EDGE-LLM: Enabling Efficient Large Language Model Adaptation on Edge Devices via Layerwise Unified Compression and Adaptive Layer Tuning & Voting

Zhongzhi Yu[1], Zheng Wang[1], Yuhan Li[1], Haoran You[1], Ruijie Gao[1],
Xiaoya Zhou[3], Sreenidhi Reedy Bommu[1], Yang (Katie) Zhao[2], Yingyan (Celine) Lin[1]
[1]*Georgia Institute of Technology*, [2]*University of Minnesota, Twin Cities*, [3]*University of California, Santa Barbara*
{zyu401, zwang2478, yli3326, hyou37, eiclab.gatech, sbommu3, celine.lin}@gatech.edu,
yangzhao@umn.edu, xiaoyazhou@umail.ucsb.edu

Abstract

Efficient adaption of large language models (LLMs) on edge devices is essential for applications requiring continuous and privacy-preserving adaptation and inference. However, existing tuning techniques fall short because of the high computation and memory overhead. To this end, we introduce a **computation**- and **memory**-efficient LLM tuning framework, called Edge-LLM, to facilitate affordable and effective LLM adaptation on edge devices. Specifically, Edge-LLM features three core components: **(1)** a layer-wise unified compression (LUC) technique to reduce the **computation** overhead by generating layer-wise pruning sparsity and quantization bit-width policies, **(2)** an adaptive layer tuning and voting scheme to reduce the **memory** overhead by reducing the backpropagation depth, and **(3)** a complementary hardware scheduling strategy to handle the irregular computation patterns introduced by LUC and adaptive layer tuning, thereby achieving improved real hardware efficiency. Extensive experiments demonstrate that Edge-LLM achieves on-device adaptation with comparable task accuracy as vanilla tuning methods with a 2.92× speed up and a 4× reduction in memory overhead. Our code is available at https://github.com/GATECH-EIC/Edge-LLM

ACM Reference Format:
Zhongzhi Yu[1], Zheng Wang[1], Yuhan Li[1], Haoran You[1], Ruijie Gao[1],, Xiaoya Zhou[3], Sreenidhi Reedy Bommu[1], Yang (Katie) Zhao[2], Yingyan (Celine) Lin[1]. 2024. EDGE-LLM: Enabling Efficient Large Language Model Adaptation on Edge Devices via Layerwise Unified Compression and Adaptive Layer Tuning & Voting. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658473

DAC'24, June 23–27, 2024, San Francisco, CA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3658473

1 Introduction

In recent days, large language models (LLMs), such as GPT-4 [1], have shown dominating performance across various applications that revolutionize human life. Following this trend, there is an increasing demand to develop efficient on-device tuning techniques for LLMs to enable them on applications that require continuous and privacy-preserving adaptation. However, the massive model size of LLMs hinders directly achieving on-device adaptation of LLMs (e.g., on edge GPUs and smartphones). The challenges are twofold: (1) *the excessive **computation** overhead* encountered when calculating the forward and backward passes of LLMs [2], and (2) *the cumbersome **memory** overhead* introduced by storing massive model weights and activations through the tuning process. As shown in recent works [2, 11], LLMs are typically tuned on cutting-edge GPUs (e.g., with 40GB or 80GB GPU memory), taking more than a GPU day to complete. Even for the state-of-the-art (SOTA) efficient tuning method, effectively tuning relatively small-scale LLMs (e.g., LLaMA-7B) on edge devices remains impractical [2].

Although several existing efforts aim to address the aforementioned challenges, each has its own drawbacks. (1) To reduce computation overhead, compressing target LLMs first to reduce the model size is a common approach [2, 3]. However, **how to effectively reduce the redundancy of LLMs while maintaining their adaptability is still largely unexplored** [2]. (2) To mitigate memory overhead, existing methods primarily focus on shortening the backpropagation depth [19, 23]. Unfortunately, the reduced backpropagation depth results in only **a fraction of blocks in LLMs being updated, limiting the achievable performance**.

