Confidant: Customizing Transformer-based LLMs via Collaborative Training on Mobile Devices

ABSTRACT

Large Language Models (LLMs) have emerged as a cornerstone for advancing AI technologies. It changes the way we interact with devices, websites, and information, and paves the way for the development of highly intuitive and capable personal assistants. Training of today's LLMs happens in cloud data centers due to the requirement of enormous data sets and a significant amount of computing power. Despite extensive research in continuous learning and mobile edge computing, fine-tuning pre-trained LLMs using resource constrained devices like commodity smartphones remains highly under-explored. In this paper, we propose Confidant, a practical collaborative training framework that allows modern LLMs to be fine-tuned across multiple off-the-shelf mobile devices. To this end, Confidant firstly partitions an LLM into several sub-models, allowing each of them to fit in the memory of a mobile device. Multiple mobile devices then collaborate in training the LLM, employing a novel pipeline parallel training approach to accelerate the training process. Confidant also encompasses an inter-device dynamic model partitioning and intra-device multi-processor scheduler to minimize the training time across heterogeneous systems. To ensure resilient distributed training, Confidant incorporates a hybrid fault tolerance mechanism to proactively manage potential device and network failures. We fully implemented Confidant in C++/Python, and deployed it on a diverse range of mobile devices. Our experimental results demonstrate that Confidant excels in achieving computation-, memory-efficient, and robust customization of LLMs across mobile devices - it successfully fine-tunes billion-sized LLMs using three mobile devices, and achieves up to 5.56 times LLM training acceleration.

1 INTRODUCTION

Transformer-based large language models (LLMs), such as BERT [9], GPT [2, 27, 29], LLaMA [37], Phi2 [47], have garnered significant attention in the field of natural language processing (NLP). These advanced models are trained and typically served by the cloud due to the prohibitive cost of training and inference [11] – Meta recently has unveiled its foundational training infrastructure for LLaMA-3, which utilizes two 24k H100 GPU clusters [24], and OpenAI has paused, for several times, upgrades to ChatGPT Plus due to overwhelming inference demand and the requirement of massive amount of compute capacity.

With the increase of edge and mobile computing power, *on-device model training* becomes popular in the era of deep learning. On-device model training allows a pre-trained model to learn domain- and user-specific knowledge, and ensures that personal and sensitive data is processed without leaving the local network. It harnesses provisioned but not actively utilized local computing resources, lowering the prices of running AI applications. It is also less dependent on network connectivity, providing a lower inference latency and higher reliability. With the unparalleled advancements in generative AI, in this paper, we seek to answer the question that *how off-the-shelf mobile personal devices can work collectively to fine-tune modern LLMs?*

We list two motivating scenarios that could greatly benefit from such systems. (1) *Personal and Family Chat Assistant.* Many of us nowadays possess multiple mobile devices, including smartphones, pads and laptops. Collaborative fine-tuning LLMs using private data and these mobile devices would allow us to have personal/family assistants accessible exclusively to family members within the home network, and revolutionize the way we interact with technology, promising a future where smart living is not just integrated but deeply personalized to the unique dynamics of each household. (2) *Event Chat Assistant.* Domain-specific LLMs play a pivotal role in elevating user experience and customer interactions. Imagine you are organizing a large event like academic conferences. It would be extremely helpful to the participants if they could query near real-time event-related information (e.g., a summary of a talk given earlier today) in natural language. Such domain-specific LLMs can possibly be customized (e.g., via the event mobile app) by a pool of devices of the event participants.

Compared with on-device training of traditional deep neural networks (DNN) like convolutional neural networks (CNN) [4, 12, 39] and recurrent neural networks (RNN) [17, 38], local training of transformer-based LLMs has been much less studied. In fact, executing large language models on modern mobile hardware and software platforms poses significant challenges, and there are currently no solutions readily available for LLMs training on mobile devices like smartphones. The reasons are threefold. First, both training and inference of an LLM require way more memory than traditional neural networks. As will be shown in Section 2, it is not always feasible to fit in modern LLMs, even after partitioning, on commodity mobile devices. Second, despite the prevalence of mobile accelerators, platform and software

support for LLM training on mobile hardware is still lacking. On one hand, key operators in modern LLMs like LayerNorm and Embedding are not yet supported by mobile deep learning frameworks (e.g., MNN [16]). On the other hand, a large number of mobile devices like smartphones do not have CUDA-capable GPUs, and OpenCL, an alternative computing library that mobile GPUs often use, provides much higher computation latency on par with mobile CPUs (as shown in Section 2.1). The heterogeneity of mobile SoCs only exacerbates the problem. Third, training across a group of mobile devices is more prone to failures, and a fault tolerant training mechanism tailored for mobile devices is yet to be designed. Existing fault tolerance studies [4, 21] either focus on robust convergence in data-parallel training (e.g., federated learning), which does not apply to mobile LLM training where memory is the bottleneck, or deal with failures in a passive way, which inevitably adds up the recovery delay.

To address these challenges, we introduce **Confidant**, a collaborative training framework that enables LLM fine-tuning on various mobile devices including smartphones and laptops. In a nutshell, Confidant partitions an LLM into several sub-models, and deploys them across multiple mobile devices to collaboratively fine-tune an LLM using a novel pipeline parallel training approach. To cope with device heterogeneity, Confidant introduces a memory-aware dynamic model partitioning technique, which estimates the real-time memory and computing capacity of participating devices, and partitions and places sub-models accordingly to minimize the total training time while satisfying the memory constraints on each device. Based on the parallel nature of attention heads in LLMs, we propose a novel intra-device multi-processor scheduler that can optimally allocate different numbers of attention heads to heterogeneous mobile processors. To reduce recovery delay when faults happen during training, Confidant employs a hybrid fault tolerance mechanism that combines legacy passive fault tolerance design with a novel proactive adaptation method, which leverages device dynamics to preemptively address types of training interruptions led by device mobility, network failure or power depletion. Last but not least, we develop a cross-framework adapter in C++ and Python to enable collaborative training across devices utilizing different DNN frameworks. We deploy Confidant and evaluate its performance across various mobile devices including mobile phones and laptops. Experiment results demonstrate that Confidant achieves fast, memory-efficient, highly robust, and widely applicable edge collaborative training of LLMs.

We summarize our main contributions as follows:

- We propose a collaborative edge training framework called Confidant that uses pipeline parallel training to fine-tune LLMs across multiple edge devices using different DNN frameworks. To the best of our knowledge, this marks the first implementation of collaborative training for Transformer-based LLMs on edge devices, including mobile devices.
- We propose the memory-aware dynamic model partitioning and the novel intra-device multi-processor scheduler to satisfy the memory constraints of the participating devices while speeding up the collaborative edge training. We also enable collaborative training across DNN frameworks through a cross-framework adapter.
- We introduce a hybrid fault tolerance mechanism that consists of both the passive fault tolerance method and a novel proactive fault tolerance method, which jointly provide a more robust fault-tolerance capability.
- We deploy Confidant on a set of commonly used mobile devices and comprehensively evaluate its performance. Evaluation results prove the efficiency in time and memory, high robustness, and broad applicability of Confidant, where 1.13 to 5.56 times acceleration is achieved in the experimental settings. We also fine-tune billion-sized LLMs using only three mobile devices.

2 BACKGROUND AND MOTIVATION

We begin by characterizing the execution of LLMs on mobile devices in Section 2.1. Afterwards we introduce pipeline parallel training and the problems of directly applying it to fine-tuning LLMs on mobile devices in Section 2.2.

