# Confidant: Customizing Transformer-based LLMs via Collaborative Edge Training

Yuhao Chen, Yuxuan Yan, Qianqian Yang, Yuanchao Shu, Shibo He and Jiming Chen Zhejiang University

# **ABSTRACT**

Transformer-based large language models (LLMs) have demonstrated impressive capabilities in a variety of natural language processing (NLP) tasks. Nonetheless, it is challenging to deploy and fine-tune LLMs on mobile edge devices with limited computing, memory, and energy budgets. In this paper, we propose Confidant, a multi-backend collaborative training framework for customizing state-of-the-art LLMs on commodity mobile devices like smartphones. Confidant partitions an LLM into several sub-models so that each fits into a mobile device's memory. A pipeline parallel training mechanism is further developed to ensure fast and efficient distributed training. In addition, we propose a novel backend scheduler to allocate different attention heads to heterogeneous compute hardware, including mobile CPU and GPUs, to maximize the compute resource utilization on each edge device. Our preliminary experimental results show that Confidant achieves at most 45.3% memory reduction and 8.03x inference speedup in practical settings.

## 1 INTRODUCTION

Transformer-based large language models (LLMs), exemplified by models like BERT [3], LLaMa [9] and GPT-4 [7], have ushered in a remarkable era of progress in machine learning and artificial intelligence. Presently, LLM-based applications are predominantly deployed in the cloud, necessitating users to transmit their data to remote servers and await responses. Deploying LLMs on mobile devices allows for the processing of data locally, enhancing data privacy and real-time response. Moreover, compared with cloud-hosted LLMs that provide only general knowledge, locally customized LLMs could learn domain- and user-specific knowledge through fine-tuning, providing much higher inference accuracy and user experience.

Fine-tuning an LLM on mobile devices indeed encounters two main challenges. Firstly, LLM training requires way more memory compared to both model inference and the training of CNNs [1]. To illustrate this, we implemented a BERT model [3] on a Redmi K10X Pro mobile phone, using MNN [5] and measured the memory usage of the model itself, inference and training with different batch sizes, respectively. As shown in Figure 1, even fine-tuning the BERT model with a batch size of 8 requires approximately 5GB of memory. Furthermore, we provide a comparison of the fine-tuning

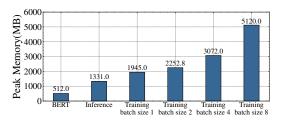


Figure 1: Memory usage of BERT, inference and training with different batch sizes on Redmi K10X Pro using MNN.

Table 1: Fine-tuning memory usage of LLMs and memory of the mobile devices released recently.

LLMs	Fine-tuning Memory Usage	Mobile Devices	Memory
Bert-Large	~16GB	Huawei Mate 60 Pro	12GB
GPT2-XL	~24.5GB	Google Pixel 8 Pro	12GB
LLaMA-7B	~28GB	iPhone 15 Pro	8GB
LLaMA-13B	~48GB	Galaxy s23 Ultra	12GB

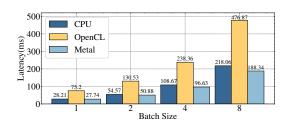


Figure 2: Computation time of the self-attention layer with different batch sizes on an M1 Macbook Pro using different backends.

memory usage of some state-of-the-art LLMs and the memory capacities of recently released mobile devices in Table 1. While some mobile devices may have memory capacities exceeding 5GB, we note that 5GB of free memory is not always guaranteed, as users may be simultaneously running other applications. These observations highlight the practical limitations of fine-tuning LLMs on resource-constrained mobile devices, emphasizing the need for memory-efficient training techniques.