In this paper, we develop a comprehensive solution to tackle the two aforementioned memory and computation challenges, achieving effective on-device LLM adaptation. Specifically, we make the following contributions.

- We propose a comprehensive framework, dubbed Edge-LLM, that tackles the memory and computation challenges of on-device LLM adaptation from both algorithm and hardware perspective, enabling effective LLM adaptation on edge devices with limited memory and computation resources.

- **On the algorithm side**, we accomplish this goal from two directions, each primarily focusing on one of the aforementioned challenges: (1) To reduce the **computation** overhead, we propose a low-cost layer-wise unified compression (LUC) method based on our empirical observation on LLMs' layer-wise sensitivities to quantization and pruning. (2) To reduce the **memory** overhead, we introduce an adaptive layer tuning and voting scheme. In adaptive layer tuning, we propose to selectively update distinct segments of the target LLM and reduce the memory footprint by directly connecting the output of the current updating segment to the final layer. Further, in adaptive layer voting, we harness the outputs of different segments of the target LLM by voting for an optimized output.
- **On the hardware side**, to better handle the irregular computation patterns (i.e., diverse layer-wise quantization bit-width, layer-wise pruning sparsity, and LLM segments to update) introduced by the proposed algorithms, we further integrate a complementary hardware scheduling module into Edge-LLM. The hardware scheduling module includes a search space and a search strategy considering potential offloading strategies, computation schedules, and tensor placements, aiming to better convert the theoretical reduction in computation overhead to real hardware efficiency improvement.
- Experiment results and ablation studies show the effectiveness of our proposed Edge-LLM framework. Specifically, Edge-LLM achieves a 0.70%~1.29% higher MMLU score compared with the baseline methods tuned under the same resource constraints and a comparable perplexity on WikiText-2 as LoRA tuning with a 2.92× lower latency and a 4× reduction in memory overhead in each iteration.

2 Background and Motivation

2.1 Efficient Tuning Techniques

Parameter-efficient tuning (PET) comprises techniques for tuning LLMs to new tasks using a limited number of trainable parameters, typically less than 10% of the total parameters in the target LLMs [5, 6, 10, 19]. It offers two major advantages: (1) reduced storage overhead, facilitating scalable multitask deployment, and (2) a marginal reduction in computation and memory overhead, thanks to the reduced number of trainable parameters [10]. Despite PET's widespread use, directly applying it for on-device LLM adaptation remains impractical due to the remaining memory overhead is still significant. This is because PET typically inserts a learnable adapter to most, if not all, layers of the target LLM, leading to significant memory overhead to store intermediate activations during tuning.

Memory-efficient tuning (MET) aims to minimize the memory footprint during the tuning process by reducing backpropagation depth, thereby decreasing the number of activations required to be stored in memory [19, 23]. Existing MET techniques achieve this goal either using partial tuning to only tune the final few layers [23] or leveraging side tuning to add a bypass connection between each adapter module with the final output [19]. While the reduction of memory footprint during tuning is highly desirable, existing MET techniques still face an unsatisfactory trade-off between accuracy and memory footprint in LLM tuning. Specifically, for partial tuning, existing attempts on LLMs need to tune more than 80% of layers of the target LLM to achieve a satisfactory task accuracy [23], while side tuning suffers from biased optimization and struggles to achieve task accuracy comparable to SOTA PET techniques [19].

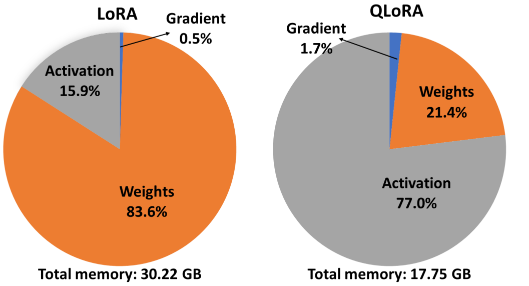


Figure 1. Profiling results on the memory footprint when tuning LLaMA-7B with LoRA [10] and QLoRA [2] on the Alpaca [20] dataset.