2.1 Executing LLMs on Mobile Devices

We first motivate the design of Confidant by examining the characteristics of executing LLMs on mobile devices.

(1) *Excessive Memory Requirements of LLMs vs. Limited Memory on Mobile Devices.* Compared to data center servers and high-end edge devices like NVIDIA Jetson nodes, commodity mobile devices (e.g., smartphones) have much less on-board memory, making LLM fine-tuning challenging. We conducted a preliminary experiment to show the memory usage of fine-tuning (with batch size set to 8) four state-of-the-art LLMs using PyTorch. We also list the parameter sizes of these LLMs, and the on-board memory of four recently released smartphones for comparison. It can be seen from Table 1 that considering the memory taken by mobile OS and apps, current mobile devices can hardly meet the memory demand for LLM fine-tuning.

(2) *The Need of Collaborative Training vs. Highly Dynamic Compute, Memory and Energy Supplies.* The excessive memory requirements of LLM fine-tuning calls for a collaborative

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

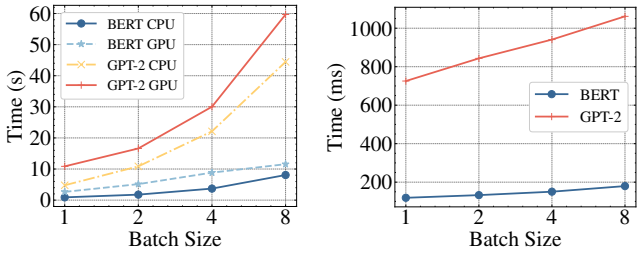
LLMs	Param.	Mem. Usage	Mobile Devices	Mem.
BERT-Base	110M	5.8GB	Huawei Mate 60 Pro	12GB
GPT-2-Medium	355M	18.5GB	Google Pixel 8 Pro	12GB
Phi-2	2B	46.5GB	iPhone 15 Pro	8GB
LLaMA-7B	7B	58.3GB	Galaxy S23 Ultra	12GB

way to train LLMs with a group of mobile devices. Nonetheless, the highly dynamic compute, memory and energy supplies of mobile devices make the design of a collaborative training mechanism extremely difficult. Unlike enterprise-grade servers, the majority of mobile devices share their memory between CPU and GPU [15], and could be subject to energy depletion and network disconnection due to the nature of mobility. All of these factors need to be considered in the design of a collaborative training framework.

(3) *Insignificant Differences in LLM computation Latency between Mobile CPUs and GPUs.* While programming models and platforms like CUDA [26] significantly facilitate GPU acceleration by leveraging its parallel processing capabilities, non-CUDA-capable mobile GPUs exhibit a similar or even higher LLM computation latency than traditional mobile CPU-centric approaches. To illustrate this, we conducted CPU-only and GPU-only experiments on a Redmi K50, and measured the compute time of one batch forward propagation under different batch sizes for two models: BERT-base and GPT2-medium, respectively. The experiments are based on Mobile Neural Network (MNN) [16], a state-of-the-art mobile deep learning framework. Note that we use OpenCL as the computing library for GPU since CUDA is not supported on the mobile platform. As demonstrated in Figure 1a, the compute latency employing CPU and GPU are of the same order of magnitude. We have also measured the overhead of copying tensors from CPU to GPU when GPU is used for computation. As shown in Figure 1a, the copy overhead is negligible compared to forward computation time in Figure 1a. This suggests that one should leverage multiple processors including both CPUs and GPUs on a mobile device for parallel computation of LLM, thereby enhancing training efficiency. Unfortunately, today's mobile deep learning frameworks only support DNN *training* on one kind of processor.

2.2 Pipeline Model Parallel Training

To address the issue of insufficient memory on a single mobile device for fine-tuning LLMs, one common solution is to apply pipeline model parallel training to multiple mobile devices for collaborative model training [4]. Pipeline model parallel training was originally introduced for training CNN models on GPU clusters [25]. Since then, several works [5, 22, 28]



(a) Forward compute latency (b) Tensor copy overhead

Figure 1: The copy overhead and computation overhead of forwarding BERT-base and GPT2-medium separately on Redmi K50.

have applied this technique to training LLMs on GPU clusters since the memory of a single GPU may also be insufficient for training an LLM. This approach optimally divides a DNN model into multiple sub-models based on the compute capacities of GPU servers, and deploys each sub-model onto a separate GPU node. During training, each GPU sequentially processes a batch through forward and backward computations. To expedite the training process, GPUs forward a batch with stale sub-model weights, eliminating the need to wait for weight updates from the backward pass. To ensure model convergence, two techniques are introduced: the one-forward-one-backward (1F1B) rule and weight stashing. The 1F1B rule dictates that forward and backward pass should alternate, with consecutive operations executed on different batches. Weight stashing involves storing the weights used in forward pass so as to guarantee consistent weights during backward propagation of the same batch.

Figure 2: An illustration of the pipeline parallel training technique.

Figure 2 illustrates the concept of pipeline parallel training using three devices as an example, where device 1 is the central node that owns the raw data, and other two are *worker nodes*. i_{f}^{ver} and i_{b}^{ver} represent the forward and backward pass of the i-th training batch with weights of the version number ver, respectively. Consider the training of batch 1 in device 1, instead of waiting for the backward pass of batch 0, device 1 forwards batch 1 using the same weights as those used by batch 0. Following the 1F1B rule, device 2 forwards batch 3 after backward pass of batch 1, immediately followed by

the backward pass of batch 2. For weight stashing, when device 2 backwards batch 2, instead of using the weights of version 2, it uses the weights with version 0, which are the same weights utilized in the forward pass of batch 2. It can be observed from Figure 2 that pipeline parallel training significantly reduces the idle time of each device, hence speeding up the training process. Despite of active research on pipeline model parallel training, there is no viable solution at this moment that allows a set of off-the-shelf mobile devices to collectively train LLMs.