Secondly, the limited computing capacities of mobile devices have a direct impact on real-time response and fine-tuning speed. In server environments, current deep neural

network (DNN) frameworks generally utilize CUDA, a GPU computation toolkit, as the backend to accelerated computation. The backend, in this context, refers to the underlying engine or computational platform for executing neural network operations. However, mobile devices do not support CUDA on their GPUs. Instead, current industry-scale mobile frameworks typically use alternative backends like OpenCL and Metal to perform DNN computations on mobile GPUs. Despite these efforts, the limited computational capacity of mobile GPUs and the lack of optimization for on-GPU backends in current mobile frameworks results in similar computation latency for DNNs by a CPU and a mobile GPU. To illustrate, we measure the latency of calculating a selfattention layer in BERT with different batch sizes using CPU, OpenCL and Metal, respectively, as the backend on an M1 Macbook Pro laptop, shown in Figure 2. It indicates that the computation times using these three backends are of the same order of magnitude, and in some cases, using OpenCL is even slower than using CPU. Additionally, it is worth noting that current mobile frameworks only support computing DNNs with a single backend on one device, leaving the available computational capacities not fully utilized.

To tackle the aforementioned challenges, we introduce Confidant in this paper, a multi-backend edge collaborative training framework for fine-tuning transformer-based LLMs on mobile devices. Motivated by the fact that individuals often possesses multiple mobile devices, Confidant partitions an LLM into several sub-models, and place them on multiple mobile devices to facilitate collaborative fine-tuning. Confidant follows a pipeline-parallel training mechanism to reduce potential stalling, and further introduces a novel backend scheduler that allocates different numbers of attention heads within a submodel to all available backends on each participating device, accelerating fine-tuning through concurrent training on heterogeneous hardware. Preliminary results show that fine-tuning an LLM using Confidant achieves up to 45.3% memory reduction and 8.03x speedup compared to fine-tuning using a single mobile device.

Compared with prior work on distributed edge training which targets convolutional-based models, NVIDIA (e.g., Jetson) and non-GPU devices (e.g., Raspberry Pi) [2, 14], to the best of our knowledge, we are the first to implement distributed training of transformer-based LLMs on mobile devices with both CPU and GPU using an industry-scale mobile DNN framework. Confidant further accelerates the training by a novel pipeline-parallel design which leverages unique characteristics of both the Transformer structure and the mobile hardware.

## 2 BACKGROUND

### 2.1 Self Attention Mechanism

The impressive performance of the LLM is rooted in the transformer structure [10], where the self-attention mechanism plays a pivotal role. The self-attention mechanism typically consists of multiple attention heads which process the input in a parallel manner, as shown in the left-bottom box of Figure 3. Each attention head functions as follows: the input X is initially transformed by three matrices  $W_i^Q$ ,  $W_i^K$  and  $W_i^V$ , producing the query matrix  $Q_i$ , key matrix  $K_i$ and value matrix  $V_i$ , respectively. The query matrix  $Q_i$  is then multiplied with  $K_i$ , where the resulting matrix is passed through a softmax function to obtain the attention score matrix. Then the value matrix is multiplied with the attention score matric to obtain the final output of the attention head  $Z_i$ . Finally, the outputs from all attention heads are concatenated for subsequent processing in the transformer structure. Each attention head with a different set of weights captures distinct patterns within the input data, enabling the transformer structure to capture different dependencies and relationships between elements in the input data.

# 2.2 Pipeline Parallel Training

Pipeline parallel training is one of the widely-used techniques to speed up collaborative training, which was first proposed by Microsoft for training DNN models on GPU clusters [6]. It first divides a DNN model into several sub-models and trains each sub-model in a different GPU, where the forwarding and the backwarding of a certain batch are executed in a consecutive order. It then accelerates the training by allowing each GPU to forward a batch using stale sub-model weights instead of waiting for the newest sub-model weights.

