EE-LLM: Large-Scale Training and Inference of Early-Exit Large Language Models with 3D Parallelism

Yanxi Chen [*1] **Xuchen Pan** [*1] **Yaliang Li** [1] **Bolin Ding** [1] **Jingren Zhou** [1]

Abstract

We present EE-LLM, a framework for large-scale training and inference of early-exit large language models (LLMs). While recent works have shown preliminary evidence for the efficacy of early exiting in accelerating LLM inference, EE-LLM makes a foundational step towards scaling up early-exit LLMs by supporting their training and inference with massive 3D parallelism. Built upon Megatron-LM, EE-LLM implements a variety of algorithmic innovations and performance optimizations tailored to early exiting, including a lightweight method that facilitates backpropagation for the early-exit training objective with pipeline parallelism, techniques of leveraging idle resources in the original pipeline schedule for computation related to early-exit layers, and two approaches of early-exit inference that are compatible with KV caching for autoregressive generation. Our analytical and empirical study shows that EE-LLM achieves great training efficiency with negligible computational overhead compared to standard LLM training, as well as outstanding inference speedup without compromising output quality. To facilitate further research and adoption, we release EE-LLM at https://github.com/pan-x-c/EE-LLM.

1. Introduction

Large language models (LLMs) have amazed the world with their astonishing abilities and performance in solving a wide range of problems (Brown et al., 2020; OpenAI, 2023; Chowdhery et al., 2023; Zhang et al., 2022; Touvron et al., 2023a;b). This is accompanied by excessive costs and carbon emissions for training and deploying these models, as their sizes have grown rapidly in recent years. In general, costs for inference are dominant in the long run, as each model will be deployed to solve many problems for a long period of time. This has inspired researchers and engineers to develop various approaches for accelerating LLM inference.

The focus of this work is *early exiting*, which accelerates inference by allowing a deep neural network to make predictions and exit early in the network for certain inputs, without running the forward pass through the full network. This is achieved by augmenting a standard neural network architecture (with a single exit at the end) with additional early-exit layers that transform intermediate hidden states into early outputs. The early-exit model architecture, as visualized in Figure 1, not only retains the full capacity of a large model, but is also capable of adaptively using a smaller amount of computation for solving simpler problems. The idea of early exiting is a natural analogue of how humans speak, think, and make decisions: not every problem requires or deserves the same amount of consideration, and one shall opt for fast reaction to simple problems without overthinking (Kaya et al., 2019). Early exiting has been an active research area and widely applied in natural language processing, computer vision, and other areas (Liu et al., 2020; Elbayad et al., 2020; Schwartz et al., 2020; Xin et al., 2020; Schuster et al., 2021; Teerapittayanon et al., 2016; Huang et al., 2018; Scardapane et al., 2020; Han et al., 2022; Xu & McAuley, 2023). More recently, it has started to gain attention in the generative LLM domain (Schuster et al., 2021; Corro et al., 2023; Bae et al., 2023; Varshney et al., 2023), and is recognized as a promising direction for further reducing the latency and costs of LLM inference (Pope et al., 2022).

Goal and motivations. Our primary goal is to build the infrastructure for *scaling up* training and inference of early-exit LLMs. This is motivated by the observation that sizes of early-exit models in prior works are still relatively small. While the largest early-exit LLM that we are aware of has 13 billion parameters (Varshney et al., 2023), standard LLMs at much larger scales, e.g. the 175B GPT-3 (Brown et al., 2020), 530B Megatron-Turing NLG (Smith et al., 2022), 540B PaLM (Chowdhery et al., 2023), or even larger

[*]Equal contribution [1]Alibaba Group. {chenyanxi.cyx, panxuchen.pxc, yaliang.li, bolin.ding, jingren.zhou}@alibaba-inc.com. Correspondence to: Yaliang Li <yaliang.li@alibaba-inc.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

sparsely activated models, have been well trained and deployed in many applications. It is an urgent need for the community to truly understand the efficacy of early exiting for LLMs at larger scales, which is indispensable for making early exiting a useful and practical option in complex scenarios that only sufficiently large LLMs can handle.

Challenges. The first and foremost question is how to train an early-exit LLM that is too large to fit into the memory of one single device (e.g. GPU). While state-of-the-art frameworks like Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021b), DeepSpeed (Rasley et al., 2020; Smith et al., 2022), Alpa (Zheng et al., 2022), and many more, support training standard LLMs at large scales with data parallelism and model parallelism (including tensor, sequence, and pipeline parallelism), they do not provide native support for early-exit LLMs. One particular challenge lies in pipeline parallelism (Narayanan et al., 2019; 2021a; Fan et al., 2021), which partitions the model along the depth dimension into multiple pipeline stages, connected by limited point-to-point communication between devices; this seems to contradict training early-exit models, as the training objective is typically an aggregation of losses for multiple exits that are now located separately on different pipeline stages. Despite the necessity of pipeline parallelism in many scenarios, we are not aware of any implementation that supports training early-exit LLMs with it.

Moreover, training efficiency for early-exit generative LLMs requires special design, since each early exit contains (at least) a large output embedding matrix that transforms hidden states into logits on the vocabulary, which can constitute a non-trivial proportion of the whole model. Naive implementation of early-exit LLM training can cause large computational overhead compared to standard LLM training.

Finally, with regard to autoregressive generation (where tokens are generated one by one, depending on previously generated tokens via the attention mechanism), naive implementation of early-exit inference is not compatible with KV caching, a standard technique of storing the keys and values of previously generated tokens at each layer. Indeed, if the current token is generated via early exiting at some layer, then its KV caches in later layers are missing, which hinders the generation of future tokens. Given that KV caching is enabled by default in most cases, the efficacy of early exiting for autoregressive generation might be questionable if its conflict with KV caching is not well resolved.

Main contributions. We propose EE-LLM, a system for large-scale training and inference of early-exit (EE) LLMs with 3D parallelism, which is designed to tackle the aforementioned challenges. EE-LLM is built upon Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021b; Smith et al., 2022), and augments it with various functionalities for early exiting. In addition to compatibility with existing functionalities of 3D parallelism provided by Megatron-LM, EE-LLM also implements a variety of algorithmic innovations, including a lightweight method that facilitates backpropagation for the early-exit training objective through pipeline stages, various techniques of leveraging idle resources in the original pipeline schedule for computation related to early-exit layers, and two approaches of early-exit inference that are compatible with KV caching. Implementation of EE-LLM has been well optimized for maximum training and inference efficiency. Our analytical and empirical study confirms that, with negligible computational overhead caused by early-exit layers during training with 3D parallelism, one obtains an early-exit LLM that generates tokens with adaptive token-wise exit selection, achieving outstanding inference speedup without compromising output quality. With EE-LLM, it is now possible to train and deploy early-exit LLMs that are as large as the maximum sizes of standard LLMs allowed by Megatron-LM, given the same amount of computational resources. The source code for EE-LLM can be found at https://github.com/pan-x-c/EE-LLM.

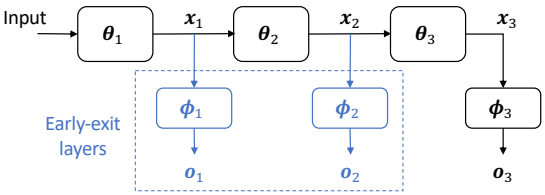


Figure 1: The model architecture of an early-exit LLM. Additional components compared to a standard LLM are highlighted in blue. Each $\boldsymbol{\theta}_i$ represents a sequence of Transformer layers in the backbone of the LLM, with some additional modules in $\boldsymbol{\theta}_1$ for input processing. Each $\boldsymbol{\phi}_i$ represents an early or final-exit layer that converts hidden states $\boldsymbol{x}_i$ into output $\boldsymbol{o}_i$, e.g. logits for next-token prediction.

Organization. Section 2 provides a high-level overview of EE-LLM, while Sections 3 and 4 focus on training and inference, respectively. The efficacy of EE-LLM is validated by numerical experiments in Section 5. The appendix includes extended preliminaries and related works, additional experiments, as well as more details about the training efficiency, advanced features, and implementation of EE-LLM.

2. An overview of EE-LLM

This section provides an overview of our system for scaling up sizes, training and inference of early-exit LLMs, with flexible configurations and a wide range of functionalities.

Model architectures. We implement in EE-LLM an early-exit Transformer architecture, which is built upon the generative pre-training (GPT) Transformer architecture (Radford

et al., 2018; 2019) originally implemented in Megatron-LM. EE-LLM allows users to (1) specify arbitrary layers to add early exits to; (2) add trainable modules to each early-exit layer, e.g. a multi-layer perceptron (MLP) or a complete Transformer layer, on top of the *minimalistic* structure (with an output embedding matrix, plus an optional layer normalization module in front of it); and (3) choose to tie (Press & Wolf, 2017; Schuster et al., 2022; Varshney et al., 2023) or untie the input and output embedding matrices of all early/final-exit layers. Each option has its own pros and cons, as will be discussed in later sections. With this in mind, EE-LLM has been designed to cover a wide range of configurations, so that users can easily try them out and choose the most suitable ones for their own use cases.

Training. EE-LLM contains the essential functionalities for training early-exit LLMs, which tackle the main challenges outlined in Section 1, i.e. how to train with 3D parallelism while minimizing the computational overhead caused by early-exit layers. In addition to substantial engineering efforts for compatibility with existing functionalities in Megatron-LM, we design and implement a simple yet effective algorithm that facilitates pipeline parallelism with multiple early/final-exit training losses located on different pipeline stages, which is not possible in standard pipeline parallelism implemented by Megatron-LM or other frameworks. Moreover, our analytical and empirical study shows that training an early-exit LLM with EE-LLM is almost as efficient as training a standard LLM, in terms of training time and peak GPU memory. This is achieved by various performance optimizations that we design and implement in EE-LLM, especially the ones that leverage idle computational resources in standard pipeline parallelism. Finally, EE-LLM contains some advanced features for fine-grained control or optimization of the training process, including the option of changing early-exit loss weights during training, and a novel method of further improving resource utilization by filling pipeline bubbles with useful computation.

Inference. We design and implement two methods to tackle the major challenge of early-exit LLM inference for autoregressive generation, namely the conflict between early exiting and KV caching (as explained in Section 1). One method is based on KV recomputation, which runs the forward pass with a batch of recent tokens when generating each token. The other method is based on a novel form of pipeline parallelism, which parallelizes the forward pass of the current token at a certain pipeline stage with some KV-related computation of previous tokens at later stages.

3. Training

We first present in Section 3.1 the essentials of scaling up early-exit LLM training with 3D parallelism, including a novel approach to executing backpropagation for the early-exit training objective through pipeline stages. Then, we analyze the training efficiency achieved by EE-LLM in Section 3.2, and explore some advanced features in Section 3.3.