3 SYSTEM DESIGN

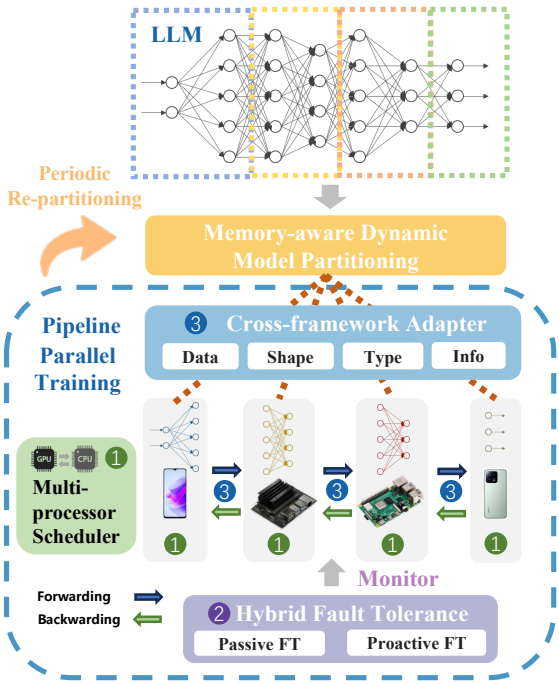


Figure 3: The system overview of the proposed Confidant

3.1 Design Overview

Figure 3 provides an overview of the proposed Confidant, designed for distributed pipeline parallel training to collaboratively fine-tune an LLM across heterogeneous edge devices. The *central node*, responsible for training, is the device who may own the training data or have the most significant computational resources, such as a desktop in a household environment. The distributed training process starts by searching for available devices, denoted as *worker nodes*, within the local area network (LAN). Then, the central node optimally divides an LLM into several sub-models according to the computing, memory and communication resources of each device, utilizing the proposed Memory-aware Dynamic Model Partitioning scheme, which will be introduced in Section 3.2. This scheme is executed periodically throughout the entire distributed training process to adapt to the time-varying resources of edge devices. To address compatibility challenges among different DNN frameworks, such as MNN and PyTorch, Confidant incorporates a Cross-framework Adapter, enabling collaborative training across various devices. Each participating device also exploits a novel Intra-device Multi-processor Scheduler, making full use of the device's computing resources by optimally scheduling available processors for parallel computation of attention heads. Additionally, Confidant integrates a Hybrid Fault Tolerance Mechanism that combines both proactive fault tolerance (Proactive FT) and passive fault tolerance (Passive FT) methods, to combat potential faults during the training, enhancing the system's resilience to unforeseen issues.

3.2 Memory-aware Dynamic Model Partitioning

To avoid assigned sub-model exceeding the available memory limit of participating devices, we present a memory-aware dynamic model partitioning in this section, which calculates the real-time optimal partition points by jointly considering the time-varying computing capacity, memory limits, and bandwidth of each participating device.

We first estimate the memory usage (GB) of each device $M_{\text{tot},i}$ by using the following formula,

$$M_{\text{tot},i} = M_{\text{model}} + M_{\text{grad}} + M_{\text{pipe},i} + M_{\text{act},i}, \tag{1}$$

where M_{model} equals the memory usage of parameters in the local sub-model and M_{grad} represents the memory usage of temporary tensors that need to be stored during gradient updates. Here i represents the i-th device in the pipeline parallel training following the computing order. $M_{\text{pipe},i}$ is the memory usage introduced by the weight stashing in the pipeline parallel training. Since for a pipeline parallel training with N devices, there are $(N-i-1)$ extra versions of trainable weights stored in memory, $M_{\text{pipe},i}$ equals $(N-i-1)$ times the memory usage of trainable parameters for the i-th device. $M_{\text{act},i}$ denotes the memory used for storing intermediate activation values during the forward and backward.

Note that the embedding layer is generally not updated in fine-tuning LLMs. M_{act} can be approximated by the total memory usage of the activation in the encoders and decoders, given by [20],

$$M_{\text{act},i} = (N-i) \times sbh(18 + 8\frac{h_{\text{f}}}{h} + 5\frac{as}{h}) \times C, \tag{2}$$

where a, b, s, h, and h_{f} represent the number of attention heads, the batch size, the sequence length, the hidden dimension size, and feed-forward dimension size respectively. Here C is a normalization factor that converts the unit into GB.

Then during the training, the central node periodically collects the time required to compute the local sub-LLM, the memory constraint, and the bandwidth between any two nodes from each worker node. Following [4], the optimal model partition points are computed iteratively by

$$A(l,n) = \min_{1\le p<l} \max \begin{cases} A(p,n-1) \\ 2 \times T_{\mathrm{c},p}^{n-2} \\ T_{\mathrm{e}}^{n-1}(p+1,l) \end{cases} \quad \forall l \in [n:L], \quad (3)$$

for $n = 1, ..., N$ and $l = 1, ..., L$, where $A(l,n)$ denotes the time used by the slowest device among n devices to collaboratively train a l-layer model in the optimal pipeline-parallel training manner. For the k-th node, $T_{\mathrm{e}}^{k}(i,j)$ indicates the training time from layer i to layer j and $T_{\mathrm{c},i}^{k}$ denotes the time of transmitting the output of layer i to the $(k+1)$-th node. To exclude model partitioning points that do not meet the devices' memory constraint, we compare the i-th node's memory constraints with the estimated memory usage of the sub-model assigned to it. If the memory constraint is not satisfied, we set T_e^i to infinity, which ensures that such partition points will not be chosen as the optimal solution.

3.3 Intra-device Multi-processor Scheduler

In this section, we introduce a novel intra-device multi-processor scheduler that optimally assigns varying numbers of attention heads to the available processors on a device. While existing work such as Megatron[33] has leveraged parallel computation of attention heads to accelerate the computation of LLMs in GPU clusters, it primarily targets homogeneous GPUs. However, distributing attention heads evenly across multiple heterogeneous processors on mobile devices may not minimize computation time efficiently. The proposed scheduler takes into account the computing capacity of heterogeneous processors by operating two steps: processor profiling and attention head allocation, which will be detailed in the following.

Processor Profiling. This step starts by identifying all available processors on a device. For each processor, we then profile the computation time for varying numbers of attention heads. Note that a processor may have multiple compute libraries supporting DNN computations, such as OpenCL and Vulkan for mobile GPUs. In such case, we select the fastest computation time by available libraries as the computation time for that processor. We denote the profiling result of computing k attention heads with the processor j as P_j^k. All P_j^k values collectively form a profiling dataset $\mathbb{P}$ for use in the subsequent attention head allocation step.

Attention Heads Allocation. This step utilizes a binary search approach to allocate attention heads to each heterogeneous processor on a device to minimize the overall computation time, as illustrated in Algorithm 1.

Assume that our optimization goal is to allocate K attention heads to M available processor to minimize their computation time. We initiate the lower bound l to 0 and the upper bound r to the minimum time for any processor to compute all K attention heads, serving as the starting point for the binary search. We also initialize a global allocation vector $\mathbb{S}$ to store the allocation strategies discovered during this process. In each iteration, we calculate the mid-value as $mid = (l+r)/2$. We then check the feasibility of allocating attention heads such that the overall computation time is close to mid. Particularly, we initialize a empty allocation vector $\mathbb{S}'$ for the current validation. For each processor j, we find the k in $[1,K]$ with P_j^k closest to mid, denoted as O_j. If the absolute difference between the selected P_j^k and mid exceeds a given threshold ϵ, we set $O_j = 0$, indicating that we skip the processor computing either too fast or too slow. We then include the pair (j, O_j) in $\mathbb{S}'$. If the sum of all O_j is greater than or equal to K, it implies the existence of an allocation strategy where the overall computation time is approximately mid. In this case, we set $\mathbb{S} = \mathbb{S}'$ and adjust the upper bound r of the binary search to $mid - \sigma$, where σ is a relatively small value to prevent infinite looping. If the sum of all O_j is less than K, it implies that computing all attention heads within mid time is unattainable. In such cases, we adjust the lower bound l to mid - σ. The binary search iteration concludes when $l > r$. Finally, attention heads are distributed to the selected processors according to $\mathbb{S}$. These attention heads are then computed in parallel, thereby expediting the computation of Transformer-based LLMs on edge devices.

3.4 Hybrid Fault Tolerance Mechanism

In this section, we present the proposed hybrid fault tolerance mechanism which integrates both passive and proactive fault tolerance methods, offering a complementary and comprehensive fault tolerance capability.

3.4.1 Passive Fault Tolerance. Passive fault tolerance, initially proposed in [4, 21], requires each device to replicate the weights of its local sub-model to other nodes periodically.