We illustrate the idea of pipeline parallel training using three mobile devices as an example in Fig 4, where device 1 is the central node and  $i_{\rm f}^{ver}$  and  $i_{\rm b}^{ver}$  denote the forwarding and backwarding of the i-th training batch with the weights of the version number ver, respectively. Consider the training of batch 1 in device 1, instead of waiting for the backwarding of batch 0, device 1 forwards batch 1 with the same weights used by batch 0. To guarantee the model convergence, the pipeline parallel training applies the oneforward-one-backward (1F1B) rule, guaranteeing edge devices performing the forwarding and backwarding of the local sub-model alternatively. Additionally, the weights utilized to forward a certain batch are stashed for subsequent backwarding of the same batch. In our previously proposed framework FTPipeHD [2], we have extended the pipeline parallel training to edge devices with a novel dynamic model partitioning approach. The proposed approach periodically

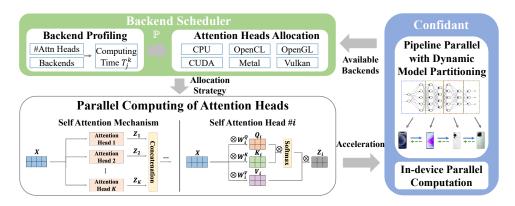


Figure 3: The overview of the proposed system.

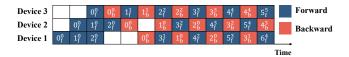


Figure 4: An illustration of the pipeline parallel training technique.

estimates the computing capacity of each worker node and calculates the optimal model partition points accordingly.

# 3 DESIGN OF CONFIDANT

# 3.1 System Overview

As shown in Figure 3, the proposed system *Confidant* follows the pipeline parallel training mechanism in our previous work, FTPipeHD [2], which dynamically partitions the model into several sub-models and trains each of them on a different device in a pipeline parallel manner.

While FTPipeHD was designed for image models like MobileNetV2 and was implemented on edge devices such as Raspberry Pi, our current work focuses on the distributed training of transformer-based large language models, leading to notable variations in implementation, as discussed in Section 3.3. To further accelerate the training process, we extend the approach by utilizing multiple backends within a single device, enabling parallel computation of attention heads across these backends. We propose a novel backend scheduler that optimally allocates varying numbers of attention heads to all available backends on a device, with the aim of minimizing computation time and improving training efficiency.

#### 3.2 Backend Scheduler

The proposed backend scheduler encompasses two key steps: backend profiling and attention head allocation.

3.2.1 Backend Profiling. The backend profiling step begins by identifying all available backends on the mobile devices. It then proceeds to profile the computation time of different numbers of attention heads on each of these backends. There are two ways for computing multiple attention heads on a given backend: (1)Compute them as one large attention head and (2) Compute them separately. In the first way, the  $W_i^Q$ ,  $W_i^K$ , and  $W_i^V$  tensors are concatenated into three larger tensors,  $W_O$ ,  $W_K$ , and  $W_V$ , respectively. This allows for subsequent computations of Q, K, V, and Z to be performed as a single tensor, without the need for separate calculations for each attention head. In the second way, each attention head is calculated individually. The experimental results reveal that, for certain backends, the first way of computing multiple attention heads as a single large attention head is faster than the second method. This performance difference may be attributed to the fact that modern deep neural network (DNN) frameworks incorporate optimizations to accelerate large tensor computations [4, 12]. For each backend denoted as *j*, we record the time taken to compute *k* attention heads using both the first and second methods. The faster of the two times is then chosen as the profiling result and is represented as  $T_i^k$ . These  $T_i^k$  values collectively constitute a profiling dataset denoted as P, which is subsequently utilized in the attention head allocation step.

3.2.2 Attention Heads Allocation. In this step, the allocation of attention heads is carried out according to Algorithm 1. The objective is to distribute the attention heads to each available backend in such a way that the computation time for the allocated attention heads on each backend is roughly equivalent. More precisely, we employ a binary search approach to identify the minimum execution time required for all the available backends to collectively complete the computation of all the attention heads.