3.1. Backpropagation through pipeline stages

The standard objective function for training an early-exit model is a weighted sum of losses at early and final exits. More formally, to train an early-exit LLM with N exits (including the final output), we aim to solve

$$\min \quad \mathcal{L} := \sum_{i \in [N]} w_i \mathcal{L}_i^{\mathsf{exit}}, \tag{1}$$

where $[N] = \{1, 2, \ldots, N\}$, and each $\mathcal{L}_i^{\mathsf{exit}}$ is a standard loss function for LLM pre-training (e.g. negative log-likelihood of next-token prediction), calculated with outputs from the i-th exit. The loss weights $\{w_i\}$ are hyperparameters specified by the user.

Our implementation for optimizing the loss function in Eq. (1) is compatible with all types of parallelism in Megatron-LM. Indeed, with some engineering efforts, existing functionalities in Megatron-LM for data and tensor/sequence parallelism are directly applicable. The major challenge lies in pipeline parallelism, since it is not immediately clear how to calculate gradients for Eq. (1) via backpropagation through pipeline stages. In a single-GPU scenario with vanilla PyTorch, one simply needs to define `loss` as the weighted sum of losses at all exits, and then run `loss.backward()` for gradient calculation. This is not feasible with pipeline parallelism, since losses $\{\mathcal{L}_i^{\mathsf{exit}}\}$ are now located on different GPUs, and there is only limited P2P communication between each pair of adjacent stages. On the other hand, Megatron-LM only supports backpropagation for a single loss function defined in the last stage.

3.1.1. Methodology

To tackle this challenge, we propose a lightweight algorithm that instructs each pipeline stage to calculate the desired gradients correctly, without any additional communication overhead between stages. To explain our method, let us first re-write the training objective defined in Eq. (1) as $\mathcal{L} = \sum_{i \in [K]} \mathcal{L}_i$, where K represents the number of pipeline stages, and each $\mathcal{L}_i$ is itself a weighted sum of one or multiple early/final-exit losses within Stage i.[1] Consider one data sample $\boldsymbol{x}$ for simplicity, and each loss function is calculated with $\boldsymbol{x}$, i.e. $\mathcal{L}_i = \mathcal{L}_i(\boldsymbol{x})$; in addition, let $\boldsymbol{x}_i$ be the hidden states that Stage i calculates and sends to its next stage during the forward step. Then, during the backward step, Stage i receives some gradient tensor $\boldsymbol{g}_i$ from Stage $i+1$, defines

[1] Obviously, the objective in Eq. (1) is a special case of this formulation. In fact, our analysis in the following allows each $\mathcal{L}_i$ to be a general objective function defined locally in Stage i.

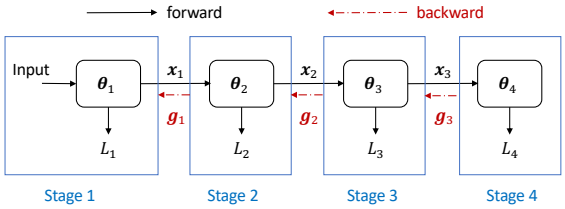


Figure 2: The backpropagation process for an early-exit model partitioned into four pipeline stages.

some *auxiliary loss* $\mathcal{L}_i^{\text{aux}}$, and performs usual backward computation for $\mathcal{L}_i^{\text{aux}}$. The auxiliary losses $\{\mathcal{L}_i^{\text{aux}}\}_{i\in[K]}$ and gradient tensors $\{\boldsymbol{g}_i\}_{i\in[K-1]}$ are defined inductively:

$$\mathcal{L}_K^{\text{aux}} := \mathcal{L}_K; \quad \text{for } i = K-1, K-2, \ldots, 1, \tag{2a}$$

$$\mathcal{L}_i^{\text{aux}} := \mathcal{L}_i + \langle \boldsymbol{g}_i, \boldsymbol{x}_i \rangle, \quad \text{where} \quad \boldsymbol{g}_i := \frac{\partial \mathcal{L}_{i+1}^{\text{aux}}}{\partial \boldsymbol{x}_i}. \tag{2b}$$

Intuitively, the linear term $\langle \boldsymbol{g}_i, \boldsymbol{x}_i \rangle$, i.e. the sum of entrywise product between $\boldsymbol{g}_i$ and $\boldsymbol{x}_i$, summarizes information about the gradients of all losses located in later stages. Note that $\boldsymbol{g}_i$ is regarded by Stage i as a constant tensor, and no gradient is calculated with respect to it. A visualization of this process can be found in Figure 2. It has the same P2P communication scheme as in the case of training a standard LLM with pipeline parallelism; the only difference is how each gradient tensors $\boldsymbol{g}_i$ is defined locally in Stage $i+1$. In the following, we prove that the proposed method leads to correct gradient calculation for the training objective $\mathcal{L}$.

3.1.2. Rationale

Let us first prove the correctness of our solution under the assumption that there is no tied/shared parameter across pipeline stages, just for simplicity; we will see very soon that this assumption is not essential and can be safely discarded.

Proposition 3.1. *Suppose that there is no tied parameter across pipeline stages, and consider the auxiliary losses defined in Eq. (2). Then, for any $i \in [K]$ and any model parameter or activation tensor $\boldsymbol{z}$ in Stage i, it holds that*

$$\frac{\partial \mathcal{L}_i^{\text{aux}}}{\partial \boldsymbol{z}} = \frac{\partial (\sum_{j=i}^K \mathcal{L}_j)}{\partial \boldsymbol{z}}. \tag{3}$$

Notice that for any model parameter $\boldsymbol{z}$ in Stage i, one has $\partial \mathcal{L}_j / \partial \boldsymbol{z} = \boldsymbol{0}$ for any $j < i$, due to the sequential structure of early-exit LLMs (or other deep neural networks). Combining this with $\mathcal{L} = \sum_{i\in[K]} \mathcal{L}_i$ and Eq. (3) yields

$$\frac{\partial \mathcal{L}_i^{\text{aux}}}{\partial \boldsymbol{z}} = \frac{\partial (\sum_{j=i}^K \mathcal{L}_j)}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}},$$

implying correctness of gradient calculation for $\mathcal{L}$.

Proof of Proposition 3.1. The claim of Eq. (3) is obviously true for the base case $i = K$, by definition of $\mathcal{L}_K^{\text{aux}} = \mathcal{L}_K$. Let us prove by induction for the remaining stages. Suppose that Eq. (3) holds true for Stage $i+1$, namely $\partial \mathcal{L}_{i+1}^{\text{aux}} / \partial \boldsymbol{z} = \partial (\sum_{j=i+1}^K \mathcal{L}_j) / \partial \boldsymbol{z}$. To prove Eq. (3) for Stage i, first note that by definition of $\boldsymbol{g}_i$, we have

$$\boldsymbol{g}_i = \frac{\partial \mathcal{L}_{i+1}^{\text{aux}}}{\partial \boldsymbol{x}_i} = \frac{\partial (\sum_{j=i+1}^K \mathcal{L}_j)}{\partial \boldsymbol{x}_i}.$$

Then, for any model parameter or activation tensor $\boldsymbol{z}$ in Stage i, the following holds:

$$\begin{aligned}
\frac{\partial \mathcal{L}_i^{\text{aux}}}{\partial \boldsymbol{z}} &= \frac{\partial (\mathcal{L}_i + \langle \boldsymbol{g}_i, \boldsymbol{x}_i \rangle)}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{z}} + \frac{\partial \langle \boldsymbol{g}_i, \boldsymbol{x}_i \rangle}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{x}_i}{\partial \boldsymbol{z}} \\
&= \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{z}} + \boldsymbol{g}_i \frac{\partial \boldsymbol{x}_i}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{z}} + \frac{\partial (\sum_{j=i+1}^K \mathcal{L}_j)}{\partial \boldsymbol{x}_i} \frac{\partial \boldsymbol{x}_i}{\partial \boldsymbol{z}} \\
&= \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{z}} + \frac{\partial (\sum_{j=i+1}^K \mathcal{L}_j)}{\partial \boldsymbol{z}} = \frac{\partial (\sum_{j=i}^K \mathcal{L}_j)}{\partial \boldsymbol{z}}.
\end{aligned}$$

The above lines simply follow the definition of $\mathcal{L}_i^{\text{aux}}$ and the chain rule. This concludes our proof of Eq. (3) for Stage i, and thus our proof for the proposition. □

Let us move on to relax the assumption. In the broader scenario with tied model parameters, e.g. word embedding matrices (Press & Wolf, 2017), across pipeline stages, gradient calculation via backpropagation is equivalent to the following two-step procedure: (1) compute gradients *as if* all parameters are untied, then (2) sum up and synchronize gradients for tied parameters via all-reduce operations. Hence our proposed auxiliary-loss approach, when applied to the first part of this two-step procedure, is still valid.

3.1.3. Integration with the 1F1B Schedule

We have shown how to modify the forward and backward steps for *each* microbatch, in order to execute backpropagation of the early-exit training loss through pipeline stages. In principle, this lightweight modification can be integrated with general pipeline schedules of forward and backward steps for *multiple* microbatches. The classical 1F1B (one-forward-one-backward) schedule, also called PipeDream-Flush (Narayanan et al., 2019; Fan et al., 2021; Narayanan et al., 2021b), achieves a good balance of algorithmic simplicity, training time, memory usage and communication latency, in comparison to other schedules such as GPipe (Huang et al., 2019) (larger activation memory) or interleaved 1F1B (Narayanan et al., 2021b) (higher memory and communication requirements). Hence for concreteness, we focus on the 1F1B schedule throughout this work. Within one iteration of this schedule, each stage goes through a *warm-up* phase (forward steps of the beginning microbatches), a *steady* 1F1B phase, and a *cool-down* phase (backward steps of the final microbatches). A visualization can be found in Figure 3(a).