When a failure occurs, the central node identifies the failed nodes through broadcasting and initiates the three-phase recovery process to restore training. Firstly, in the model repartitioning phase, the central node redistributes the model among the remaining nodes. Subsequently, in the weight redistribution phase, each surviving node retrieves the weights of the new sub-model from local replication or other nodes. Finally, in the commit phase, each node constructs the new local sub-model, and the training resumes. Although passive fault tolerance effectively addresses various faults during collaborative training, it introduces some time overhead,

Algorithm 1: Allocation of attention heads on multiple processors

Input: The profiling set $\mathbb{P}$, total attention heads K, total available processors M, threshold ϵ.
Output: The allocation strategy $\mathbb{S} = \{(j, k_j) | j = 1, ..., M, k_j = 0, 1, ..., K\}$.

Initialize $l \leftarrow 0$, $r \leftarrow \min_{j=1,...M} P_j^K$, $\mathbb{S} \leftarrow \{\}$;
while $l \leq r$ **do**
  $mid \leftarrow (l + r)/2$;
  if $isValid(mid, K, \mathbb{P})$ **then**
    $r \leftarrow mid - \sigma$;
  else
    $l \leftarrow mid + \sigma$;
  end
end
return $\mathbb{S}$;

Function $isValid(mid, K, \mathbb{P})$:
  Initialize $totalHeads \leftarrow 0$, $\mathbb{S}' \leftarrow \{\}$;
  for $j \leftarrow 1$ **to** M **do**
    $k_j = \arg\min_{k=1,...K} abs(P_j^k - mid)$;
    if $abs(P_j^{k_j} - mid) > \epsilon$ **then**
      $k_j \leftarrow 0$;
    end
    $totalHeads \leftarrow totalHeads + k_j$;
    insert (j, k_j) into S';
  end
  if $totalHeads >= K$ **then**
    $\mathbb{S} \leftarrow \mathbb{S}'$;
    return $true$;
  else
    return $false$;
  end

including the time to detect the fault (detection time), redistribute the weights (redistribution time), and retrain the batches that have been forwarded but not yet backwarded when the fault occurred (retrain time).

3.4.2 Proactive Fault Tolerance. To reduce the time overhead in addressing node failures, we further introduce a novel proactive fault tolerance mechanism, which comprises three phases, i.e., the notification phase, the proactive substitution phase, and the commit phase.

Notification Phase. When a worker node is about to quit the training due to its device dynamics, referred to as the *quitting node* d_q, it notifies the central node at least α seconds before quitting. To eliminate the time overhead for recovering, the value of α should be greater than or equal to the time needed to complete the Proactive Substitution Phase, as described later.

Proactive Substitution Phase. Upon receiving the notification from d_q, the central node initiates the collection of computing capacities from all idle devices in the LAN. If no idle devices are available, the central node resorts to the passive fault tolerance mechanism. However, if there are idle devices, the central node proceeds to identify a *substitute node* that is the most compatible. To determine compatibility, each idle device sends back the *computing capacity vector (CCV)* $\boldsymbol{h}_i$ and the *remaining battery* b_i (mAh) to the central node. Specifically, $\boldsymbol{h}_i \triangleq \{t_{i,1}, t_{i,2}, ..., t_{i,L}\}$, where $t_{i,j}$ denotes the time required for computing j encoders, and L is the total number of encoders. Including different encoder numbers is essential because $t_{i,j}$ is not linear in j due to computation optimization in DNN frameworks [46]. The remaining battery b_i is a value between 0 and 1, where a higher value indicates a greater remaining battery capacity.

Using $\boldsymbol{h}_i$ and b_i, the central node adopts the metric *Device Compatibility (DC)* to assess the compatibility of an idle device:

$$DC_i = \frac{p * \hat{b}_i}{\hat{H}_i + \eta}, \tag{4}$$

where p denotes the portion of the remaining training process, which equals $(B_r + (T - T_{cur}) * B)/T * B$. Here, T, B, T_{cur} and B_r denote the total epochs, the total batches, the current epoch, and the remaining batches of the current epoch when the proactive fault handler is triggered, respectively. η is a relatively small value to avoid division by 0. $\hat{H}_i$ is derived by first summing up all values in $\boldsymbol{h}_i$ to H_i and normalized by $\hat{H}_i = (H_i - H_{min})/(H_{max} - H_{min})$, where H_{max} and H_{min} are the maximum value and the minimum value among all H_i. Similarly, b_i is normalized to $\hat{b}_i$ by $(b_i - b_{min})/(b_{max} - b_{min})$. As it can be seen from Eq. 4, the selection of substitute node considers both computing capacity and the remaining battery of the device to ensure training stability. However, as training progresses, i.e., with the decrease of p, attention to the device battery diminishes. The central node selects the device with the highest DC as the substitute node, which then retrieves all weights from the sub-model of d_q.

Commit Phase. The substitute node creates the corresponding sub-model and notifies the central node. Subsequently, the quitting node exits the training without interrupting the training process. The central node then broadcasts a commit message to all worker nodes, informing them of the change in one worker node. Finally, all the nodes retrain the batches whose training was interrupted during the proactive fault handling, resuming the training process.

3.5 Cross-framework Adapter

Edge devices may utilize various DNN frameworks, resulting in different data formats for the intermediate results of DNN. To address this, Confidant integrates a **Cross-framework Adapter** designed to convert intermediate results output by different devices into a unified structure compatible with each other, which comprises the following four components:

- **Data**: A one-dimensional array storing all the data of a tensor, acquired by flattening the tensor.
- **Tensor Shape**: A one-dimensional array that characterizes the shape of a tensor, e.g., [8, 256, 4096].
- **Data Type**: An integer indicating the type of the data, e.g., int8, int32 and float64.

- **Framework Specific Info**: A map storing specific information required by different frameworks. One instance can be $\{MNN : \text{“}NHWC\text{”}\}$, which includes the information needed by the MNN framework.

Before transmission to the next device, this adapter converts tensors into the described four components. Upon receipt, these components are then converted back into tensors suitable for the framework used by the receiving device.

Table 2: Device Specifications

Device	SoC	CPU	GPU	Memory
Redmi K50	Dimensity 8100	Cortex-A78	Mali-G610	12GB
Redmi 10X Pro	Dimensity 820	Cortex-A76	Mail-G57 MC5	8G
Mi 10 Lite	Snapdragon 765G	Cortex-A76	Adreno 620	8G
Huawei P30	Kirin 980	Cortex-A76	Mali G76 MP10	8G
Samsung A9	Snapdragon 660	Kryo 260 Gold	Adreno 512	4G
Colorful X15-AT	–	Core i7-13650HX	–	32G
Macbook Pro	M1	M1	Apple-designed	16G
Jetson Nano	——	ARM Cortex-A57	Nvidia Maxwall	4G

4 IMPLEMENTATION

We have implemented and deployed Confidant on various devices with about **10,900** lines of code (LoC) in total. For mobile devices, we develop Confidant as an application (C++: ~**3,200** and Java: ~ **3,600** lines of code (LoC)). MNN[16] version 2.7.2 serves as the deep learning framework within the Android application. We choose MNN for its support for training on mobile devices and its superior performance in terms of computation time and memory usage, as demonstrated in [39]. Given that MNN is implemented in C++ while our application is developed in Java, we utilize the Java Native Interface (JNI) [42] to invoke C++ functions from within the application.