Compressing-then-tuning is a series of emerging efficient tuning techniques motivated by the observation that the computation overhead in LLM tuning is dominated by the forward and backward passes of the LLM's backbone, due to the excessive size of the LLM's backbone [2]. Thus, some pioneering works propose to compress the LLM backbone before tuning to alleviate the computation overhead [2]. However, existing SOTA compressing-then-tuning techniques primarily aim to improve tuning speed, neglecting the memory overhead (e.g., the SOTA compressing-then-tuning method still needs an A100 GPU with 40GB memory to achieve effective tuning on Llama-70B [2]). This oversight limits the effectiveness of compressing-then-tuning techniques in tuning LLMs on resource-constraint edge devices.

2.2 Memory Overhead During Tuning

To better understand the gap between the memory needed in existing tuning techniques and the memory available on edge devices, we profile the **memory** requirements to tune a Llama-7B [23] with LoRA [10], one of the SOTA PET techniques, and QLoRA [2], one of the SOTA compressing-then-tuning techniques, respectively. As shown in Fig. 1, the memory overhead of LoRA is dominated by storing the LLM's backbone weights and the activations for backpropagation. Even after QLoRA compressed the LLM backbone to 4-bit and reduced the overall memory footprint by 41.2% over LoRA, there remains a 1.48~2.22× gap between the memory required for tuning and the memory available on commonly used edge devices (e.g., 8 GB for TX2 [14] and 12 GB for Quest Pro [13]).

2.3 Opportunities for On-device LLM Tuning

Despite the limitations of existing tuning techniques, we also identify opportunities to improve upon these methods to develop effective on-device LLM tuning frameworks.

On the one hand, to further reduce the **computation** overhead, we identify a mismatch between the previously successful practice aimed at reducing the model redundancy and the vanilla compression technique used in existing compressing-then-tuning techniques. Specifically, previous efforts (e.g., [4] observe that deep learning models exhibit redundancy across different dimensions (e.g., bit-width and sparsity) and at different layers. In contrast, existing compressing-then-tuning techniques often adopt a uniform compression approach, reducing redundancy from only one dimension [2].

On the other hand, to further reduce the **memory** overhead, based on our analysis in Sec. 2.1, we summarize that the key to improving the achievable accuracy-memory trade-off lies in the ability to update all layers in the LLM with a limited backpropagation depth. Inspired by the early exit mechanism developed for efficient model inference [21], we hypothesize that the outputs from early layers in the LLM can provide meaningful information for prediction. Thus, it is possible to start backpropagation from an early exit layer and still effectively update the model. In this scenario, since backpropagation can be initiated from various early exit layers, the backpropagation depth required for updating all layers in the LLM can be minimized.

3 Edge-LLM Algorithm

3.1 Overview

Motivated by the opportunities identified in Sec. 2.3, we then introduce the algorithm design of our proposed Edge-LLM framework to facilitate effective on-device LLM adaptation with limited computation and memory overhead. As shown in Fig. 2, our proposed Edge-LLM tuning algorithm integrates two key enablers each leveraging one of the aforementioned opportunities in reducing the computation and memory overhead. Specifically: (1) To reduce the **computation** overhead, we propose a layer-wise unified compression (LUC) technique to diminish the redundancy of the target LLM. This technique is motivated by our empirical observation of LLMs' diverse layer-wise sensitivities to quantization and pruning. Based on the observation above, we develop a low-cost, mean-square-error-based (MSE-based) identifier in LUC to generate a layer-wise compression policy (e.g., layer-wise bit-width and pruning sparsity allocation), aiming to improve the accuracy-efficiency trade-off of LUC over existing compression techniques in compressing-then-tuning frameworks (Sec. 3.2). (2) To reduce the **memory** overhead, we propose an adaptive layer tuning scheme that dynamically connects the output of a selected layer (potentially different in each iteration) to the final classification layer with a skip connection during the forward pass. During backpropagation, this scheme only updates a few preceding layers of the selected layer. The proposed scheme effectively updates all layers with minimal memory overhead, thanks to the reduced backpropagation depth introduced by the skip connection. Furthermore, we introduce a voting mechanism to leverage the characteristic of LLMs tuned with adaptive layer tuning can generate reasonable outputs from multiple layers, aiming to further improve the performance of LLMs tuned with adaptive layer tuning (see Sec. 3.3).