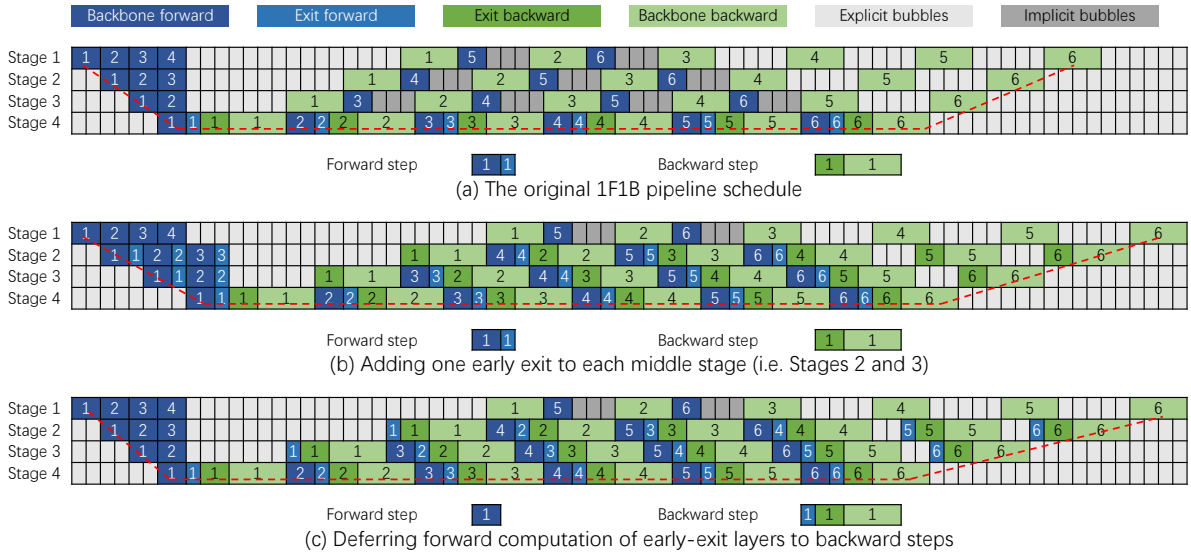


Figure 3: One iteration of the 1F1B pipeline schedule, in a setting with $P = 4$ pipeline stages and $M = 6$ microbatches per batch. At the top of this figure, "Backbone forward/backward" stands for computation of Transformer layers on the backbone, while "Exit forward/backward" stands for computation of early-exit or final-exit layers. The number in each block denotes the index of the corresponding microbatch. Critical paths are marked by dashed red lines. From Figure (a) to (b), additional "Exit forward/backward" blocks are added, due to the introduction of early exits to middle stages. From Figure (b) to (c), the order of computation is slightly adjusted for the purpose of reducing memory usage. For clarity, we ignore computation related to the input embedding layer, and P2P communication latency between pipeline stages.

3.2. Training efficiency

At first glance, one might expect that adding early exits to a standard LLM will incur a training overhead, in terms of time and memory, that is (at least) proportional to the number of additional model parameters. Fortunately, this is not the case for training with pipeline parallelism. We discover that computation related to the additional early exits can leverage idle resources in the original 1F1B schedule, thus causing *negligible overhead to training time and zero overhead to peak GPU memory* under mild conditions.

To explain this, let us first identify three types of idle resources in the original 1F1B schedule.

- **Explicit bubbles**, i.e. the light gray areas in Figure 3(a), during which GPUs are idle.
- **Implicit bubbles**, i.e. the dark gray areas in Figure 3(a), caused by load imbalance across pipeline stages. In particular, the last pipeline stage requires additional computation for the output layer.
- **Idle memory**, caused by unbalanced memory usage: earlier stages have to save the intermediate activations for more microbatches, and the first/last stage need to store an extra input/output layer, plus the corresponding gradients and optimizer states. As a result, the first stage is typically the bottleneck of peak memory usage.

Now, suppose that we choose k middle stages and add one minimalistic early-exit layer to each of them. As shown in Figure 3(b), by utilizing implicit bubbles, computation of k early exits will increase the training time per iteration by only $k\times(f_{\text{EE}}+b_{\text{EE}})$, where f_{EE} (resp. b_{EE}) represent the time for one forward (resp. backward) pass of one microbatch for one early-exit layer. Moreover, memory usage by model parameters for each middle stage will not exceed that of the first or last stage. Deferring forward computation of early exits to backward steps, as shown in Figure 3(c), further decreases the memory overhead by early-exit logits in Stage $i \in [P]$ from $s \times b \times V \times (P - i + 1)$ to $s \times b \times V$, where s is the sequence length, b is the microbatch size, V is the vocabulary size, and $P - i + 1$ is the number of in-flight microbatches (Korthikanti et al., 2022) for Stage i. Putting these together, the peak memory usage across stages remains unchanged, as long as $s \times b \times V$ is less than the activation memory of *all* backbone Transformer layers within one stage for one microbatch.

Due to limited space, we defer detailed and formal analysis of training efficiency in broader settings to Appendix A.

3.3. Advanced features

EE-LLM incorporates some advanced features that can potentially improve the training process. One of them is the option of changing early-exit loss weights during training, just like learning rate or other hyperparameters. Another one is a novel method of improving resource utilization by filling explicit bubbles with *partial* forward and backward computation for *additional microbatches*, which is visualized in Figure 4. One can prove formally that such

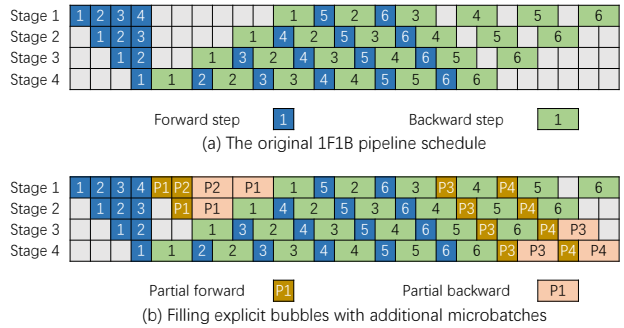


Figure 4: The proposed method of filling bubbles with additional microbatches. In this example, P1 and P2 go through the forward and backward passes for the first few stages, while P3 and P4 go through the full forward pass, followed by the backward pass for the last few stages.

additional computation provides useful gradient information for optimizing the early-exit training objective, without increasing the training time per iteration. More details of both features can be found in Appendix C.

4. Inference

This section first explains the major challenge faced by early-exit LLM inference for autoregressive generation, and a few recent attempts to resolve it. Then, we introduce a novel solution based on a new type of pipeline parallelism. Throughout this section, we focus on the latency-oriented setting, with a batch size of 1 during sequence generation.

Main challenge: KV caching. Accelerating inference with early exiting is, in principle, orthogonal to and compatible with many other common techniques of acceleration, such as kernel fusion, FlashAttention (Dao et al., 2022), quantization (Yao et al., 2022; Xiao et al., 2023), among others (Kim et al., 2023a). For autoregressive generation, one exception is KV caching, i.e. saving keys and values of all attention layers for previously generated tokens, which reduces redundant computation at the cost of higher memory usage. This is contradictory to vanilla early-exit inference: if the current token is generated via early exiting, then its KV caches in later layers are missing, which hinders the generation of future tokens that go beyond the exiting layer of the current token. This challenge has been well recognized in the literature, and several approaches have been recently proposed to resolve it, including state propagation (Elbayad et al., 2020; Li et al., 2021; Schuster et al., 2022), SkipDecode (Corro et al., 2023), and synchronized parallel decoding (Bae et al., 2023; Tang et al., 2023). See Appendix F for more discussion on these methods.

A variant of the last method, which we call *KV recomputation*, is implemented in `EE-LLM`. In this approach, we maintain a list of the most recent tokens that have missing

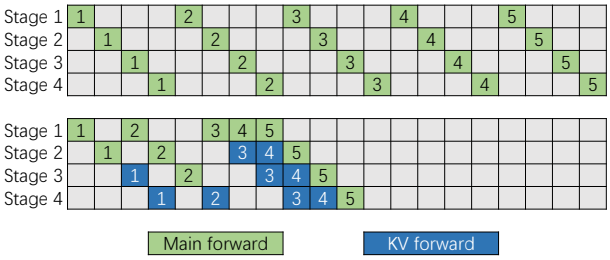


Figure 5: Standard full-model inference (top) and our pipeline-based early-exit inference (bottom). Numbers in the blocks denote the tokens within one generated sequence. For simplicity of visualization, we assume here that (1) each early exit is located at the end of some pipeline stage, and (2) the latency for generating each token is the same (while in practice, generating the first token via the prefilling phase usually takes longer than generating another token during the decoding phase).

KV caches in deep layers due to early exiting. During each forward pass, we include these early-exit tokens in the current forward pass, which allows for direct recomputation of the KV caches for these tokens and thus avoids the issue of missing KV caches. It relies on the batching effect of GPU computation for early-exit acceleration, yet might not achieve any acceleration on other hardware platforms, due to its high theoretical complexity.

New solution: pipeline parallelism. We propose a novel type of pipeline parallelism to tackle the aforementioned challenge. The key idea is that, in the process of inference with multiple pipeline stages, the following two processes run *in parallel* whenever the model decides to do early exiting for the current token at a certain exit:

- The generated token is sent back to the first stage, and the forward pass for generating the next token is started immediately;
- The full-model forward pass of the current token is continued from the exiting layer, which fulfills its KV caches in all later layers.

See Figure 5 for a visualization of this approach. Although each token essentially goes through a forward pass of the full model, the computation after the exiting layer is *parallelized* with the computation of later tokens, which is how acceleration is achieved in this approach. It can be checked that the inference latency for generating one token at a certain exit matches exactly the time needed for the forward computation before returning an output at that exit, unless the selected exit is located in the middle of the first pipeline stage, in which case generation of the next token has to wait until the forward pass of the first stage for the current token is completed. Note that this is true not just in practice but also for the *theoretical* time complexity,

without relying on the batching effect of GPU computation like KV recomputation does. An implementation of this pipeline-based inference method is provided in EE-LLM. One potential limitation of the proposed method is that it requires multiple devices to facilitate pipeline parallelism, although parallelism within a single GPU or other device might be possible with more advanced implementation.

5. Experiments

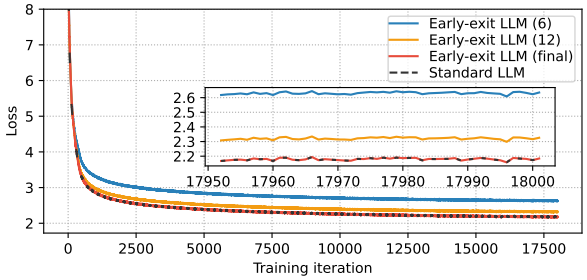
This section provides an empirical evaluation of the training and inference efficiency achieved by EE-LLM.

5.1. Training

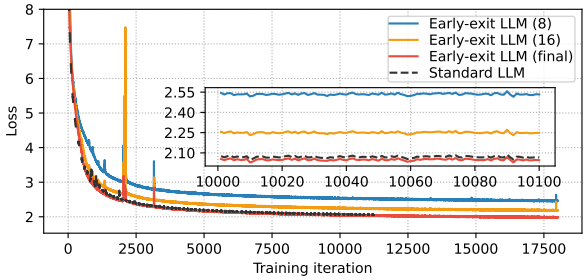
In the following experiments, we empirically investigate the convergence of training early-exit models with EE-LLM, as well as the training efficiency of EE-LLM for early-exit LLMs up to an unprecedented scale of 30B. This scale is only limited by the hardware resources available to us, namely an 8-node cluster with 8 Nvidia A100-80GB GPUs in each node and hence 64 GPUs in total. We use a random subset of the pre-training data provided by Data-Juicer (Data-Juicer, 2023; Chen et al., 2023). Experiments for models of the same size use the same subset and order of data. In all cases, we use the Adam optimizer (Kingma & Ba, 2014) with $\beta_1 = 0.9, \beta_2 = 0.95, \epsilon = 10^{-8}$, and a cosine schedule for the learning rate, with a maximum value of 3×10^{-4}.