Our work is the first attempt to implement the Transformer model for training within the MNN framework, with plans for the source code to be open-sourced in the future. The current approach for deploying Transformer-based LLMs on MNN involves converting a PyTorch-formatted model into an MNN-formatted one. However, the derived MNN-formatted model is limited to inference and lacks partitioning capabilities. Hence, we reimplement key operations in LLM, such as LayerNorm, RMSNorm, Embedding, and complex number computations, utilizing fundamental MNN operators. We then implement LLM models with these operations, which are compatible with training and model partitioning.

Loading pre-trained weights of an LLM from the corresponding PyTorch model involves the following steps: We first convert the PyTorch-formatted pre-trained weights into the ONNX format [8], an open standard for representing machine learning models. Then we employ the MNNConvert tool provided by MNN to convert the ONNX-formatted weights into the MNN format, which can then be loaded into the MNN-formatted model.

To custom MNN to our specific requirements, we make modifications to its source code. In pipeline parallel training, the device may need to call $step(x)$ with x being gradient tensors. Thus, the first modification we made is to extend MNN's support for passing tensors to the $step(x)$ function, which initially only supported scalars. Additionally, to facilitate parallel computation of multiple attention heads across multiple processors, we create a separate computation graph for each processor. Within these graphs, we allocate different numbers of attention heads as determined by the proposed multi-processor scheduler, thus enabling parallel computation across multiple processors.

For edge devices that support training with PyTorch, we execute Confidant as a Python program (Python: ~ 4,100 LoC) using PyTorch version 2.0.1. We directly load pre-trained PyTorch-formatted weights available online into these devices.

To facilitate data transmission between devices utilizing different DNN frameworks, we rely on standard HTTP requests and lightweight web frameworks to handle communcation requests from other devices. We implement with Java Spark [7] for the mobile application and Flask [6] for the Python program, respectively. To implement the proposed cross-framework adapter, we encapsulate the four data components using JSON (JavaScript Object Notation).

5 EVALUATIONS

5.1 Experiment Settings

In this section, we evaluate the proposed Confidant on various commercially available edge devices, including mobile devices, with detailed specifications provided in Table 2. Our evaluations cover three representative LLMs: BERT [9], GPT2 [29], Phi2-2.7B [47] and LLaMA-7B [37], representing Transformer-based LLMs of varying sizes. We consider fine-tuning tasks such as classification using the Conll2003 [31] dataset for BERT, question answering with the SQuAD2.0 [30] dataset for GPT2 and GPT2-Medium, and text generation employing the Alpaca [36] dataset for Phi2-2.7B and LLaMA-7B. Table 3 summarizes the specifics of these LLMs and their corresponding datasets, where #Blocks, #Heads and #Embedding Size represent the total number of encoder or decoder blocks, the number of self-attention heads in each block, and the feature dimension for each token in the input sentence, respectively. Note that all evaluations in this section are performed with no other user processes or user applications running in the background on the participating mobile devices.

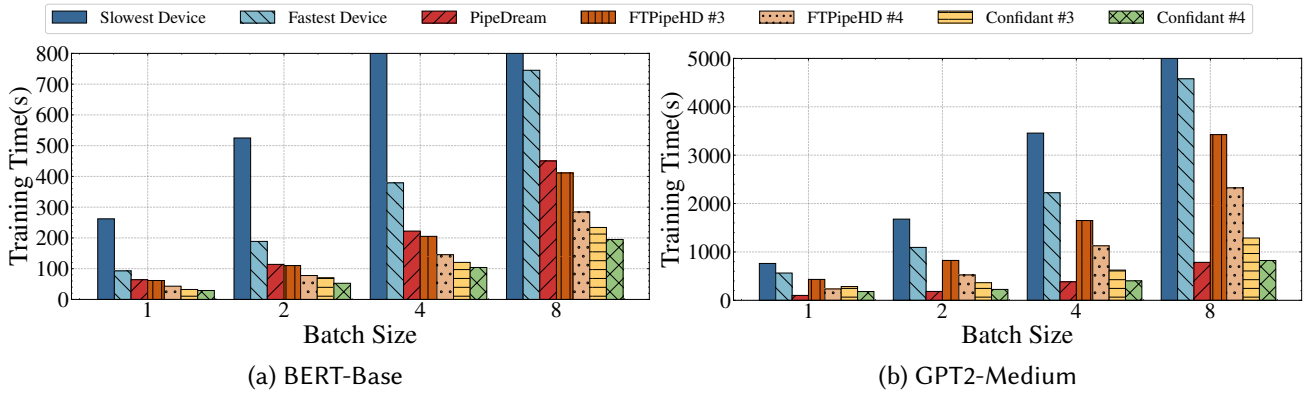


Figure 4: The training time comparisons using different LLMs under different batch sizes. "#N": pipeline parallel training using N device.

Table 3: Model Details and Corresponding Dataset

Model	#Blocks	#Heads	#Embedding Size	Dataset
BERT-Base	12	12	768	Conll2003
GPT2-Medium	24	16	1024	SQuAD2.0
Phi2	32	32	2560	Alpaca
LLaMA-7B	32	32	4096	Alpaca

Table 4: Evaluation Device Setting

Model	Confidant #3	Confidant #4
BERT-Base	Redmi K50 Mi 10 Lite Redmi 10X Pro	Redmi K50, Mi 10 Lite Redmi 10X Pro, Huawei P30
GPT2-Medium		

5.2 Training Performance

In this section, we evaluate the training performance of Confidant executing the fine-tuning tasks described above, in terms of the training time and memory usage.

5.2.1 Model Performance. We first evaluate the model performance of using Confidant to fine-tune an LLM, where we use the model performance of fine-tuning an LLM in a traditional way as the baseline. For both fine-tuning ways, we set the total epoch to 3. To reduce the experiment time, we conduct this experiment on a server and the results are listed in Table 5, where we utilize 3 independent processes to simulate three devices in Confidant. Due to limitations in server resources, we employ LoRA [13], a parameter-efficient fine-tuning technique, for Phi2 and do not validate the performance of LLaMA. Note that for different datasets, we adopt different evaluation metrics, which are provided in the footnotes of Table 5. It can observed from Table 5 that Confidant almost does not cause any accuracy drop, and even performs slightly better than traditional training, thus guaranteeing the model performance of Confidant.

Table 5: Model Performance Comparisons

Task	Traditional	Confidant
BERT-Base + Conll2003[1]	97.70%	97.66%
GPT2-Medium + SQuAD2.0[2]	72.55%/82.23%	73.03%/83.05%
Phi2 + Alpaca[3]	0.52	0.66

[1] Token classification accuracy
[2] Exact match / F1 score
[3] Test dataset loss

5.2.2 Training Time. Then, we measure the total time of forwarding and backwarding ten batches considering different batch sizes. The number of mobile devices participating in Confidant is set to 3 or 4, as listed in Table 4. For comparisons, we consider three baseline scenarios: (1) Fine-tuning an LLM on a single device, (2) Conventional pipeline parallel training by average partitioning of the LLM (PipeDream [25]) and (3) Conventional pipeline parallel training using computing capacity-aware model partitioning (FTPipeHD [4]). In baseline (1), we select the mobile devices with the highest and lowest computing capacities from Table 2 as the experiment devices. In cases where a device's memory is not sufficient for the entire model, we adopt an incremental measurement approach, where during both the forward and backward passes, we load only a portion of the model that fits into memory for computation, releasing it before loading the next segment. This incremental measurement approach is consistently applied in subsequent evaluations as well. For baseline (2) and (3), we choose the same devices employed by Confidant. We record the average time for training 10 batches using the three baselines and Confidant. The results are presented in Figure 4.