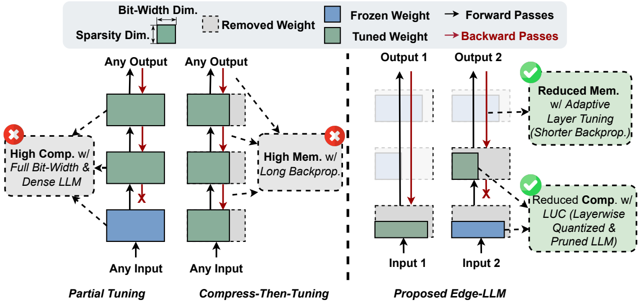


Figure 2. Comparison between different tuning techniques and our proposed Edge-LLM framework.

3.2 Layer-wise Unified Compression (LUC)

Motivating observation on LLM's layer-wise sensitivity. In prior studies on model compression, a common understanding is that different layers in the model exhibit different sensitivities to different compression techniques [4]. However, the sensitivities of different layers in LLMs to different compression techniques remain an open question. To address this question, we first explore the layer-wise sensitivities of the target LLM to pruning and quantization. Specifically, we apply different quantization bit-widths and pruning sparsities to each layer of a pretrained LLaMA-7B [22]. By comparing the averaged MSE of the compressed and original layer outputs in the target LLM fed with the same input from the WikiText dataset [12], we observe that, as shown in Fig. 3, only a small fraction of layers in the LLM have high sensitivities to compression.

Our hypothesis and the proposed LUC. Based on the observation above, we hypothesize that the high sensitivity (i.e., high MSE) is due to limited redundancy in the corresponding layer, thereby necessitating a lower compression ratio. To this end, we propose the following mapping functions to map the layer-wise MSE to the layer-wise quantization bit-width and pruning sparsity, respectively. For **quantization**, given an LLM M with L layers, formulating $\mathcal{L} = \{l_0, l_1, \cdots, l_{L-1}\}$, a base quantization bit-width B, and the quantization sensitivity (i.e., the MSE between the output of the original layer and the output of the B-bit quantized layer) for layer l_i as s^i_{quant}, we define the optimized quantization bit-width b_j at layer l_j as

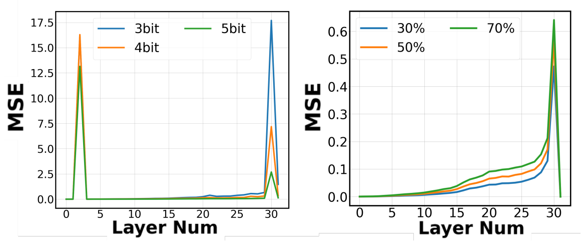


Figure 3. Visualization of LLaMA-7B's layer-wise sensitivity to (a) quantization and (b) pruning.

$$b_j = B + \mathbb{1}(s^j_{quant} \geq \frac{\sum_{i=0}^{L-1} s^i_{quant}}{L}), \tag{1}$$

where $\mathbb{1}(.)$ is the indicator function. For **pruning**, given a target overall pruning sparsity P, we define the pruning sparsity p_j at layer l_j as

$$p_j = P \times L \times \frac{s^j_{prune}}{\sum_{i=1}^{L-1} s^i_{prune}}, \tag{2}$$

where s^j_{prune} is the pruning sensitivity for layer l_j.