Convergence of training losses. We first consider a 1.3B GPT Transformer with 24 layers, add one minimalistic early-exit layer without layer normalization to the 1/4 depth and the other to the 1/2 depth, set their early-exit loss weights to 1/4 and 1/2 respectively (while the final-exit loss has a weight of 1), and tie all input and output embedding matrices. A standard LLM of the same architecture is trained using the same hyperparameters and pre-training data. We further train a 7B early-exit model with 32 layers using similar configurations, except that early-exit loss weights are set to 0.1 and 0.2, and all embedding matrices are untied. A standard 7B model is trained similarly. The batch size and sequence length are both set to 2048.

Figure 6 shows the convergence of early and final-exit training losses, i.e. negative log-likelihood of next-token prediction, for both standard and early-exit LLMs. All loss curves decay at a similar pace, and unsurprisingly, the early-exit losses are slightly higher than the final-exit loss for each model. Interestingly, the final-exit loss curve of each early-exit model is close to (or even slightly below) that of the standard model, suggesting that optimizing for early-exit losses might not hurt the full-model output in our setting. We also observe more spikes in the loss curves for the 7B models than for the 1.3B models, possibly because (1) we choose to untie the embedding matrices and use smaller early-exit loss weights, both of which incur weaker regularization for the training process; and (2) layer normalization, which is known to stabilize the training of LLMs, is not included in the minimalistic early-exit layers of our 7B model. Empirical results for the convergence of training losses under more general early-exit configurations can be found in Appendix B.3.



(a) 1.3B, 24 layers

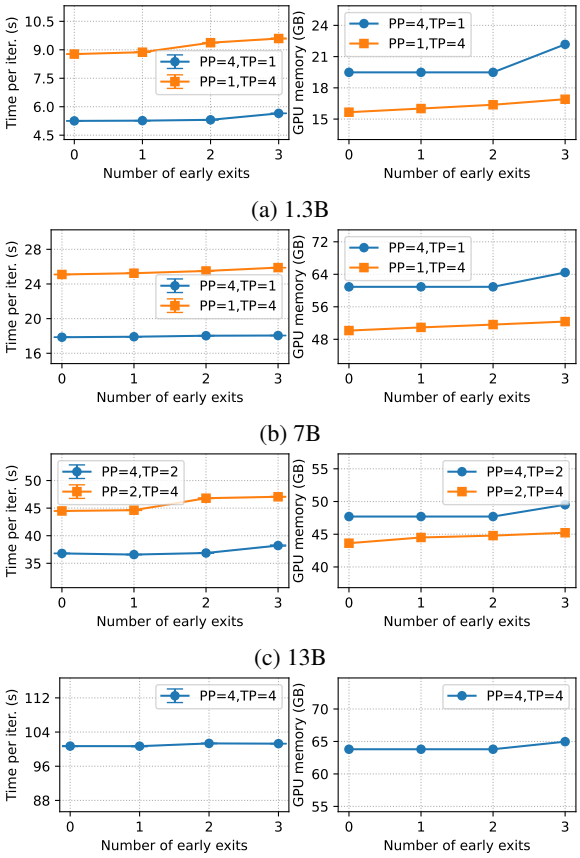


(b) 7B, 32 layers

Figure 6: Convergence of early-exit/final-exit training losses. Each curve is annotated with the index of the Transformer layer that the corresponding exit is connected to.

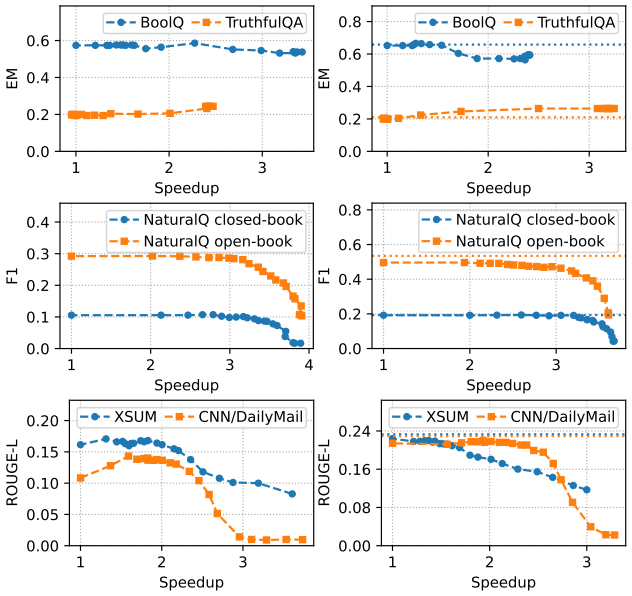
Training efficiency. Starting with a standard GPT Transformer of size ranging from 1.3B to 30B, we increase the number of early exits from 0 to 3. Minimalistic early exits are added one by one to specific locations in the following order: (1) to the 1/4 depth; (2) to the 1/2 depth; (3) to the hidden states right before the first Transformer layer, which is always located in the first pipeline stage. Based on the performance optimizations proposed in Appendix A, when an early exit is inserted into the middle of two layers located on two consecutive pipeline stages, we always add it to the beginning of the latter stage. We set the global batch size to 2048, microbatch size to 2 (for 1.3B and 7B models) or 1 (for 13B and 30B models), and sequence length to 2048. The data parallelism degree is fixed at 4, while tensor (TP) and pipeline (PP) parallelism degrees take various values.

The empirical results illustrated in Figure 7 matches our analytical study in Section 3.2. In particular, training time increases with the number of added early exits, but at a slower rate when pipeline parallelism is enabled, thanks to the proposed utilization of implicit bubbles. Without pipeline



(a) 1.3B (b) 7B (c) 13B (d) 30B

Figure 7: Training time per iteration and peak GPU memory vs. the number of added early exits. Note that wall-clock time can be impacted by other workloads on the same GPU cluster when our experiments were conducted, which inevitably causes perturbations to our numerical results.



(a) 1.3B (b) 7B

Figure 8: Evaluation scores and relative speedup with our early-exit models. For each curve, speedup increases as the confidence threshold decreases from left to right. The right column also includes horizontal dotted lines that represent the performance of a 7B standard LLM trained under the same configurations as those for our 7B early-exit model.

parallelism, peak GPU memory increases with the number of early exits; on the other hand, with the pipeline parallelism degree set to 4, peak memory remains unchanged as early exits are added to middle pipeline stages, and only increases when the last early exit is added to the first stage.

5.2. Inference

In the experiments below, we verify the downstream performance of the pipeline-based approach proposed in Section 4, with the number of pipeline stages set to 4. A server with 4 Nvidia A100-40GB GPUs is used for inference. We use the 1.3B and 7B early-exit LLMs from the previous experiment on convergence of training losses in Section 5.1, which have been pre-trained from scratch using 300B and 150B tokens respectively, without further fine-tuning or alignment. Since designing the decoding method or exit condition is not our focus, we simply adopt greedy decoding and a basic confidence-based exit condition: at each early exit, if the maximum probability of next-token prediction is above a pre-defined threshold, then the model chooses the most likely token and moves on to generate the next one. The trade-off between output quality and speed is controlled by this threshold. When the threshold is set to 1, early-exit layers are disabled, so that the time complexity of a full-model forward pass matches that of a standard LLM[2]. Inference latency in this case is used as the baseline for calculating relative speedup. For benchmarking, we evaluate our models with HELM (Liang et al., 2023) in six tasks: BoolQ (Clark et al., 2019), TruthfulQA (Lin et al., 2022), NaturalQuestions open-book and closed-book (Kwiatkowski et al., 2019), XSUM (Narayan et al., 2018), and CNN/DailyMail. The first four tasks are question-answering tasks, while the last two are summarization tasks. We use EM (ratio of exact match with the correct answers) as the metric for BoolQ and TruthfulQA, F1 for NaturalQuestions open-book/closed-book, and ROUGE-L (a measure of similarity to the reference summaries) for XSUM and CNN/DailyMail.

Figure 8 demonstrates the scores and speedup for pipeline-based inference with varying confidence thresholds. Encouragingly, in many cases, inference with early exiting achieves 2× or higher speedup compared to full-model in-

[2]We observed empirically that the actual wall-clock latency between these two cases might differ by up to 2%, which is negligible compared to the early-exit speedup.

ference, with comparable or even better evaluation scores. Table 3 provides example texts generated by our 7B early-exit model, showing that speedup can be achieved with minor or no impact on the resulting sequence. Table 4 further demonstrates the confidence at each exit for each token, confirming the existence of easy tokens that can be predicted correctly with high confidence at early exits. See also Appendix B.1 for an empirical comparison between the pipeline-based method and KV recomputation, which are shown to perform similarly in our setting. It is worth noting that performance of early-exit inference might be improved in many ways, e.g. using early exits of different structures, other decoding mechanisms and exit conditions, or simply more extensive and sufficient pre-training. We also conjecture that early exiting can, in principle, bring higher speedup for larger LLMs, as expressivity of early exits grows with the model capacity, which allows them to make confident and accurate predictions for a larger proportion of tokens.

6. Conclusions

We have introduced EE-LLM, a system for large-scale training and inference of early-exit LLMs with 3D parallelism. For training, we have presented how to execute backpropagation of the early-exit training objective across pipeline stages, performance optimizations for minimizing the computational overhead compared to training standard LLMs, and advanced features for fine-grained control and optimization of the training process. For inference, we have introduced our design and implementation of two inference methods, one based on KV recomputation and the other based on a new type of pipeline parallelism, both of which are compatible with KV caching for autoregressive generation. Along the way, we have discovered some interesting chemical reactions between early exiting and pipeline parallelism, which is likely because it is both *along the depth dimension* that an early-exit model executes forward computation selectively and gets partitioned into pipeline stages. We hope that EE-LLM will be a helpful tool for future study and applications of early-exit LLMs at scales as large as those of standard LLMs. It is worth noting that many ideas presented in this work can be extended to more general neural network architectures, deep learning frameworks, hardware platforms, or other broader settings. It would be exciting to see further developments in these aspects.

Acknowledgements

We would like to thank the reviewers for their constructive and valuable feedback that helps improve this work.

Impact Statement

This paper presents work whose goal is to advance techniques for training and inference of early-exit large language models. Therefore, it bears potential societal consequences similar to those of prior literature on the technical aspects of LLM training and inference, e.g. misuse of the resulting pre-trained LLMs. Other than that, we see nothing that we feel must be specifically highlighted here.

References

Ba, J., Kiros, J. R., and Hinton, G. E. Layer normalization. *ArXiv*, abs/1607.06450, 2016.

Bae, S., Ko, J., Song, H., and Yun, S.-Y. Fast and robust early-exiting framework for autoregressive language models with synchronized parallel decoding. In *EMNLP*, 2023.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Bengio, E., Bacon, P.-L., Pineau, J., and Precup, D. Conditional computation in neural networks for faster models. *ArXiv*, abs/1511.06297, 2015.

