parallelism because this paper focuses only on inference latency while pipeline parallelism optimizes throughput and has no improvement for individual latency.

The inference workload is evenly partitioned among the devices, which means each of them computes $\frac{1}{K}$ of the positions for *Voltage*, or $\frac{1}{K}$ of attention heads for tensor parallelism, where K is the number of available devices.

B. Experiment Results

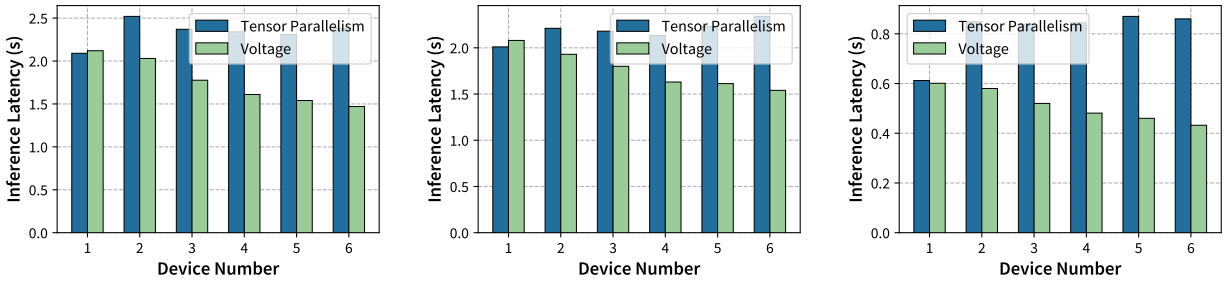
Inference Latency. We first evaluate the overall inference latency of these three transformer models. Our results are presented in Fig. 4, we can see that with the increasing of available device, *Voltage* manages to reduce the inference latency which verifies the effectiveness of our approach. The experimental results show that compared with single device deployment, *Voltage* exhibits a better scalability, reducing the inference latency of `BERT` by up to 27.9% with six devices, 29.1% and 32.1% for `ViT` and `GPT2`, respectively.

On the other hand, the results of tensor parallelism show that it fails to achieve acceleration regarding inference latency. Though both *Voltage* and tensor parallelism efficiently partitions the computation workload among devices, the two All-Reduce operation of tensor parallelism not only incurs significant communication overhead, but also interrupts the computation process, adding extra delay. As a result, distributing inference workloads with tensor parallelism is even slower than a single device.

Impact of Bandwidth. As cross-device distributed inference requires intermediate results to be exchanged over network connections, the performance can be extremely sensitive to network conditions.

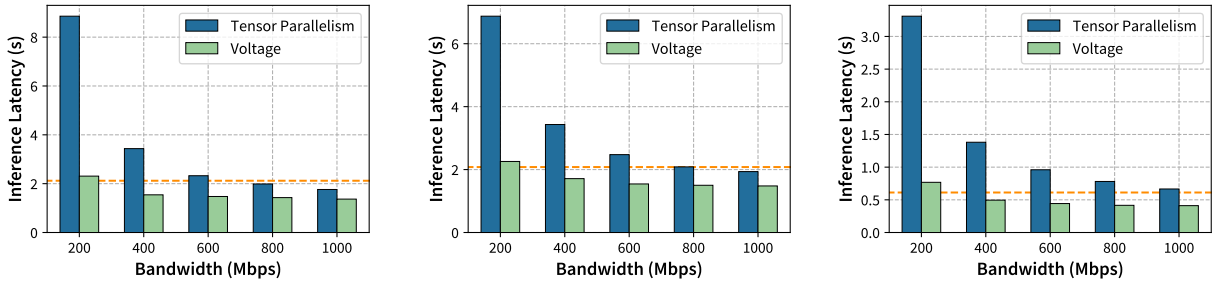
Next, we fix the device number at 6 and compare the inference latency of *Voltage* and tensor parallelism under different bandwidth settings. As presented in Fig. 5 where the inference latency of single device deployment is represented by the orange dashed line, Voltage consistently outperforms tensor parallelism across all scenarios.

Due to the huge communication overhead, the inference time of tensor parallelism is dominated by the communication which makes it extremely sensitive to the network bandwidth. With the increasing of available bandwidth, the inference



(a) BERT (b) ViT (c) GPT2

Fig. 4. Overall inference latency with increasing device number.



(a) BERT (b) ViT (c) GPT2

Fig. 5. Overall inference latency with different bandwidth limit, orange dashed line represents the single device deployment.