Suppose the goal is to allocate K attention heads to M available backends in an optimized manner. We initialize the

binary search by setting the lower bound l to 0 and the upper bound r to the minimum time required by any backend to compute all K attention heads. Then, iteratively, we calculate the mid-value as mid = (l+r)/2. Based on this mid-value, we assess the feasibility of allocating attention heads so that each backend's computation time approximates mid. Specifically, for each backend j, we identify a number k within the range [1, K] such that  $T_i^k$ , time taken by backend j to compute k attention heads, is closest to mid. If the deviation of  $T_i^k$ from mid exceeds a given threshold  $\epsilon$ , we assign  $k_i = 0$ . This means we bypass backends that are either too slowly or too fast. If the sum of all assigned  $k_i$  values meets or surpasses K, an allocation strategy exists where the total computation time for all K attention heads is roughly mid. In such a case, we adjust the upper bound for the binary search as  $r = mid - \sigma$ , where  $\sigma$  is a small value to prevent infinite looping. If the sum of all  $k_i$  is less than K, it suggests that processing all the attention heads within mid time is not feasible. Consequently, the lower bound is adjusted to  $l = mid - \sigma$ . The binary search ends when l > r. Finally, based on the determined allocation vector S, attention heads are distributed among the device's backends. These backends then execute their allocated computations in parallel during training, enhancing the efficiency of the process.

# **Algorithm 1:** Allocation of attention heads on multiple backends

```
Input: The profiling set \mathbb{P}, total attention heads K, total available backends
    Output: The allocation strategy \mathbb{S} = \{(j, k_j) | j = 1, ..., M, k_j = 0, 1, ..., K\}.
1 Initialize l \leftarrow 0, r \leftarrow \min_{j=1,\dots M} T_j^K, \mathbb{S} \leftarrow \{\};
    while l \leq r do
           mid \leftarrow (l+r)/2;
           if isValid(mid, K, \mathbb{P}) then
                 r \leftarrow mid - \sigma;
 5
           else
                 l \leftarrow mid + \sigma;
           end
9 end
10 return S;
    Function isValid(mid, K, \mathbb{P}):
11
           Initialize totalHeads \leftarrow 0, S' \leftarrow \{\};
12
           for i \leftarrow 1 to M do
13
                  k_j = \arg\min_{\substack{k=1,\dots K}} abs(T_j^k - mid);
                  if abs(T_i^{k_j} - mid) > \epsilon then
15
                    k_j \leftarrow 0;
16
17
                  totalHeads \leftarrow totalHeads + k_i;
18
19
                  insert (j, k_j) into S';
20
           if totalHeads >= K then
21
                  \mathbb{S} \leftarrow \mathbb{S}':
22
                  return true;
23
           else
24
                  return false;
25
           end
26
```

# 3.3 Implementation

We then develop an application on mobile devices for the implementation of Confidant by developing based on MNN [5] of version 2.7.0 as the deep learning framework. MNN supports training DNN models on Android and iOS and outperforms other frameworks in terms of computation time and memory usage as shown in [11]. Nonetheless, given that MNN is implemented in C++ while the application is developed in Java, we employ the Java Native Interface (JNI) [13] to invoke C++ functions from within the application. The communication between mobile devices is facilitated using standard HTTP requests.

Our work is pioneering in implementing the transformer model for training on MNN. The process of loading pretrained weights of a LLM from the corresponding PyTorch model involves the following steps: PyTorch-formatted pretrained weights are first converted into the ONNX format, which is an open standard for representing machine learning models; The MNNConvert tool provided by MNN is then employed to load the weights in ONNX format into the MNN framework.

To enhance MNN's functionality for our specific requirements, certain modifications are made to its source code. These modifications include extending MNN's support for passing tensors to the step(x) function, which initially only supports scalars. In pipeline parallel training, the device may need to call step(x) with x being gradient tensors, thus necessitating this extension. To parallelly compute multiple attention heads on multiple backends, we create separate computation graphs for each backend. In these graphs, we allocate different numbers of attention heads decided by the backend scheduler. This ensures efficient parallelization of attention head computations across the backends, ultimately contributing to the overall acceleration of the training process.