Bengio, Y., Léonard, N., and Courville, A. C. Estimating or propagating gradients through stochastic neurons for conditional computation. *ArXiv*, abs/1308.3432, 2013.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., Brynjolfsson, E., Buch, S., Card, D., Castellon, R., Chatterji, N. S., Chen, A. S., Creel, K. A., Davis, J., Demszky, D., Donahue, C., Doumbouya, M., Durmus, E., Ermon, S., Etchemendy, J., Ethayarajh, K., Fei-Fei, L., Finn, C., Gale, T., Gillespie, L., Goel, K., Goodman, N. D., Grossman, S., Guha, N., Hashimoto, T., Henderson, P., Hewitt, J., Ho, D. E., Hong, J., Hsu, K., Huang, J., Icard, T. F., Jain, S., Jurafsky, D., Kalluri, P., Karamcheti, S., Keeling, G., Khani, F., Khattab, O., Koh, P. W., Krass, M. S., Krishna, R., Kuditipudi, R., Kumar, A., Ladhak, F., Lee, M., Lee, T., Leskovec, J., Levent, I., Li, X. L., Li, X., Ma, T., Malik, A., Manning, C. D., Mirchandani, S., Mitchell, E., Munyikwa, Z., Nair, S., Narayan, A., Narayanan, D., Newman, B., Nie, A., Niebles, J. C., Nilforoshan, H., Nyarko, J. F., Ogut, G., Orr, L. J., Papadimitriou, I., Park, J. S., Piech, C., Portelance, E., Potts, C., Raghunathan, A., Reich, R., Ren, H., Rong, F., Roohani, Y. H., Ruiz, C., Ryan, J., R'e, C., Sadigh, D., Sagawa, S., Santhanam, K., Shih, A., Srinivasan, K. P., Tamkin, A., Taori, R., Thomas, A. W., Tramèr, F., Wang, R. E., Wang, W., Wu, B., Wu, J., Wu, Y., Xie, S. M., Yasunaga, M., You, J., Zaharia, M. A.,

Zhang, M., Zhang, T., Zhang, X., Zhang, Y., Zheng, L., Zhou, K., and Liang, P. On the opportunities and risks of foundation models. *ArXiv*, abs/2108.07258, 2021.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners. In *NeurIPS*, 2020.

Chen, D., Huang, Y., Ma, Z., Chen, H., Pan, X., Ge, C., Gao, D., Xie, Y., Liu, Z., Gao, J., Li, Y., Ding, B., and Zhou, J. Data-juicer: A one-stop data processing system for large language models. *ArXiv*, abs/2309.02033, 2023.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost. *ArXiv*, abs/1604.06174, 2016.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023.

Clark, C., Lee, K., Chang, M., Kwiatkowski, T., Collins, M., and Toutanova, K. Boolq: Exploring the surprising difficulty of natural yes/no questions. In *NAACL*, 2019.

Corro, L. D., Giorno, A. D., Agarwal, S., Yu, T., Awadallah, A. H., and Mukherjee, S. Skipdecode: Autoregressive skip decoding with batching and caching for efficient llm inference. *ArXiv*, abs/2307.02628, 2023.

Dai, Y., Pan, R., Iyer, A., Li, K., and Netravali, R. Apparate: Rethinking early exits to tame latency-throughput tensions in ml serving. *ArXiv*, abs/2312.05385, 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Re, C. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *NeurIPS*, 2022.

Data-Juicer. Refined open source dataset by data-juicer. https://github.com/alibaba/data-juicer/blob/main/configs/data_juicer_recipes/README.md, 2023.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pp. 4171–4186, 2019.

Din, A. Y., Karidi, T., Choshen, L., and Geva, M. Jump to conclusions: Short-cutting transformers with linear transformations. *ArXiv*, abs/2303.09435, 2023.

Duggal, R., Freitas, S., Dhamnani, S., Chau, D. H., and Sun, J. Elf: An early-exiting framework for long-tailed classification. *ArXiv*, abs/2006.11979, 2020.

Elbayad, M., Gu, J., Grave, E., and Auli, M. Depth-adaptive transformer. In *ICLR*, 2020.

Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., Diao, L., Liu, X., and Lin, W. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*, pp. 431–445, 2021.

Fedus, W., Zoph, B., and Shazeer, N. M. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *J. Mach. Learn. Res.*, 23:120:1–120:39, 2021.

Gera, A., Friedman, R., Arviv, O., Gunasekara, C., Sznajder, B., Slonim, N., and Shnarch, E. The benefits of bad advice: Autocontrastive decoding across model layers. In *ACL*, 2023.

Graves, A. Adaptive computation time for recurrent neural networks. *ArXiv*, abs/1603.08983, 2016.

Han, Y., Huang, G., Song, S., Yang, L., Wang, H., and Wang, Y. Dynamic neural networks: A survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 44(11):7436–7456, 2022.

Hou, L., Huang, Z., Shang, L., Jiang, X., Chen, X., and Liu, Q. Dynabert: Dynamic BERT with adaptive width and depth. In *NeurIPS*, 2020.

Hu, B., Zhu, Y., Li, J., and Tang, S. Smartbert: A promotion of dynamic early exiting mechanism for accelerating bert inference. In *IJCAI*, 2023.

Huang, G., Chen, D., Li, T., Wu, F., van der Maaten, L., and Weinberger, K. Q. Multi-scale dense networks for resource efficient image classification. In *ICLR*, 2018.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M. X., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, pp. 103–112, 2019.

Ilhan, F., Su, G., and Liu, L. Scalefl: Resource-adaptive federated learning with heterogeneous clients. In *CVPR*, pp. 24532–24541, 2023.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive mixtures of local experts. *Neural Computation*, 3:79–87, 1991.

Kaya, Y., Hong, S., and Dumitras, T. Shallow-deep networks: Understanding and mitigating network overthinking. In *ICML*, volume 97, pp. 3301–3310, 2019.

Kim, S., Hooper, C., Wattanawong, T., Kang, M., Yan, R., Genç, H., Dinh, G., Huang, Q., Keutzer, K., Mahoney, M. W., Shao, Y. S., and Gholami, A. Full stack optimization of transformer inference: a survey. *ArXiv*, abs/2302.14017, 2023a.

Kim, S., Mangalam, K., Moon, S., Malik, J., Mahoney, M. W., Gholami, A., and Keutzer, K. Speculative decoding with big little decoder. In *NeurIPS*, 2023b.

Kim, Y., Denton, C., Hoang, L., and Rush, A. M. Structured attention networks. In *ICLR*, 2017.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2014.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L. C., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *ArXiv*, abs/2205.05198, 2022.

Kwiatkowski, T., Palomaki, J., Redfield, O., Collins, M., Parikh, A. P., Alberti, C., Epstein, D., Polosukhin, I., Devlin, J., Lee, K., Toutanova, K., Jones, L., Kelcey, M., Chang, M., Dai, A. M., Uszkoreit, J., Le, Q., and Petrov, S. Natural questions: a benchmark for question answering research. *Trans. Assoc. Comput. Linguistics*, 7:452–466, 2019.

Langedijk, A., Mohebbi, H., Sarti, G., Zuidema, W. H., and Jumelet, J. Decoderlens: Layerwise interpretation of encoder-decoder transformers. *ArXiv*, abs/2310.03686, 2023.

Laskaridis, S., Kouris, A., and Lane, N. D. Adaptive inference through early-exit networks: Design, challenges and directions. In *International Workshop on Embedded and Mobile Deep Learning*, 2021.

Lee, C.-Y., Xie, S., Gallagher, P. W., Zhang, Z., and Tu, Z. Deeply-supervised nets. In *AISTATS*, 2014.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *ICML*, 2022.

Li, S. and Hoefler, T. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *SC*, pp. 27, 2021.

Li, X., Shao, Y., Sun, T., Yan, H., Qiu, X., and Huang, X. Accelerating bert inference for sequence labeling via early-exit. In *ACL*, 2021.

Liang, P., Bommasani, R., Lee, T., Tsipras, D., Soylu, D., Yasunaga, M., Zhang, Y., Narayanan, D., Wu, Y., Kumar, A., Newman, B., Yuan, B., Yan, B., Zhang, C., Cosgrove, C., Manning, C. D., R'e, C., Acosta-Navas, D., Hudson, D. A., Zelikman, E., Durmus, E., Ladhak, F., Rong, F., Ren, H., Yao, H., Wang, J., Santhanam, K., Orr, L. J., Zheng, L., Yuksekgonul, M., Suzgun, M., Kim, N. S., Guha, N., Chatterji, N. S., Khattab, O., Henderson, P., Huang, Q., Chi, R., Xie, S. M., Santurkar, S., Ganguli, S., Hashimoto, T., Icard, T. F., Zhang, T., Chaudhary, V., Wang, W., Li, X., Mai, Y., Zhang, Y., and Koreeda, Y. Holistic evaluation of language models. *Annals of the New York Academy of Sciences*, 1525:140 – 146, 2023.

Lin, S., Hilton, J., and Evans, O. Truthfulqa: Measuring how models mimic human falsehoods. In *ACL*, pp. 3214–3252, 2022.

Liu, D., Kan, M., Shan, S., and Chen, X. A simple romance between multi-exit vision transformer and token reduction. In *ICLR*, 2024.

Liu, W., Zhou, P., Wang, Z., Zhao, Z., Deng, H., and Ju, Q. Fastbert: a self-distilling BERT with adaptive inference time. In *ACL*, pp. 6035–6044, 2020.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization. In *EMNLP*, pp. 1797–1807, 2018.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for DNN training. In *SOSP*, pp. 1–15, 2019.

Narayanan, D., Phanishayee, A., Shi, K., Chen, X., and Zaharia, M. Memory-efficient pipeline-parallel DNN training. In *ICML*, volume 139, pp. 7937–7947, 2021a.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on GPU clusters using megatron-lm. In *SC*, pp. 58, 2021b.

OpenAI. Gpt-4 technical report. *ArXiv*, abs/2303.08774, 2023.

Osawa, K., Li, S., and Hoefler, T. Pipefisher: Efficient training of large language models using pipelining and fisher information matrices. In *MLSys*, 2023.

Parikh, A., Täckström, O., Das, D., and Uszkoreit, J. A decomposable attention model for natural language inference. In *EMNLP*, pp. 2249–2255, 2016.

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference. *ArXiv*, abs/2211.05102, 2022.

Press, O. and Wolf, L. Using the output embedding to improve language models. In *EACL*, pp. 157–163, 2017.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners, 2019.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC*, 2019.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD*, pp. 3505–3506, 2020.

Santilli, A., Severino, S., Postolache, E., Maiorca, V., Mancusi, M., Marin, R., and Rodolà, E. Accelerating transformer inference for translation via parallel decoding. In *ACL*, 2023.

Scardapane, S., Scarpiniti, M., Baccarelli, E., and Uncini, A. Why should we add early exits to neural networks? *Cognitive Computation*, 12:954 – 966, 2020.

Schuster, T., Fisch, A., Jaakkola, T. S., and Barzilay, R. Consistent accelerated inference via confident adaptive transformers. In *EMNLP*, pp. 4962–4979, 2021.