As shown in Figure 4, the conventional pipeline parallel training approach leverages the resources of multiple devices to accelerate training. The training time decreases with the increasing number of participating devices. Compared to training using only the fastest device, pipeline parallel training with three devices achieves acceleration ratios of 1.81x and 1.33x on BERT-Base and GPT2-Medium respectively. When using four devices, the acceleration ratios further increase to 2.62x and 1.97x.

Confidant significantly accelerates the collaborative training compared to FTPipeHD. With three participating mobile devices, Confidant achieves acceleration ratios of 1.75x and 2.66x on BERT-Base and GPT2-Medium, respectively. When four devices are used, it achieves acceleration ratios of 1.45x and 2.82x. Compared to training solely on the fastest device, Confidant achieves acceleration ratios of 3.18x and 3.55x under three-device configurations. For the four-device configuration, it demonstrates acceleration ratios of 3.82x and 5.56x. These results affirm Confidant's remarkable acceleration performance for training on mobile devices.

We also note that the transmission of the output of the sub-model on the current batch and the computation of the next batch occur simultaneously by two threads on a mobile device. Our evaluations show that the transmission time is only 8% to 23% of the computation time when fine-tuning BERT. Therefore, the transmission overhead is completely covered by the computation of the model and, hence, can be ignored.

5.2.3 Memory Usage. Next, we assess the memory usage of Confidant, with the memory usage for fine-tuning on a single mobile device as the baseline. We select three representative mobile devices with available memory of 4G, 8G and 12G, respectively, reflecting common memory scenarios in mainstream smartphones. In cases where the memory of a single device may be insufficient for LLM fine-tuning, we record the memory consumption when accommodating the maximum number of encoders. Additionally, we measure the average memory usage across devices during fine-tuning with both three and four devices under Confidant. The evaluation results are detailed in Table 6.

Table 6 illustrates that fine-tuning a complete LLM on a single device becomes increasingly challenging as the model size grows. Even the mobile device with the largest memory, i.e., 12GB, can only accommodate a limited number of GPT2-Medium encoders. Moreover, the idle memory of a mobile device in daily usage is unlikely to exceed 50% of the total memory, with a portion of this memory reserved for temporary mobile applications. Consequently, the available resources for fine-tuning LLMs on mobile devices are further constrained. Confidant effectively reduces the average memory usage on each device through model partitioning-based collaborative training. This allows devices with smaller memory capacities to participate in training by handling a subset of computations for the entire model. As the number of devices increases, the average memory usage per device decreases.

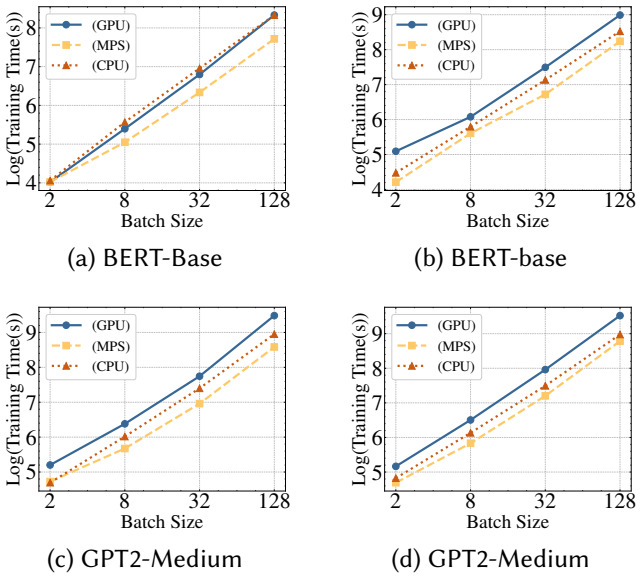


Figure 5: The computation latency of the self-attention layer of two models under various batch sizes. Left column: Redmi K50, Right column: Redmi 10X Pro.

5.3 Multi-processor Scheduler Acceleration

In this section, we assess the performance of the intra-device multi-processor scheduler (MPS) by comparing it with the scenarios where only the CPU or GPU (OpenCL) is utilized. Note that in this evaluation, when computing using only one processor, we select the best approach among the three methods described in Section 3.3 for computing attention heads.

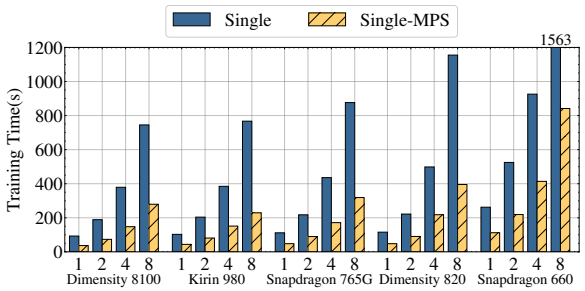
We first compare the computation time of the self-attention layer for BERT-Base and GPT2-Medium, as depicted in Figure 5. Due to the significant difference in computation time between different batch sizes, we adopt the logarithmic scale for the vertical axis. When working with small batch sizes, the computation time using MPS is comparable to using either only the CPU or GPU. However, as the batch size increases, the acceleration effect of MPS becomes more evident, indicating that the proposed MPS effectively leverages processors for LLM computation.

We also compare the computation time of the entire BERT-Base and GPT2-Medium using the MPS against using only

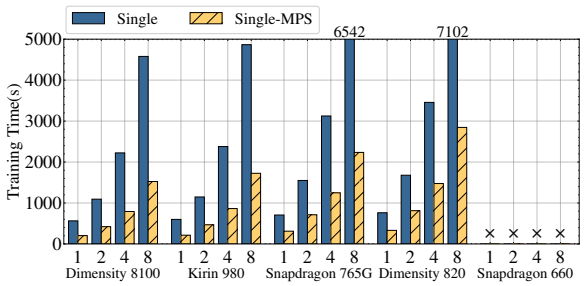
Table 6: Memory usage comparisons between Confidant and a single device

Model	BatchSize	Memory Usage (GB)				
		Samsung A9	Mi 10 Lite	Redmi K50	Confidant #3 (Average)	Confidant #4 (Average)
BERT-Base	4	2.7(**10/12**)[1]	3.1	3.1	2.10	1.65
	8	2.8(**6/12**)	5.5	5.5	3.17	2.43
GPT2-Medium	1	2.8(**10/24**)	6.2	6.2	4.20	3.17
	2	2.8(**8/24**)	7.0(**16/24**)	7.2	5.13	3.83
	4	2.7(**5/24**)	6.9(**12/24**)	9.8	6.57	4.73
	8	2.6(**2/24**)	6.9(**5/24**)	9.8(**20/24**)	7.20	5.81

[1] (**i/j**): Only accommodate a maximum of **i** encoders while current model comprises **j** encoders



(a) BERT-Base



(b) GPT2-Medium

Figure 6: Training latency comparisons between single-device fine-tuning and MPS-based fine-tuning

the CPU, where we use two kinds of CPU to perform the comparisons. The results are presented in Figure 6, where × indicates that it is impractical to conduct experiments on the specified device. The results highlight the significant acceleration achieved by our proposed MPS in complete model training. Furthermore, the MPS demonstrates versatility and exceptional performance across different models and devices.

5.4 Hybrid Fault Tolerance Overhead

We then evaluate the performance of the proposed hybrid fault tolerance mechanism. We consider training with three mobile devices, along with two idle devices in LAN. We evaluate the passive and proactive fault tolerance methods independently.