# 4 EVALUATION

# 4.1 Evaluation Settings

In our preliminary evaluations, we leverage BERT for a classification task using the Conll2003 dataset [8]. We first load the pre-trained BERT weights from Pytorch following the procedure outlined in Section 3.3. Then we append a fully connected layer to the output of BERT for classification purposes. The pretrained BERT model, along with the added fully connected layer, is collectively fine-tuned using the proposed Confidant across multiple mobile devices, with their specifications detailed in Table 2.

# 4.2 Memory Usage

We compute the average memory usage across the listed three phones in Table 2 during training with Confidant and

**Table 2: Device List** 

Device	Redmi 10X Pro	Redmi K50	Mi 10 Lite	MBP
Soc	Dimensity 820	Dimensity 8100	Snapdragon 765G	M1
GPU	Mali-G57 MC5	Mali-G610	Adreno 620	M1
Memory	8G	12G	8G	16G

compare it to the memory usage when training with a single device, Redmi 10X Pro, as illustrated in Figure 5. We can observe a substantial reduction in average memory usage when employing Confidant compared to training on a single device. Note that while Confidant utilizes three mobile phones for fine-tuning, its memory usage doesn't scale linearly. This is because each phone stores multiple versions of weights and intermediate outputs of the sub-model to ensure compliance with the 1F1B (One Forward, One Backward) rule for pipeline parallel training. Nonetheless, Confidant manages to achieve a notable memory reduction of 45.3% when employing a batch size of 8.

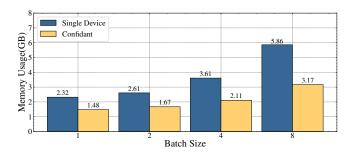


Figure 5: Memory usage comparison on BERT finetuning between Confidant and a single device.

# 4.3 Multi-backend Parallel

Next, we evaluate the performance of the proposed multi-backend parallelism. We execute the backend scheduler on a Macbook Pro, utilizing the available backends, namely CPU, OpenCL, and Metal, to collaboratively compute a self-attention layer with 12 attention heads. The backend scheduler selects CPU and Metal while omitting OpenCL due to its slower computation speed, as previously demonstrated in Figure 2. Figure 6 illustrates the latency of single-backend computations using either CPU or Metal, alongside the latency of multi-backend computations that utilize both CPU and Metal. It is clearly shown that multi-backend computations offer a significant acceleration compared to using a single backend.

# 4.4 Training Latency

Finally, we analyze the training latency by Confidant under different batch sizes, which encompasses the total time

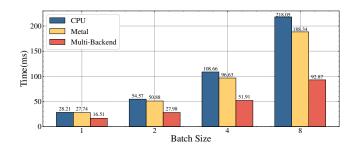


Figure 6: Computation time of a self-attention layer using only CPU, only Metal, and both.

required for both the forward and backward passes of a single batch. We compare it against three baselines here, i.e., training on a single device with a single backend, referred to as *Single*, training on a single device with mutiple backends utilizing the proposed backend scheduler, referred to as *Single-MBS*, pipeline parallel training across three devices using one backend on each, referred to as *Pipeline*.

We conduct a comparison of *Single* and *Single-MBS* using three different mobile phones individually to demonstrate the performance of the proposed multiple backend scheduler. The results are presented in Figure 7a, 7b and 7c. From the results, it is evident that by implementing the multiple backend scheduler, significant speedups of up to 4.53x, 3.97x, and 4.09x are achieved on the three respective phones.

We then compare the training latency by Confidant with that by *Pipeline*, as shown Figure 7d. The experimental results clearly demonstrate that Confidant achieves a maximum speedup of 2.09 times compared to pipeline parallel training. This outcome further confirms the effectiveness of the proposed multiple backend scheduler.

Comparing Figure 7d with Figure 7a, 7b and 7c, we can observe that the pipeline parallel training achieves a maximum speedup of 3.84x compared to training on a single phone. This validates the advantage of pipeline parallel training in accelerating the training process. Furthermore, it's worth noting that the proposed Confidant achieves remarkable speedup ratios of up to 8.03x compared to training on a single phone, underlining its significant performance gains.