Schuster, T., Fisch, A., Gupta, J., Dehghani, M., Bahri, D., Tran, V., Tay, Y., and Metzler, D. Confident adaptive language modeling. In *NeurIPS*, 2022.

Schwartz, R., Stanovsky, G., Swayamdipta, S., Dodge, J., and Smith, N. A. The right tool for the job: Matching model and instance complexities. In *ACL*, pp. 6640–6651, 2020.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. Mesh-tensorflow: Deep learning for supercomputers. In *NIPS*, pp. 10435–10444, 2018.

Shazeer, N. M., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*, 2017.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *ArXiv*, abs/1909.08053, 2019.

Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V. A., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *ArXiv*, abs/2201.11990, 2022.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. E., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *CVPR*, 2015.

Tang, P., Zhu, P., Li, T., Appalaraju, S., Mahadevan, V., and Manmatha, R. Deed: Dynamic early exit on decoder for accelerating encoder-decoder transformer models. *ArXiv*, abs/2311.08623, 2023.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *ACM Comput. Surv.*, 55(6), 2022. doi: 10.1145/3530811.

Team, I. Internlm: A multilingual language model with progressively enhanced capabilities. https://github.com/InternLM/InternLM, 2023.

Teerapittayanon, S., McDanel, B., and Kung, H. T. Branchynet: Fast inference via early exiting from deep neural networks. In *ICPR*, pp. 2464–2469, 2016.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023a.

Touvron, H., Martin, L., Stone, K. R., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D. M., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A. S., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I. M., Korenev, A. V., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith,

E. M., Subramanian, R., Tan, X., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *ArXiv*, abs/2307.09288, 2023b.

Varshney, N., Chatterjee, A., Parmar, M., and Baral, C. Accelerating llama inference by enabling intermediate layer decoding via instruction tuning with lite. *ArXiv*, abs/2310.18581, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *NeurIPS*, pp. 5998–6008, 2017.

Wang, H., Wang, Y., Liu, T., Zhao, T., and Gao, J. Hadskip: Homotopic and adaptive layer skipping of pre-trained language models for efficient inference. In *EMNLP*, pp. 4283–4294, 2023.

Wang, J., Chen, K., Chen, G., Shou, L., and McAuley, J. Skipbert: Efficient inference with shallow layer skipping. In *ACL*, 2022.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *ICML*, volume 202, pp. 38087–38099, 2023.

Xin, J., Tang, R., Lee, J., Yu, Y., and Lin, J. Deebert: Dynamic early exiting for accelerating BERT inference. In *ACL*, pp. 2246–2251, 2020.

Xin, J., Tang, R., Yu, Y., and Lin, J. Berxit: Early exiting for BERT with better fine-tuning and extension to regression. In *EACL*, pp. 91–104, 2021.

Xu, C. and McAuley, J. J. A survey on dynamic neural networks for natural language processing. In *EACL*, pp. 2325–2336, 2023.

Yang, F., Peng, S., Sun, N., Wang, F., Tan, K., Wu, F., Qiu, J., and Pan, A. Holmes: Towards distributed training across clusters with heterogeneous nic environment. *ArXiv*, abs/2312.03549, 2023.

Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In *NeurIPS*, 2022.

Yuan, B., He, Y., Davis, J., Zhang, T., Dao, T., Chen, B., Liang, P., Ré, C., and Zhang, C. Decentralized training of foundation models in heterogeneous environments. In *NeurIPS*, 2022.

Zeng, D., Du, N., Wang, T., Xu, Y., Lei, T., Chen, Z., and Cui, C. Learning to skip for language modeling. *ArXiv*, abs/2311.15436, 2023.

Zhang, B. and Sennrich, R. Root mean square layer normalization. In *NeurIPS*, pp. 12360–12371, 2019.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M. T., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022.

Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., Du, Y., Yang, C., Chen, Y., Chen, Z., Jiang, J., Ren, R., Li, Y., Tang, X., Liu, Z., Liu, P., Nie, J., and rong Wen, J. A survey of large language models. *ArXiv*, abs/2303.18223, 2023a.

Zhao, Y., Xie, Z., Zhuang, C., and Gu, J. Lookahead: An inference acceleration framework for large language model with lossless generation accuracy. *ArXiv*, abs/2312.12728, 2023b.

Zheng, L., Li, Z., Zhang, H., Zhuang, Y., Chen, Z., Huang, Y., Wang, Y., Xu, Y., Zhuo, D., Xing, E. P., Gonzalez, J. E., and Stoica, I. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *OSDI*, pp. 559–578, 2022.

Zhong, Z., Wang, J., Bao, W., Zhou, J., Zhu, X., and Zhang, X. Semi-hfl: semi-supervised federated learning for heterogeneous devices. *Complex & Intelligent Systems*, 9: 1995–2017, 2022.

Zhou, W., Xu, C., Ge, T., McAuley, J. J., Xu, K., and Wei, F. BERT loses patience: Fast and robust inference with early exit. In *NeurIPS*, 2020.

Structure of the appendix

This appendix is organized as follows. Appendix A supplements a detailed and formal analysis of training efficiency, while Appendix B provides additional experiments and empirical results. Appendices C and D include details about advanced features and implementation of EE-LLM. Extended preliminaries and related works can be found in Appendices E and F.

A. Training efficiency

This section is an extended version of Section 3.2. In the following, we analyze the training time and peak GPU memory of training an early-exit model with pipeline parallelism, and propose some performance optimizations. We refer readers to the literature (e.g. the Megatron-LM series (Shoeybi et al., 2019; Narayanan et al., 2021b; Korthikanti et al., 2022)) for thorough study of training efficiency with 3D parallelism for standard LLMs without early exits; we will use that as a baseline, and focus on the additional computational overhead caused by early exits.

Let us first identify the major sources of low resource utilization in the original 1F1B pipeline schedule for training a standard LLM, which lays the foundation for our analysis later.

- **Explicit bubbles,** i.e. light gray areas in Figure 3(a), during which GPUs are idle. This is the most notable and well recognized source of low resource utilization in pipeline parallelism.
- **Implicit bubbles,** i.e. dark gray areas in Figure 3(a). This is caused by load imbalance across pipeline stages, even though Transformer layers are evenly divided into stages in Megatron-LM[3]. In particular, the first stage has the additional computation for the input embedding layer, and more importantly, the last pipeline stage has the additional computation for the output logits (via the output embedding layer) as well as the training loss. For LLMs, these additional computational costs are not negligible, primarily due to large vocabulary sizes.
- **Idle memory.** Memory usage is also unbalanced across pipeline stages: earlier stages in the 1F1B schedule have to save the intermediate activations for more microbatches, and the first/last stage has to save an extra input/output embedding layer, plus the corresponding gradients and optimizer states. As a result, the first stage is typically the bottleneck of peak memory usage, while later stages have idle memory (Korthikanti et al., 2022).

Remark A.1. In our analysis, we assume no activation recomputation (Chen et al., 2016; Korthikanti et al., 2022) for simplicity. For clarity, we also ignore the P2P communication latency between pipeline stages, which is generally not the major concern for the efficiency of pipeline parallelism.

A.1. Utilization of idle resources

At first glance, one might expect that adding early exits to an LLM will incur a training overhead, in terms of time and memory, that is (at least) proportional to the number of additional model parameters. Fortunately, this is not the case for training with pipeline parallelism, based on the above analysis of idle resources in the 1F1B pipeline schedule. Indeed, adding one minimalistic early-exit layer (which has the same structure as the final output layer) to some middle (i.e. not the first or last) stage will only make its model size and theoretical forward/backward time match exactly those of the last stage. Therefore, the aforementioned implicit bubbles and some of the idle memory can be automatically utilized for computation related to the early-exit layers, leading to more balanced load across pipeline stages.

More specifically, the overhead to training time caused by additional early exits can be negligible. If we choose k middle stages and add one minimalistic early-exit layer to each of them, then the training time per iteration, i.e. time for processing one data batch, will (in theory) increase only by $k \times (f_{\mathsf{EE}} + b_{\mathsf{EE}})$, where f_{EE} and b_{EE} represent the time needed for one forward and backward pass of one microbatch for one minimalistic early-exit layer, respectively. To see this, first notice that the computation of early-exit layers in the steady 1F1B phase can perfectly fit into the implicit bubbles. Therefore, the critical path remains the same as in the case without early exits, which consists of (1) the forward steps on all stages for the first microbatch, (2) the 1F1B steady phase on the last stage, and (3) the backward steps on all stages for the last microbatch. The early-exit layers, located separately on k middle stages, will cause a $k \times f_{\mathsf{EE}}$ overhead to the first part of the critical path, and a $k \times b_{\mathsf{EE}}$ overhead to the third part, leading to the aforementioned claim on the overhead to training time. See Figures 3(a) and (b) for a visualization.

[3]For simplicity, we do not consider other flexible ways of dividing an LLM into pipeline stages, as dividing Transformer layers evenly remains one of the most practical and robust options in practice. We also do not consider the case where the input/output embedding matrix occupies a separate pipeline stage, although our techniques for training an early-exit LLM can be generalized to this case in principle.

A.2. Further performance optimizations

Reducing activation memory overhead. So far, we have analyzed training time and memory usage by model parameters. Another major component of memory usage, namely the memory for gradients and optimizer states, can be bounded by the memory for model parameters multiplied by a universal constant. One remaining issue that we have not addressed is the *activation memory* overhead due to early-exit computation. Most notably, the early-exit logits for one microbatch have size $s \times b \times V$, where s is the maximum sequence length, b is the microbatch size, and V is the vocabulary size. If the i-th stage has one early exit, then a vanilla implementation of training with early exits using the 1F1B pipeline schedule (as shown in Figure 3(b)) will cause a significant memory overhead of size $s \times b \times V \times (P - i + 1)$, where $P - i + 1$ is the number of in-flight microbatches (Korthikanti et al., 2022) for Stage i.

Our solution for resolving this issue is simple: deferring the forward computation of each early-exit layer for each microbatch from the forward step to the corresponding backward step. Note that this is feasible, because forward computation of the next stage only requires as input the hidden states returned by the current stage, while the results of early-exit forward computation are optional. By adjusting the order of computation in this way, it is guaranteed that the early-exit logits for each microbatch are generated, used, and discarded immediately, within the same backward step; consequently, the activation memory overhead due to early-exit logits is reduced from $s \times b \times V \times (P - i + 1)$ to $s \times b \times V$. As long as this amount of memory usage is less than the activation memory of *all* Transformer layers within one stage for one microbatch (and no early exit is added to the first stage), the peak memory across all pipeline stages will stay unchanged, since the first stage remains the bottleneck of memory usage.

Remark A.2. One can check that, with the above adjustment, our analysis in Section A.1 about the training time overhead caused by early exits remains valid after minor modifications. More specifically, the time overhead of one training iteration is still $k \times (f_{\mathsf{EE}} + b_{\mathsf{EE}})$; the only difference is that this whole overhead comes from the backward steps of the last microbatch, i.e. the third part of the critical path in Figure 3(c). One can further reduce this overhead to $k \times b_{\mathsf{EE}}$ by moving the forward pass of the early-exit layer on each stage for each cool-down microbatch to the explicit bubble in front of the corresponding backward step (i.e. before communication with the next stage). With that said, our implementation does not include this modification, as it brings limited gains at the cost of complicating our codebase.