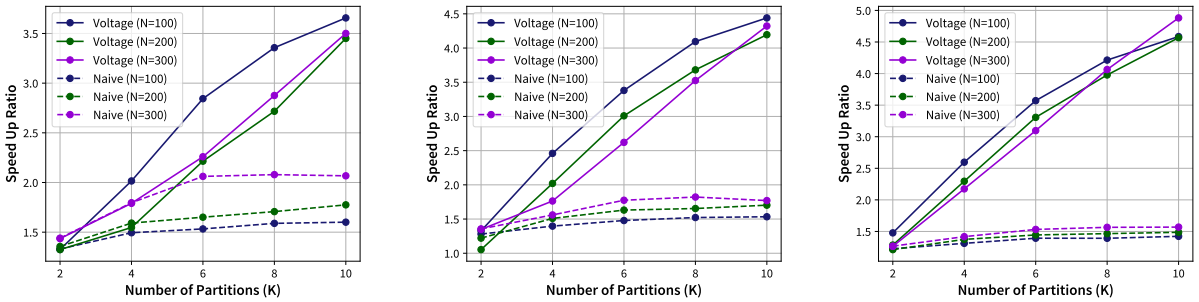
latency of tensor parallelism can be improved but still slower than *Voltage*. *Voltage* achieves improved performance starting from 400Mbps, while tensor parallelism requires at least 1000Mbps to outperform the deployment on single device. Both methods fail to improve the inference latency when the bandwidth is as low as 200Mbps, and tensor parallelism even takes about 4.2× longer to finish the inference on `BERT`. Therefore, distributed inference with tensor parallelism is impractical for edge environments.

Partition Efficiency. Dynamically selecting the best computation order is one of the highlight of Algorithm 1. Therefore, to evaluate the effectiveness of our proposed method, we isolate the multi-head self-attention mechanism and compare the speed-up ratio from different perspectives. Since Algorithm 1 is determined by the configurations of the self-attention mechanism, we manipulate the settings, including number of attention heads H and feature dimension of each head F_H, and create the following three synthetic layers to evaluate Algorithm 1 from different scenarios: $(H = 16, F_H = 64)$, $(H = 8, F_H = 128)$ and $(H = 4, F_H = 256)$. We record the time that our proposed method and the naive method need to compute the output partition with length $P = \frac{N}{K}$ where the input lengths N are chosen from (100, 200, 300). The time is then compared with computing the full-size output to obtain the speed-up ratio.

The results are presented in Fig. 6. We can observe that across all three settings, the speed-up ratio of the naive partition can be slightly improved but soon stops increasing, which verifies our claim that the naive partition method cannot distribute the self-attention mechanism efficiently. In contrast, *Voltage* has a similar performance as the naive method when the partition number is small, but as the number of partitions increases, *Voltage* is able to achieve linear acceleration, which verifies the correctness of Theorem 3 and proves *Voltage* has better scalability if there are more available devices to join the partition.

Also, we can see that the advantage of our proposed method is closely related to the attention settings. When the attention feature dimension F_H increases from 64 to 256, the gap between the naive and proposed method becomes greater, and our method can be up to 3.4× faster than the naive one. This is because the naive method has to compute the intermediate matrices $K, V \in \mathbb{R}^{N \times F_H}$ in advance, which takes more time when F_H increases. On the contrary, our proposed method doesn't have to compute these two matrices thus won't be affected by the size of F_H.

While our method achieves linear acceleration relative to a single transformer layer, this acceleration is not that significant when looking at the entire model due to communication overhead. Specifically, although *Voltage* only requires 1/4 of the communication overhead of tensor parallelism, it still remains a bottleneck when the number of devices scales up. Further optimizations to communication protocols and exchange mechanisms may help relieve this bottleneck in future work.



(a) $H = 16, F_H = 64$ (b) $H = 8, F_H = 128$ (c) $H = 4, F_H = 256$

Fig. 6. Speed-up ratio when partitioning the multi-head self-attention mechanism.

VII. Related Work

A. Deploying Transformer on Edge Devices

There have been enormous efforts to deploy transformer models on edge devices by compressing them into smaller models. For instance, Paul et al. proposed reducing the number of attention heads to decrease computational complexity [18]. Q8bert [19] quantizes the weights of BERT models from 32 bits to 8 bits so they can execute on memory-constrained devices. Similarly, parameter factorization [20] and knowledge distillation [7] have been adopted to find more compact transformer architectures suitable for edge hardware. These methods aim to find an acceptably accurate compact model that fits edge resource constraints.

In contrast, *Voltage* makes no modifications to the original model architecture or weights. Therefore, it does not need to balance model compression and accuracy loss. More importantly, the compressed transformer models from above techniques can also leverage *Voltage*'s distributed inference system for further acceleration, as long as they retain the core transformer architecture. This provides an orthogonal performance boost without re-engineering model architecture or re-training for different hardware targets.

B. Distributed Inference System

Distributed inference is an effective way to accelerate inference speed on edge devices. Techniques like DeepThings [10] take advantage of properties like partial receptive fields in convolutional neural networks to parallelize inference by splitting input feature maps across devices. The follow-up works like CoEdge [11], DeepSlicing [21] and EdgeFlow [22] take the network and device heterogeneity into consideration for further improvements. However, these methods are exclusively designed for CNNs and cannot be applied to transformer models.