### 5 DISCUSSION AND FUTURE WORK

This paper summarizes our progress and preliminary experimental results on collaborative edge training of LLMs. We intend to further explore and investigate the following aspects.

**Memory Adaptation.** Given that users may concurrently run applications during LLM training on a mobile device, the available memory for training can vary over time and even abruptly change, leading to training interruptions caused by insufficient memory. In our forthcoming research, we plan

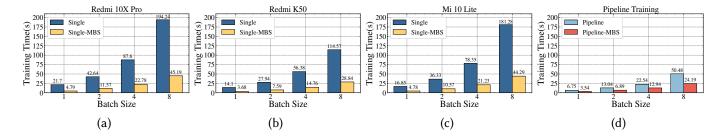


Figure 7: Training latency comparisons between single devices and our system

to develop a memory adaption technique that can effectively adapt the training process to varying memory budgets.

Fault Tolerance for Energy-aware Training. Given that many mobile devices operate on battery power, an energy-aware training algorithm could effectively improve the reliability of the fine-tuning process. We also aim to improve the reliability by designing a proper fault tolerance mechanism. This strategy involves predicting the remaining battery life and proactively relocating training workloads to devices with sufficient battery power.

Cross Framework Implementation. As previously discussed in Section 1, individuals may possess edge devices of different types, each of which supports different deep neural network (DNN) frameworks (e.g., MNN for mobile phones and Pytorch for laptops). As such, it becomes imperative to develop methods to collaboratively train a large language model (LLM) using diverse mobile DNN frameworks.

#### 6 CONCLUSION

In this paper, we introduced Confidant, a multi-backend edge collaborative training framework designed for the finetuning of transformer-based large language models (LLMs) on mobile devices. Our approach involves partitioning the LLM into multiple sub-models and distributing them across several mobile devices. We leveraged pipeline parallel training and dynamic model partitioning to expedite the training process. Additionally, we introduced a novel backend scheduler to further enhance training speed, dynamically allocating varying numbers of attention heads to multiple backends and enabling parallel computation across multiple backends within one device. We implemented Confidant on mobile devices using an industry-scale deep neural network (DNN) platform, and conducted preliminary evaluations to demonstrate its efficacy in reducing memory usage on individual devices while simultaneously accelerating the training process.

#### REFERENCES

- Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016.
   Training deep nets with sublinear memory cost. arXiv preprint arXiv:1604.06174 (2016).
- [2] Yuhao Chen, Qianqian Yang, Shibo He, Zhiguo Shi, Jiming Chen, and Mohsen Guizani. 2023. FTPipeHD: A Fault-Tolerant Pipeline-Parallel Distributed Training Approach for Heterogeneous Edge Devices. IEEE Transactions on Mobile Computing (2023), 1–13.
- [3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805 (2018).
- [4] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. Codl: efficient cpu-gpu co-execution for deep learning inference on mobile devices. In Mobisys.
- [5] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In MLSys.
- [6] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In SOSP.
- [7] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]
- [8] Erik F Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. arXiv preprint cs/0306050 (2003).
- [9] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 (2023).
- [10] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. NIPS (2017).
- [11] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: Breaking the memory wall for resource-efficient on-device machine learning. In *Mobisys*.
- [12] Xiaohui Wang, Yang Wei, Ying Xiong, Guyue Huang, Xian Qian, Yufei Ding, Mingxuan Wang, and Lei Li. 2022. Lightseq2: Accelerated training for transformer-based models on gpus. In SC.
- [13] Wikipedia contributors. 2023. Java Native Interface Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Java\_ Native\_Interface&oldid=1166267665. [Online; accessed 3-October-2023]
- [14] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023. EdgeMoE: Fast On-Device Inference of MoE-based Large Language Models. arXiv preprint arXiv:2308.14352 (2023).