We trigger the passive FT by manually closing the Confidant application on a worker node and measure the detection time, weight redistribution time and retrain time as described in Section 3.4.1. For proactive FT, we trigger it by setting a worker node to notify the central node of its exit after Confidant has been running for a specified duration. We then measure the time taken to find a substitute node (search time), the time for the substitute node to retrieve all weights from the quitting node (substitution time), and retrain time. For both fault tolerance methods, we record the time required for training to resume after being interrupted (recovery time). The evaluation results are detailed in Table 7.

The results presented in Table 7 reveal that the detection time in passive FT is greater than the search time in proactive FT. While the weight redistribution time in passive FT is shorter than the substitution time in proactive FT, it is important to note that weight redistribution in passive FT occurs during the training interruption, whereas substitution in proactive FT is carried out concurrently with training. The retrain times for both methods are relatively similar. We highlight that the recovery time for proactive FT is significantly shorter than that for passive FT. This indicates that the advantages of the proposed proactive FT in reducing the time overhead caused by device failures.

Table 7: Recovery overhead of the passive FT and the proactive FT

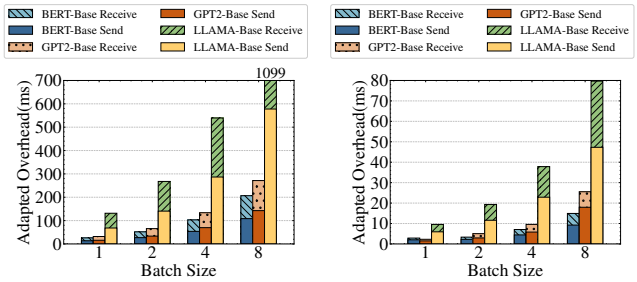
Passive FT	Time (ms)	Proactive FT	Time (ms)
Detection	4631	**Search**	6325
WR	5764	**Substitution**	23168
Retrain	4439	**Retrain**	4245
Recovery	16834	**Recovery**	4487

WR: Weight Redistribution.

5.5 Cross-framework Adapter Overhead

We then assess the time overhead introduced by the cross-framework adapter, measuring the time required to adapt the intermediate results of three models to different frameworks with four different batch sizes. We conduct experiments on both the sender and receiver sides, with one device running PyTorch and the other running MNN, respectively. The average of 10 measurements is calculated and presented in Figure 7.

Experiments demonstrate that the time overhead of the cross-framework adapter is negligible when compared to the training time shown in Figure 4. It is worth noting that the adaptation to PyTorch takes longer than the adaptation to MNN, mainly due to the higher execution efficiency of C++ compared to Python.



(a) Nvidia Jetson Nano (b) Redmi K50

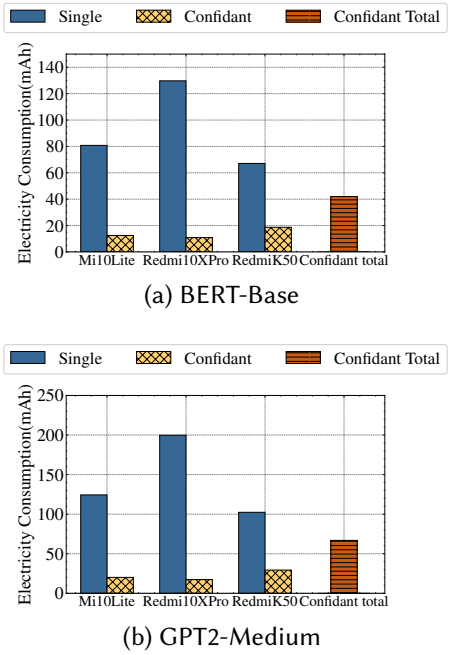
Figure 7: Conversion Overhead cross Different Devices

5.6 Energy Consumption

Finally, we measure the energy consumption of Confidant by fine-tuning LLMs utilizing Mi 10 Lite, Redmi 10X Pro and Redmi K50. We record the battery usage using BatteryStats tool [1]. Specifically, we record the electricity consumption (mAh) of fine-tuning 10 batches with BERT-Base and GPT2-Medium utilizing Confidant and one single device, respectively, as illustrated in Figure 8. We also provide the total electricity consumption of three devices utilizing Confidant. Figure 8 illustrates a significant reduction in power consumption when employing Confidant compared to fine-tuning the entire model on a single device. This reduction remains substantial even when considering the total energy consumption of all three devices, showcasing a remarkable energy saving of Confidant compared to the single-device training.

5.7 Attempts on Billion-sized LLMs

We also attempt to fine-tune LLMs with billions of parameters, i.e., Phi2-2.7B and LLaMA-7B, on edge devices using Confidant. Due to the substantial memory requirements for



(a) BERT-Base

(b) GPT2-Medium

Figure 8: Comparisons of the electronic consumption between single-device fine-tuning and Confidant-based fine-tuning.

fine-tuning these two models, we conduct LoRA-based fine-tuning using one mobile device (Redmi K50) and two laptops (Macbook Pro M1 and Colorful X15-AT), supported by the proposed Cross-framework Adapter. We measure the average training time per batch and the average memory usage of fine-tuning these two models, as listed in Table 8.

Table 8: Evaluations on Billion-sized LLMs

Model	Batchsize	Time per Batch	Average Memory Usage
Phi2-2.7B	1	19.29 s	8.27 GB
	2	19.46 s	10.84 GB
LLaMA-7B	1	29.92 s	12.51 GB
	2	88.06 s	14.10 GB

It can be observed from Table 8 that the billion-sized LLMs are successfully fine-tuned through three mobile devices. However, training a batch still requires a significant amount of time, especially for the LLaMA-7B. Accelerating the training of billion-scale LLMs on mobile devices is among our future endeavors.

6 RELATED WORKS

On-device LLM Inference. Research on on-device LLM inference mainly focuses on accelerating the inference and

reducing memory usage. Techniques employed for speeding up LLM inference are those commonly used in edge deep learning acceleration, including efficient model structure [10, 32], quantization [43], feature pruning [34], and sparsification [23]. Building upon accelerated inference, some studies have also reduced memory usage [35, 44]. Kim *et al.* [18] proposed BiLD that generates text at a low computation and memory cost through a relatively small LLM, corrected by a large LLM when the output of the small LLM is inaccurate. In EdgeMoE, it only loads the LLM weights that are frequently used in memory, with the other weights only loaded from disk when they are used. While there have been many studies dedicated to on-device LLM inference, few studies focus on fine-tuning LLMs on edge devices. Our work addresses this gap in the existing literature.

On-device DNN Training. Since training requires much more memory than inference [3], current research on edge training primarily focuses on reducing the memory footprint introduced during the training process. Gim *et al.* introduced Sage [12], a framework that reduces memory usage through graph-level and operator-level optimizations, incorporating both gradient checkpointing and gradient accumulation techniques. Melon [39] is a framework that intelligently allocates tensors to the appropriate memory address by considering the tensors' lifetime, which enables training a model beyond physical memory capacity. Both frameworks only consider models as large as BERT. In FwdLLM [45], it achieves fine-tuning billion-sized LLMs across mobile devices by utilizing a backpropagation-free training method called perturbed inferences. However, the generality of BP-free methods is lower than that of traditional backpropagation methods. Additionally, FwdLLM is built upon federated learning across a large number of mobile devices to ensure the model convergence speed and training time. Confidant instead employs the traditional backpropagation to fine-tune LLMs on mobile devices, providing more generality and requiring fewer mobile devices.