3.3 Adaptive Layer Tuning and Voting

In this enabler, our objective is to facilitate effective tuning with reduced memory overhead, thereby fitting the tuning process into edge devices with limited memory capacity. To accomplish this, the primary challenge we've identified is enabling efficient updates across all layers of the target LLM with restricted backpropagation depth, as analyzed in Section 2.3.

In Edge-LLM, we alleviate this challenge by constructing a set of exit layers $\mathcal{T} = \{t_0, t_1, \cdots, t_{T-1}\}$, where each t_i connects to the output of layer $l_{\text{Ceil}((i+1)\times L/T)}$ in the target LLM, note that t_{T-1} is the original exit layer in the target LLM. In each tuning iteration, we randomly select $t_i \in \mathcal{T}$ as the only exit layer to use, and update the following set of layers $\{t_i, l_{\text{Ceil}((i+1)\times L/T)}, l_{\text{Ceil}((i+1)\times L/T)-1}, \cdots, l_{\text{Ceil}((i+1)\times L/T)-m}\}$ with LoRA adapters attached to each layer, where $m = \text{Ceil}(L/T)$.

Furthermore, with the adaptive layer tuning described above, the tuned LLM can generate outputs from all layers $t \in \mathcal{T}$. Although directly using the final output layer t_{T-1} can achieve competitive performance, having multiple available exit layers provides an opportunity to further enhance the performance at inference time by adaptively combining the outputs of different layers. To this end, we propose a voting mechanism to enhance the performance by making predictions based on the outputs from all exit layers. Specifically, inspired by existing findings about the relationship between post-softmax probability and prediction confidence [15], we determine the final output index by choosing the one with the highest post-softmax probability across all exit layers. Specifically, given an output probability matrix $\mathbf{M}$, with each element $\mathbf{m}_{(i,j)}$ representing the output probability for index j from layer $t_i \in \mathcal{T}$. We first find the location of the maximum value in $\mathbf{M}$ with $(i_{max}, j_{max}) = \arg\max_{i,j}(\mathbf{m}_{(i,j)})$, then we generate the final output as $o = j_{max}$.

4 Edge-LLM Hardware Scheduling

Motivation. The aforementioned algorithm designs introduce an irregular computation pattern (i.e., diverse layer-wise quantization bit-width, layer-wise pruning sparsity, and layers to update). This complexity makes it challenging for real devices to fully benefit from the algorithm's theoretical reduction in computation overhead. To address this challenge, we propose a complementary hardware scheduling module, focusing on efficient scheduling and offloading strategies tailored for optimizing LLM inference throughput. The on-chip accelerator SRAM size limitation (512KB~1MB) highlights the inability to load all model weights and activations, necessitating offloading to secondary storage mediums like DRAM (8GB~16GB) and SSD (128GB~256GB). Our hardware acceleration initiative is driven by the need to establish a comprehensive cost model, serving as the basis for efficient memory scheduling or offloading strategies for each early exit block in the system.

4.1 Overview

In the pursuit of optimizing the scheduling and offloading strategies for LLM hardware accelerators, our methodology allocates bit-widths and pruning sparsities to each layer based on sensitivity (see Sec. 3.2). Subsequently, we conduct a nuanced exploration to identify the optimal offloading strategy for each early exit block. As depicted in Fig. 4 (a) and (b), these two steps take algorithm hyperparameters as inputs and yield the final allocation strategy and hardware schedulings as outputs.