Some rules of thumb. Below are some tips for maximizing training efficiency.

- If possible, add early exits to the middle stages rather than to the first or last one. For example, adding one early exit to the end of the first stage leads to the same model architecture as adding to the beginning of the second stage, but the latter has higher training efficiency due to more balanced load across stages.
- Avoid adding too many early exits to the LLM. Despite higher flexibility during inference, the gain of adding many early exits (e.g. one per layer) might be marginal, and comes at the cost of excessive overhead for training and inference, which is especially the case for LLMs due to large vocabulary sizes. Similar observations and advice have been made recently by the authors of (Bae et al., 2023) as well.
- If there are multiple exits within the same pipeline stage, one might use the same output embedding matrix for all exits; similarly, if early exits are added to the first/last stage, one might reuse the original input/output embedding matrix for early exits. These choices reduce the memory usage by model parameters, at the cost of lower expressivity of early exits.

Remark A.3. Recall from Section 2 that, with `EE-LLM`, users can choose more expressive and powerful early-exit layers beyond the minimalistic structure. Similarly, more than one early exit can be added to each pipeline stage, which provides more flexible choices of exits during inference. These benefits, of course, come at the cost of a higher overhead for training, and potentially for inference as well[4]; with `EE-LLM`, users can conveniently choose the most suitable configurations for their own use cases. A formal analysis of training efficiency in these general cases will soon be presented in Appendix A.3.

Numerical examples. We complement previous analytical study with a few numerical examples. Figure 9 illustrates load imbalance in the original 1F1B pipeline schedule for a standard 7B GPT Transformer, as well as the impacts of adding one minimalistic early-exit layer to each middle stage (with all performance optimizations applied). Table 1 takes a close look at

[4]There is no clear answer to whether additional modules at early-exit layers will improve or hurt the overall inference speed. There is certainly a higher overhead for the computation of each early-exit layer; on the other hand, higher flexibility and adaptivity of the early exits can potentially enable them to produce better outputs and get selected more often during inference, leading to overall faster generation of a complete sequence. For similar reasons, there is no clear positive or negative correlation between the number of early exits and the overall speed of generating a sequence.

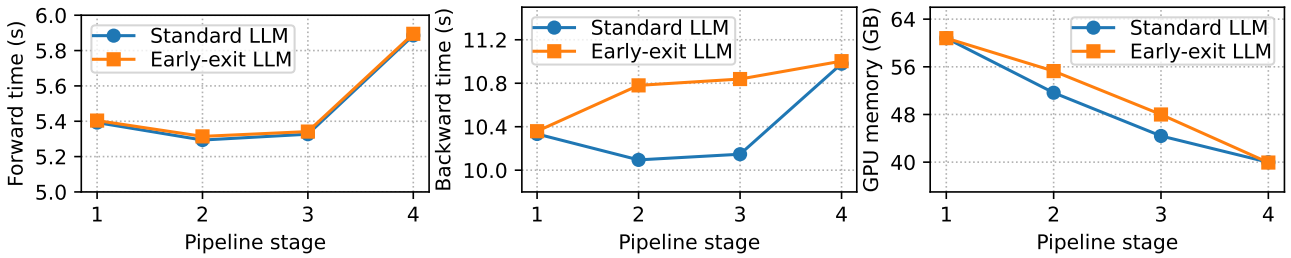


Figure 9: The forward time, backward time, and peak GPU memory of each pipeline stage for a standard 7B GPT Transformer, as well as its early-exit version that has one minimalistic early-exit layer (without layer normalization) added to each middle stage. Degrees of pipeline, tensor and data parallelism are 4, 1, and 1, respectively; the microbatch size is 2, the global batch size is 128, and the sequence length is 2048. Note that the forward computation of early-exit layers has been deferred to the backward steps, hence not included in "forward time" of the first plot, but in "backward time" of the second plot.

Table 1: Training efficiency and impacts of performance optimizations, with the same setting as in Figure 9. For the "Early-exit" row, the early exit at 1/4 depth is added to the end of Stage 1, and the exit at 1/2 depth is added to the end of Stage 2. The last three rows are annotated with the performance optimization(s) adopted, where Optimization 1 stands for deferring forward computation of early-exit layers to backward steps, and Optimization 2 stands for moving every early exit from the end of some pipeline stage to the beginning of the next stage, in order to achieve more balanced load across stages. Stage 1 is the bottleneck of peak memory in most cases, except for the numbers marked by *, for which Stage 2 is the bottleneck.

Setup	1.3B		7B	
	Time per iteration (s)	Peak memory (GB)	Time per iteration (s)	Peak memory (GB)
Standard	5.23	19.85	17.75	62.27
Early-exit	5.31	24.05	17.93	67.42
Early-exit (1)	5.29	22.56	17.91	65.79
Early-exit (2)	5.28	20.23 *	17.81	62.27
Early-exit (1&2)	5.24	19.85	17.79	62.27

the impacts of each performance optimization; unsurprisingly, the best training efficiency is achieved with all the proposed optimizations applied.

A.3. Theoretical analysis for general cases

In the following, we derive formulas for the training time per iteration and peak GPU memory usage across pipeline stages under general configurations of early exits, in terms of the quantities listed in Table 2. Some of these quantities, especially $\{f_\circ, b_\circ, m_\circ, m_\circ^\dagger\}$, can be further calculated analytically with lower-level quantities such as sequence length, microbatch size, hidden size, vocabulary size and others (Rajbhandari et al., 2019; Narayanan et al., 2021b; Korthikanti et al., 2022), or estimated empirically by profiling in practice.

It is assumed that all pipeline stages have the same number of Transformer layers on the backbone, and all early-exit layers follow the same structure. We also assume that input and output embedding matrices are untied, although it is not hard to extend our analysis to more general cases. For simplicity, we ignore the point-to-point communication latency between pipeline stages in our analysis, since it typically takes a minor proportion of the overall training time, and can often be overlapped with computation.

A.3.1. Training time per iteration

Let us analyze the training time per iteration step by step.

Step 1: simplified analysis without early exits. We start by recalling the simplified analysis in prior works for the standard 1F1B pipeline schedule (Narayanan et al., 2021b), which assumes that all stages have the same forward time f and backward time b for one microbatch. Recall from Figure 3 that the critical path of one training iteration consists of three

Table 2: A list of notation for theoretical analysis. The following abbreviations are used: IN — input processing layer; EE — early-exit layer; FE — final-exit layer; BB — Transformer backbone on one pipeline stage.

Notation	Definition
P	Number of pipeline stages
N_i	Number of early exits in Stage $i \in [P]$
M	Number of microbatches for one training iteration
$f_{\mathsf{o}}, b_{\mathsf{o}}$	Forward and backward time of one $\mathsf{o} \in \{\mathsf{IN}, \mathsf{EE}, \mathsf{FE}, \mathsf{BB}\}$ for one microbatch
m_{o}	Memory usage for storing the model parameters of one $\mathsf{o} \in \{\mathsf{IN}, \mathsf{EE}, \mathsf{FE}, \mathsf{BB}\}$
$m_{\mathsf{o}}^{\dagger}$	Activation memory of one $\mathsf{o} \in \{\mathsf{IN}, \mathsf{EE}, \mathsf{FE}, \mathsf{BB}\}$ for one microbatch
$\mathbb{1}(\cdot)$	The indicator function, $\mathbb{1}(\mathcal{E}) = 1$ if the event $\mathcal{E}$ holds true, and 0 otherwise

parts:

- Part 1: forward steps of the first microbatch on all stages except the last one, which takes time $(P-1) \times f$;
- Part 2: the steady 1F1B phase with all M microbatches on the last stage, which takes time $M \times (f+b)$;
- Part 3: backward steps of the last microbatch on all stages except the last one, which takes time $(P-1) \times b$.

Taking the sum of these three parts, we have

$$\text{time per iteration} = (P-1) \times f + M \times (f+b) + (P-1) \times b = \underbrace{(P-1) \times (f+b)}_{\text{Parts 1 and 3}} + \underbrace{M \times (f+b)}_{\text{Part 2}}.$$

Step 2: fine-grained analysis without early exits. We provide a more fine-grained analysis, by considering the computation related to the input processing layer (IN) and final-exit layer (FE) separately from the Transformer backbone (BB). It is reasonable to assume that $f_{\mathsf{IN}} < f_{\mathsf{FE}}$ and $b_{\mathsf{IN}} < b_{\mathsf{FE}}$, which implies that the last stage is the bottleneck of forward and backward time. In this setting, Parts 1 and 3 of the critical path takes time $f_{\mathsf{IN}} + (P-1) \times f_{\mathsf{BB}}$ and $b_{\mathsf{IN}} + (P-1) \times b_{\mathsf{BB}}$, respectively. Similarly, Part 2 now takes time $M \times (f_{\mathsf{BB}} + b_{\mathsf{BB}} + f_{\mathsf{FE}} + b_{\mathsf{FE}})$. Taking the sum, we arrive at

$$\text{time per iteration} = \underbrace{f_{\mathsf{IN}} + b_{\mathsf{IN}} + (P-1) \times (f_{\mathsf{BB}} + b_{\mathsf{BB}})}_{\text{Parts 1 and 3}} + \underbrace{M \times (f_{\mathsf{BB}} + b_{\mathsf{BB}} + f_{\mathsf{FE}} + b_{\mathsf{FE}})}_{\text{Part 2}}.$$

Step 3: fine-grained analysis with early exits. Finally, we are ready for our analysis in the setting with early exits (EE). First, it can be checked that early-exit layers incur a total overhead of

$$\sum_{i \in [P-1]} N_i \times (f_{\mathsf{EE}} + b_{\mathsf{EE}})$$

to Parts 1 and 3. In addition, the sum of forward and backward time of one microbatch for Stage i becomes

$$f_{\mathsf{BB}} + b_{\mathsf{BB}} + \mathbb{1}(i=1) \times (f_{\mathsf{IN}} + b_{\mathsf{IN}}) + \mathbb{1}(i=P) \times (f_{\mathsf{FE}} + b_{\mathsf{FE}}) + N_i \times (f_{\mathsf{EE}} + b_{\mathsf{EE}}).$$

Note that Part 2, namely the steady 1F1B phase on the last stage, is now *bottlenecked* by the maximum forward and backward time across all stages. Putting things together, we have the following upper bound:

$$\begin{aligned}
&\text{time per iteration} \\
&\le f_{\mathsf{IN}} + b_{\mathsf{IN}} + (P-1) \times (f_{\mathsf{BB}} + b_{\mathsf{BB}}) + \sum_{i \in [P-1]} N_i \times (f_{\mathsf{EE}} + b_{\mathsf{EE}}) \\
&\quad + M \times \max_{i \in [P]} \Big\{ f_{\mathsf{BB}} + b_{\mathsf{BB}} + \mathbb{1}(i=1) \times (f_{\mathsf{IN}} + b_{\mathsf{IN}}) + \mathbb{1}(i=P) \times (f_{\mathsf{FE}} + b_{\mathsf{FE}}) + N_i \times (f_{\mathsf{EE}} + b_{\mathsf{EE}}) \Big\}.
\end{aligned}$$

The overhead caused by early-exit layers has been highlighted. The first term corresponds to Parts 1 and 3 of the critical path, and the second term corresponds to Part 2. It is worth mentioning that the second term of the overhead might not take effect due to the maximum operator, in which case the early-exit overhead to one training iteration is independent of the number of microbatches.