Co-execution of Multiple Processors. Existing works of multi-processor co-execution mainly concentrate on leveraging the on-device CPU and GPU to compute DNN parallelly. Wang *et al.* [40] proposed OPTiC that achieved optimal co-execution by automatically partitioning the DNN workload and selecting the operating frequency of processors while satisfying hardware thermal constraints. In contrast, μlayer [19] and CoDL[14] utilized the latency estimation of DNN operators to reduce the idle time of processors, thereby maximizing both CPU and GPU computation time. To estimate the latency, μlayer employed a FLOPs-based linear-regression model, while CoDL adopted a non-linear and concurrency-aware mathematical formula. NN-Stretch [41] transformed a long and narrow model into a model with several independent branches to achieve parallel computation of branches on multiple processors. We note that current studies are designed for on-device DNN inference. Our proposed multi-processor scheduler is applied to on-device fine-tuning, which accelerates both forwarding and backwarding.

7 CONCLUSION

In this paper, we have proposed and implemented Confidant, a collaborative training framework for fine-tuning LLMs on various edge devices across different DNN frameworks. Leveraging pipeline parallel training, Confidant reduces the average memory usage on each device, achieving efficient fine-tuning through memory-aware dynamic model partitioning and an intra-device multi-processor scheduler. Additionally, a cross-framework adapter enables fine-tuning across different DNN frameworks. Confidant also introduces a hybrid fault tolerance mechanism to offer strong robustness against both predictable and unpredictable faults during collaborative training. Experiments on practical mobile devices have demonstrated that Confidant achieved up to 5.56 times acceleration in the experimental settings, realizing computation- and memory-efficient, and robust collaborative training on mobile devices. Furthermore, we have also successfully fine-tuned billion-sized LLMs using only three mobile devices with Confidant.

REFERENCES

[1] Android Studio. 2023. Profile battery usage with Batterystats and Battery Historian. https://developer.android.google.cn/topic/performance/power/setup-battery-historian.html. (2023). Accessed on November 19, 2023.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NeurIPS* (2020).

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[4] 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).

[5] Sangjin Choi, Inhoe Koo, Jeongseob Ahn, Myeongjae Jeon, and Youngjin Kwon. 2023. {EnvPipe}: Performance-preserving {DNN} Training Framework for Saving Energy. *USENIX ATC* (2023).

[6] Flask developers. 2022. Flask. https://flask.palletsprojects.com/en/3.0.x/. (2022).

[7] Java Spark developers. 2022. Java Spark. https://sparkjava.com/. (2022).

[8] ONNX Runtime developers. 2021. ONNX Runtime. https://onnxruntime.ai/. (2021).

[9] 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).

[10] Angela Fan, Edouard Grave, and Armand Joulin. 2019. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556* (2019).

[11] Forbes. 2023. What Large Models Cost You – There Is No Free AI Lunch. https://www.forbes.com/sites/craigsmith/2023/09/08/what-large-models-cost-you--there-is-no-free-ai-lunch. (2023). Accessed on March 11, 2024.

[12] In Gim and JeongGil Ko. 2022. Memory-efficient DNN training on mobile devices. *MobiSys* (2022).

[13] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[14] 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. *Mobisys* (2022).

[15] Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. 2020. Profiling and optimizing deep learning inference on mobile GPUs. *ACM APSys* (2020).

[16] 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. 2023. MNN: A Universal and Efficient Inference Engine. *MLSys* (2023).

[17] Chanwoo Kim, Dhananjaya Gowda, Dongsoo Lee, Jiyeon Kim, Ankur Kumar, Sungsoo Kim, Abhinav Garg, and Changwoo Han. 2020. A review of on-device fully neural end-to-end automatic speech recognition algorithms. *ACSSC* (2020).

[18] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W Mahoney, Amir Gholami, and Kurt Keutzer. 2023. Speculative Decoding with Big Little Decoder. *NeurIPS* (2023).

[19] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. *EuroSys* (2019).

[20] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing activation recomputation in large transformer models. *MLSys* (2023).

[21] Pengzhen Li, Erdem Koyuncu, and Hulya Seferoglu. 2021. Respipe: Resilient model-distributed dnn training at edge networks. *ICASSP* (2021).

[22] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. 2021. Terapipe: Token-level pipeline parallelism for training large-scale language models. *PMLR*.

[23] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. 2023. Deja vu: Contextual sparsity for efficient llms at inference time. *ICML* (2023).

[24] Meta Platforms, Inc. 2024. Building Meta's GenAI Infrastructure. https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/. (2024).

[25] 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. (2019).

[26] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. (2020). https://developer.nvidia.com/cuda-toolkit

[27] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).

[28] Kazuki Osawa, Shigang Li, and Torsten Hoefler. 2023. PipeFisher: Efficient Training of Large Language Models Using Pipelining and Fisher Information Matrices. *MLSys* (2023).

[29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAIblog* (2019).

[30] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *ACL* (2018).

[31] Erik F Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. *HLT-NAACL* (2003).

[32] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[34] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. Mobilebert: a compact task-agnostic bert for resource-limited devices. *ACL* (2020).

[35] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M Rush, David Brooks, et al. 2021. Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference. *MICRO* (2021).

[36] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca. (2023).

[37] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[38] Kangkang Wang, Rajiv Mathews, Chloé Kiddon, Hubert Eichner, Françoise Beaufays, and Daniel Ramage. 2019. Federated evaluation of on-device personalization. *arXiv preprint arXiv:1910.10252* (2019).

[39] 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. *Mobisys* (2022).

[40] Siqi Wang, Gayathri Ananthanarayanan, and Tulika Mitra. 2018. OPTiC: Optimizing collaborative CPU–GPU computing on mobile devices with thermal constraints. *IEEE transactions on computer-aided design of integrated circuits and systems* 38, 3 (2018), 393–406.

[41] Jianyu Wei, Ting Cao, Shijie Cao, Shiqi Jiang, Shaowei Fu, Mao Yang, Yanyong Zhang, and Yunxin Liu. 2023. NN-Stretch: Automatic Neural Network Branching for Parallel Inference on Heterogeneous Multi-Processors. *MobiSys* (2023).

[42] Wikipedia contributors. 2023. Java Native Interface — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Java_Native_Interface&oldid=1166267665. (2023). [Online; accessed 3-October-2023].

[43] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. *ICML* (2023).

[44] Daliang Xu, Wangsong Yin, Xin Jin, Ying Zhang, Shiyun Wei, Mengwei Xu, and Xuanzhe Liu. 2023. LLMCad: Fast and Scalable On-device Large Language Model Inference. *arXiv preprint arXiv:2309.04255* (2023).

[45] Mengwei Xu, Yaozong Wu, Dongqi Cai, Xiang Li, and Shangguang Wang. 2023. Federated fine-tuning of billion-sized language models across mobile devices. *arXiv preprint arXiv:2308.13894* (2023).

[46] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. *Mobisys* (2021).

[47] Yichen Zhu, Minjie Zhu, Ning Liu, Zhicai Ou, Xiaofeng Mou, and Jian Tang. 2024. LLaVA-ϕ: Efficient Multi-Modal Assistant with Small Language Model. *arXiv preprint arXiv:2401.02330* (2024).