4.2 Searching Objective

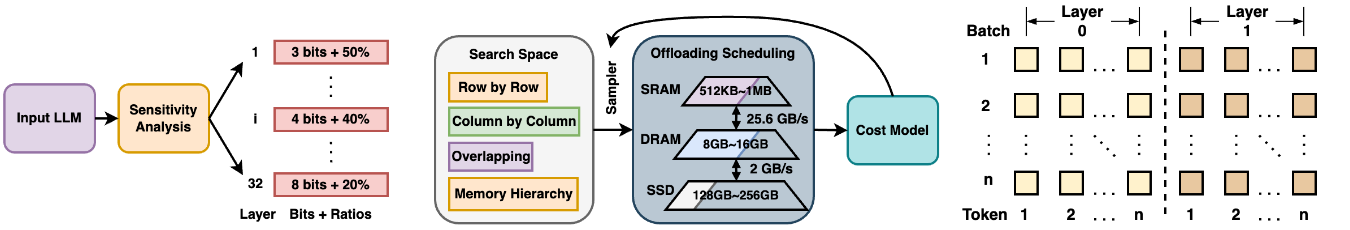
We conceptualize the LLM tuning with offloading as a graph traversal problem following [18]. In Fig. 4 (c), we present an illustrative computational graph consisting of three dimensions of batches, layers, and tokens. In the depicted graph, each square denotes the computation of a specific layer. Squares sharing the same color indicate the utilization of identical layer weights. A valid path is defined as a trajectory that traverses (i.e., computes) all squares, adhering to the following constraint:

- During LLM forwarding or backpropagation, a square's computation depends on the left or right layers in its row being completed, respectively.
- To compute a square, all its inputs (weights, activations, cache) must be loaded onto the on-chip SRAM.
- At any given time, the cumulative size of tensors stored on an accelerator must not exceed its memory capacity.

The objective is to identify a valid path that minimizes the overall execution time, encompassing both compute costs and I/O costs incurred during the movement of tensors between devices.

4.3 Block Search Space.

Building upon the aforementioned search objective, we establish a search space encompassing potential valid strategies.



(a) Bit and Prunning Ratio Allocation (b) Hardware Scheduling Search (c) Scheduling Search Problem Definition

Figure 4. The overview of our hardware scheduling.

- **Row-by-row.** Existing systems often use solely row-by-row traversal for the activation footprint savings. However, this strategy does not consider the weight sharing between adjacent squares among different bathes, leading to repetitive weight loading and I/O costs.
- **Mixed column-by-column and row-by-row.** Alternatively, to reduce I/O costs related to weights, an approach involves traversing the graph column-by-column. This leverages weight sharing among all squares in a column, allowing DRAM preservation for reuse, with activations being loaded and unloaded. As our proposed algorithm techniques can greatly reduce the activation memory footprint requirement, we include mixed column-by-column and row-by-row in search space.

Considerations. *Overlapping.* Another optimization is overlapping. This entails concurrently handling a load of weights for the next layer, the load of activations for the subsequent batch, the storage of activations from the preceding batch, and the computation of the current batch. The integration of overlapping into the block schedule is necessary for delivering the final scheduling.

Tensor Placement. In addition to the computation schedule, an effective strategy must delineate the placement of tensors within the memory hierarchy. Three variables, namely w_{sram}, w_{dram}, and w_{ssd}, define the percentages of weights stored on the SRAM, DRAM, and SSD, respectively. Similarly, three variables, a_{sram}, a_{dram}, and a_{ssd} articulate the percentages of activations; and three variables, g_{sram}, g_{dram}, and g_{ssd} articulate the percentages of gradients.

4.4 Cost Models

Having established the search objective and the search space, the next step is the development of an analytical cost model. This model serves the purpose of estimating the execution time based on the specified algorithm parameters and hardware specifications. The total latency for computing a block can be estimated as T_{dec}. Assuming perfect overlapping, T_{dec} can be estimated as

$$T_{\text{dec}} = \max(r_{\text{to\_sram}}, w_{\text{to\_dram}}, r_{\text{to\_dram}}, w_{\text{to\_ssd}}, T_{\text{comp}}) \quad (3)$$

where $r_{\text{to\_sram}}$, $w_{\text{to\_dram}}$, $r_{\text{to\_dram}}$, $w_{\text{to\_ssd}}$, and T_{comp} denote the latency of read from DRAM to SRAM, write from SRAM to DRAM, read from SSD to DRAM, write from DRAM to SSD, and computation, respectively, during LLM tuning.