On the other hand, some works have tried distributing transformer model inferences by borrowing techniques from parallel training. For instances, DeepSpeed [16] and Parallelformer [17] use tensor parallelism to partition weight matrices across devices. PipeEdge [23] adopts pipeline parallelism instead to improve overall throughput. However, as discussed previously, tensor parallelism incurs high communication costs for cross-device inference. Meanwhile, pipeline parallelism requires sufficient batch size to keep the pipeline utilized and does not optimize the latency of individual requests.

C. Transformer Optimization

Due to the significant computational complexity of the transformer layer, especially the self-attention mechanism, another line of work tries to redefine self-attention to achieve acceleration. For example, Reformer [24] uses locality-sensitive hashing to reduce the attention complexity from $O(N^2)$ to $O(N \log N)$ regarding the input length N. [25] even achieves linear complexity by exploiting the associative property of matrix multiplication. Additionally, Linformer [26] proposes to approximate the original attention function through low-rank matrix multiplications, which is also a linear transformer implementation. Since these models follow the overall transformer architecture and workflow except for modifications to the attention phase, *Voltage* can be easily extended to distribute them with minor changes to the customized attention procedures.

VIII. Concluding Remarks

In summary, this paper introduces *Voltage* to distribute the inference workload of transformer models to multiple devices for acceleration. The core design of *Voltage* is to partition the outputs of each single transformer layer such that they can be computed at different devices in a parallel manner. By investigating the computation complexities of the multi-head self-attention mechanism, *Voltage* achieves linear acceleration on a single transformer layer, which has been verified both theoretically and empirically. Due to the communication overhead, *Voltage* cannot speed up the whole transformer model as much as a single transformer layer, but still outperforms tensor parallelism, and significantly reduces the inference latency compared with single device deployment. Therefore besides the existing efforts, we believe the distributed inference paradigm introduced by *Voltage* can be a promising start for future research to deploy transformer models on resource-constrained devices.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All You Need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 6000–6010.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019, pp. 4171–4186.

[3] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language Models are Few-Shot Learners," *arXiv preprint arXiv:2005.14165*, 2020.

[4] P. Ramachandran, N. Parmar, A. Vaswani, I. Bello, A. Levskaya, and J. Shlens, "Stand-Alone Self-Attention in Vision Models."

[5] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "Mobilebert: a Compact Task-Agnostic Bert for Resource-Limited Devices," *arXiv preprint arXiv:2004.02984*, 2020.

[6] M. A. Gordon, K. Duh, and N. Andrews, "Compressing Bert: Studying the Effects of Weight Oruning on Transfer Learning," *arXiv preprint arXiv:2002.08307*, 2020.

[7] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[8] H. Tsai, J. Riesa, M. Johnson, N. Arivazhagan, X. Li, and A. Archer, "Small and Practical BERT Models for Sequence Labeling," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 3632–3636.

[9] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[10] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[11] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "CoEdge: Cooperative DNN Inference with Adaptive Workload Partitioning over Heterogeneous Edge Devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.

[12] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[13] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient Large-Scale Language Model Training on GPU Clusters using Megatron-LM," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[14] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *Advances in neural information processing systems*, vol. 32, pp. 103–112, 2019.

[15] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer Normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[16] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley *et al.*, "DeepSpeed-Inference: Enabling Efficient Inference of Transformer Models at Unprecedented Scale," in *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2022, pp. 646–660.

[17] H. Ko, "Parallelformers: An Efficient Model Parallelization Toolkit for Deployment," https://github.com/tunib-ai/parallelformers, 2021.

[18] P. Michel, O. Levy, and G. Neubig, "Are Sixteen Heads Really Better than One?" *Advances in Neural Information Processing Systems*, vol. 32, pp. 14 014–14 024, 2019.

[19] O. Zafrir, G. Boudoukh, P. Izsak, and M. Wasserblat, "Q8bert: Quantized 8bit bert," *arXiv preprint arXiv:1910.06188*, 2019.

[20] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=H1eA7AEtvS

[21] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "Deepslicing: Collaborative and Adaptive CNN Inference with Low Latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2175–2187, 2021.

[22] C. Hu and B. Li, "Distributed Inference with Deep Learning Models across Heterogeneous Edge Devices," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 330–339.

[23] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, "PipeEdge: Pipeline Parallelism for Large-Scale Model Inference on Heterogeneous Edge Devices," in *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2022, pp. 298–307.

[24] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The Efficient Transformer," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rkgNKkHtvB

[25] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5156–5165.

[26] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-Attention with Linear Complexity," *arXiv preprint arXiv:2006.04768*, 2020.