To leverage the cost model, we initiate profiling on the hardware to collect sample data points and subsequently calibrate the hardware parameters. Following this, we invoke the optimizer to obtain an offloading policy. There are occasional instances where a strategy from the policy search exceeds the available memory. In such cases, we make slight manual adjustments to the policy. While the cost model typically yields a sound policy, it is not uncommon to achieve further optimization through manual tuning.

5 Evaluation

5.1 Evaluation Setup

Datasets: Two commonly used benchmarking dataset including MMLU [9] and WikiText [12]. **Model:** LLaMA-7B [22]. **Algorithm baselines:** The SOTA PET technique LoRA [10], the SOTA MET technique LST [19], the SOTA compression techniques Sparse-GPT [7] and LLM-QAT [11], and seven variants of our proposed methods. **Hardware baselines:** The SOTA systolic accelerator [17] dedicated for transformer training. **Algorithm implementation:** We use LLM-QAT and Sparse-GPT as the quantization and pruning techniques, respectively, and tune the model following the settings in [2]. **Hardware configuration:** The accelerator's DRAM is set to 8GB LPDDR4 and on-chip SRAM to be 1MB, in line with SOTA edge devices [14], with other hardware configurations following the baseline training accelerator design. **Evaluation methodology:** We use the SOTA Scale-Sim [16] simulator to simulate both the baseline accelerator and those after applying our proposed techniques on the baseline accelerator.

5.2 Algorithm Evaluation

To evaluate the performance of our proposed method, we first benchmark our proposed method with existing baseline methods including partial tuning, LST and LoRA tuning on the commonly used MMLU dataset. As shown in Table 1, our method consistently achieves a 0.70%~1.29% higher accuracy with the same computation efficiency and a 4× reduction in memory over the baseline methods. To further validate the key enablers in Edge-LLM, we first evaluate the LUC's perplexity separately on the WikiText-2 dataset over two SOTA compression techniques including SparseGPT and LLM-QAT, and two variants: (1) Uniform: using the same quantization bit-width and pruning sparsity across all layers

Table 1. Benchmarking Edge-LLM on MMLU dataset.

Method	Avg. Bit	Sparsity	Norm. Mem.	MMLU
LoRA	8.0	0%	1.00×	33.60
Partial Tuning	5.0	50%	0.25×	30.94
Ours	5.1	50%	0.25×	**31.64**
LST	4.0	0%	0.29×	29.04
Partial Tuning	4.0	50%	0.25×	28.70
Ours	4.1	50%	0.25×	**29.89**
Partial Tuning	3.0	50%	0.25×	26.61
Ours	3.1	50%	0.25×	**27.68**

Table 2. Ablation on LUC's performance with its variants

Method	Avg. Bit	Sparsity	Perplexity
SparseGPT	8.0	50%	15.88
LLM-QAT	8.0	0%	13.34
Uniform	5.0	50%	17.61
Random	5.1	50%	16.21
Ours	5.1	50%	**15.71**
Uniform	4.0	50%	19.86
Random	4.1	50%	19.81
Ours	4.1	50%	**18.58**
Uniform	3.0	50%	32.52
Random	3.1	50%	31.71
Ours	3.1	50%	**30.03**

and (2) Random: Randomly assign our generated layer-wise pruning sparsities and quantization bits across all layers. As shown in Table 2, our proposed method achieves a 1.28∼2.49 lower perplexity compared to the Uniform baseline under similar resource constraints and a 0.50∼1.68 lower perplexity compared to the Random baseline under the same efficiency, showing the effectiveness of our proposed LUC.

5.3 Hardware Evaluation

To evaluate the proposed techniques, we use the baseline systolic accelerator designed for transformer training and make proper modifications for supporting the proposed techniques [17]: (1) Since the proposed adaptive layer tuning can be naturally run on the baseline accelerator, there is no need to modify the baseline accelerator; and (2) For the LUC, we make these modifications: First, we update the baseline to store the compressed weights on DRAM and SSD. To make the design easy, we do not modify the compute core for sparsity and use a simple spatial-temporal flexible-precision MAC unit [8]. We apply our proposed hardware scheduling searching method to find the optimal algorithm-to-hardware mappings. Scale-Sim simulation results show that the adaptive layer tuning can achieve 2.24× speedup; the pruning and adaptive layer tuning can introduce 2.37× speedup; and combing LUC (4-bit/5-bit) and the adaptive layer tuning can give 3.38×/2.92× overall speedup, respectively.

6 Conclusion

In this paper, we introduce an LLM tuning framework, Edge-LLM, achieving efficient LLM adaptation on edge devices. Experiments demonstrate that Edge-LLM achieves efficient adaptation with comparable performance as vanilla tuning with a 2.92× speed up and a 4× memory reduction.

Acknowledgement

This work was supported in part by CoCoSys, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and the National Science Foundation (NSF) through the NSF CAREER funding (Award number: 2048183).

References

[1] Bubeck et al. 2023. Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv preprint arXiv:2303.12712 (2023).

[2] Dettmers et al. 2023. Qlora: Efficient finetuning of quantized llms. arXiv (2023).

[3] Kim et al. 2024. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization. NeurIPS 36 (2024).

[4] Yu et al. 2022. Unified visual transformer compression. arXiv preprint arXiv:2203.08243 (2022).

[5] Yu et al. 2023. Hint-aug: Drawing hints from foundation vision transformers towards boosted few-shot parameter-efficient tuning. In CVPR. 11102–11112.

[6] Yu et al. 2023. Master-ASR: achieving multilingual scalability and low-resource adaptation in ASR with modular learning. In ICML. PMLR, 40475–40487.

[7] Frantar et al. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. (2023).

[8] Fu et al. 2021. Enabling random precision switch for winning both adversarial robustness and efficiency. In MICRO. 225–237.

[9] Hendrycks et al. 2020. Measuring massive multitask language understanding. arXiv preprint arXiv:2009.03300 (2020).

[10] Hu et al. 2021. Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021).

[11] Liu et al. 2023. LLM-QAT: Data-Free Quantization Aware Training for Large Language Models. arXiv (2023).

[12] Merity et al. 2016. Pointer sentinel mixture models. arXiv preprint arXiv:1609.07843 (2016).

[13] Meta. 2022. Quest Pro. https://www.meta.com/quest/quest-pro/.

[14] NVIDIA. 2020. NVIDIA Jetson TX2. www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.

[15] Pearce et al. 2021. Understanding softmax confidence and uncertainty. arXiv preprint arXiv:2106.04972 (2021).

[16] Samajdar et al. 2023. Systolic CNN AcceLErator Simulator (SCALE Sim). https://github.com/ARM-software/SCALE-Sim.

[17] Shao et al. 2023. An Efficient Training Accelerator for Transformers With Hardware-Algorithm Co-Optimization. VLSI (2023).

[18] Sheng et al. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. (2023).

[19] Sung et al. 2022. Lst: Ladder side-tuning for parameter and memory efficient transfer learning. NeurIPS 35 (2022), 12991–13005.

[20] Taori et al. 2023. Stanford alpaca: An instruction-following llama model.

[21] Teerapittayanon et al. 2016. Branchynet: Fast inference via early exiting from deep neural networks. In ICPR.

[22] Touvron et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023).

[23] Zhang et al. 2023. Llama-adapter: Efficient fine-tuning of language models with zero-init attention. arXiv (2023).