A.3.2. PEAK GPU MEMORY

Consider the GPU memory for one specific pipeline stage, say Stage i. Recall that GPU memory usage during training is mainly composed of memory for model parameters, gradients, optimizer states, and intermediate activations (Rajbhandari et al., 2019; Korthikanti et al., 2022). Among them, the memory for gradients and optimizer states (for many common optimizers, such as SGD with momentum or Adam) is proportional to the number of model parameters. Therefore, we assume that

$$\text{memory}(\text{model parameters, gradients, optimizer states}) = \alpha \times \text{memory}(\text{model parameters})$$

for some universal constant $\alpha > 1$, whose concrete value depends on the choice of optimizer and numerical precisions. Under this assumption, we have

$$\text{total memory} \approx \alpha \times \text{memory}(\text{model parameters}) + \text{memory}(\text{activations}). \tag{4}$$

Now it boils down to deriving the memory for model parameters and activations. Model parameters in Stage $i \in [P]$ include the Transformer backbone (BB), the input processing layer (IN) if $i = 1$, the final-exit layer (FE) if $i = P$, and the early exits (EE). In other words, we have

$$\text{memory}(\text{model parameters}) = m_{\mathsf{BB}} + \mathbb{1}(i = 1) \times m_{\mathsf{IN}} + \mathbb{1}(i = P) \times m_{\mathsf{FE}} + N_i \times m_{\mathsf{EE}}. \tag{5}$$

The memory for intermediate activations can be similarly divided into components corresponding to BB, IN, FE and EE. Note that according to Section A.2, the peak memory usage by activations within one early-exit layer is only $m^\dagger_{\mathsf{EE}}$ regardless of the number of in-flight microbatches $P + 1 - i$. Thus one has

$$\text{memory}(\text{activations}) = (P + 1 - i) \times m^\dagger_{\mathsf{BB}} + \mathbb{1}(i = 1) \times P \times m^\dagger_{\mathsf{IN}} + \mathbb{1}(i = P) \times m^\dagger_{\mathsf{FE}} + N_i \times m^\dagger_{\mathsf{EE}}. \tag{6}$$

Combining Eq. (5) and (6) with Eq. (4), we arrive at an estimate of memory usage for Stage i:

$$\begin{aligned}\text{total memory} \approx \alpha \times &\Big(m_{\mathsf{BB}} + \mathbb{1}(i = 1) \times m_{\mathsf{IN}} + \mathbb{1}(i = P) \times m_{\mathsf{FE}} + N_i \times m_{\mathsf{EE}}\Big) \\ &+ \Big((P + 1 - i) \times m^\dagger_{\mathsf{BB}} + \mathbb{1}(i = 1) \times P \times m^\dagger_{\mathsf{IN}} + \mathbb{1}(i = P) \times m^\dagger_{\mathsf{FE}} + N_i \times m^\dagger_{\mathsf{EE}}\Big).\end{aligned}$$

The memory overhead caused by early-exit layers has been highlighted in red. Finally, taking the maximum over $i \in [P]$ gives the peak GPU memory across all pipeline stages.

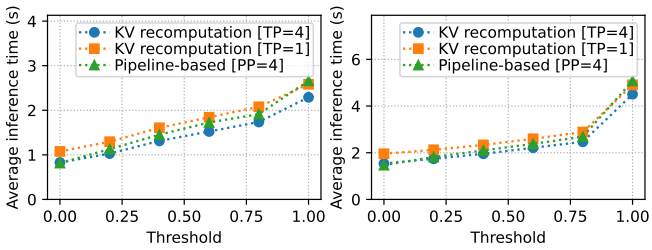
B. Additional experiments

B.1. A comparison between two inference methods

Since both pipeline-based method and KV recomputation generate the same output for the same prompt, here we only compare the inference latency of both methods. Given that the pipeline-based method uses 4 GPUs for a pipeline parallelism (PP) degree of 4, we allow KV recomputation to use a tensor parallelism (TP) degree of 1 or 4 for a fair comparison. The empirical results on XSUM and CNN/DailyMail tasks are shown in Figure 10. We first notice that the pipeline-based approach outperforms KV recomputation with TP $= 1$ for confidence thresholds smaller than 1 (i.e. when early exiting actually happens), despite the point-to-point communication overhead. Moreover, KV recomputation with TP $= 4$ is faster than the pipeline-based approach; it is worth noting, however, that the small speedup from TP $= 1$ to TP $= 4$ for KV recomputation is only possible with high-end hardware like A100 GPUs connected by high-bandwidth communication. In many use cases of LLM inference, such hardware is not available, and pipeline parallelism is the only option for partitioning a model that is too large to fit into the memory of one single device. In sum, the proposed pipeline-based approach is a more practical option for LLM inference with model partitioning in a wide range of scenarios.

B.2. Examples of generated texts

Example texts generated by our early-exit models are presented in Tables 3 and 4.



(a) XSUM (b) CNN/DailyMail

Figure 10: Inference latencies of the pipeline-based method and KV recomputation for our 7B early-exit model.

Table 3: Sequences generated by our 7B early-exit model for the same prompt, with varying confidence thresholds and inference latencies. Differences from the result of full-model inference are highlighted in red.

Prompt	Artificial General Intelligence is
Full model (Time: **1.42** s)	a branch of computer science that studies the development of computer systems that can perform tasks that normally require human intelligence. The term ”artificial intelligence” was coined by John McCarthy in 1956.
Threshold $= 0.8$ (Time: **1.17** s)	a branch of computer science that studies the development of computer systems that can perform tasks that normally require human intelligence. The term ”artificial intelligence” was coined by John McCarthy in 1956.
Threshold $= 0.4$ (Time: **0.89** s)	a branch of computer science that studies the development of computer systems that can perform tasks that normally require human intelligence. The term artificial general intelligence was coined by John McCarthy in 1965.
Threshold $= 0.2$ (Time: **0.58** s)	a branch of computer science that deals with the development of computer programs that can be used to perform tasks such as problem solving, decision making, and learning. Artificial General Intelligence is a branch of computer science that

Table 4: The predicted token and confidence at each exit of our 32-layer 7B early-exit model for generating each token within one sequence. The prompt is “The capital of China is”, and the sequence generated by full-model inference is “Beijing. \n The capital of the United States is Washington, D.C.”. Tokens that are predicted with confidence higher than 0.8 are highlighted; interestingly, all exits make the same prediction for each of these tokens.

Layer 8		Layer 16		Final layer	
located	(0.116)	Be	(0.202)	Be	(0.609)
ij	(0.992)	ij	(0.995)	ij	(0.996)
ing	(0.999)	ing	(0.999)	ing	(0.999)
,	(0.370)	.	(0.535)	.	(0.503)
\n	(0.300)	\n	(0.310)	\n	(0.227)
The	(0.126)	The	(0.150)	The	(0.158)
capital	(0.074)	capital	(0.125)	capital	(0.102)
of	(0.757)	of	(0.832)	of	(0.860)
China	(0.523)	the	(0.094)	the	(0.115)
country	(0.146)	United	(0.438)	United	(0.436)
States	(0.880)	States	(0.816)	States	(0.858)
is	(0.733)	is	(0.617)	is	(0.687)
Washington	(0.877)	Washington	(0.992)	Washington	(0.989)
,	(0.443)	,	(0.394)	,	(0.462)
D	(0.571)	D	(0.728)	D	(0.686)
.	(0.999)	.	(0.997)	.	(0.997)
C	(0.969)	C	(0.967)	C	(0.934)
.	(0.830)	.	(0.873)	.	(0.904)

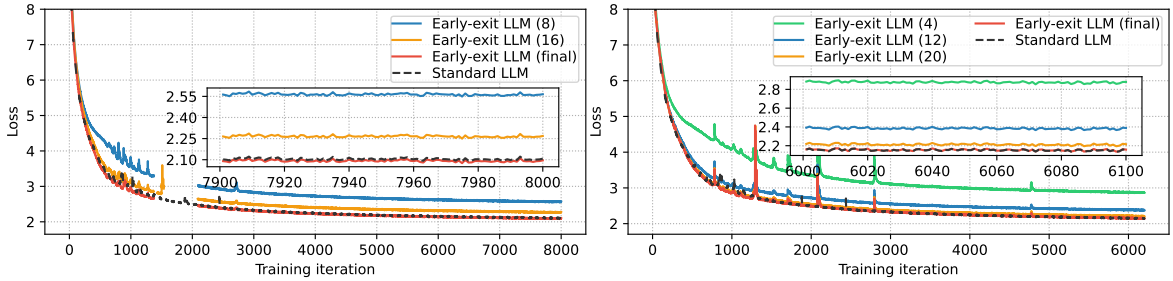


Figure 11: Convergence of early-exit/final-exit training losses for two additional models with different configurations. Each curve is annotated with the index of the Transformer layer that the corresponding exit is connected to. The missing parts in the curves of the first figure were caused by technical issues with wandb (https://wandb.ai) when the experiments were running.

B.3. Training with different early-exit configurations

As mentioned in Section 2, EE-LLM supports a wide variety of early-exit configurations, such as numbers, locations, structures and training loss weights of early exits. Here, we demonstrate the training of two additional 7B early-exit models. The first model has the same configurations as those of the 7B early-exit model considered in Section 5.1, except that each early-exit layer contains layer normalization and an MLP, on top of the output embedding matrix. The second model has three early exits located at Layer 4, 12 and 20, respectively, all containing layer normalization, with early-exit loss weights set to 0.1, 0.2 and 0.3. In addition, all embedding matrices are tied. Our empirical results, as shown in Figure 11, confirm the convergence of training losses for both models. It would be interesting future work to compare more thoroughly different early-exit configurations for both training and inference.

C. Advanced features

This section presents more details about the advanced features introduced in Section 3.3 for training early-exit LLMs.

C.1. Non-constant early-exit loss weights

The weights of early-exit losses in the training objective of Eq. (1) can be changing rather than constant during the training process, just like the learning rate or other hyperparameters. Allowing the weights to change can offer more fine-grained control of how gradients from multiple losses jointly impact the backbone and early/final output layers of the model.

One concrete option that we offer in EE-LLM is *warm-up*. With this option, early-exit loss weights start at small values, and gradually increase with training iterations until reaching the pre-specified maximum values. This approach has been adopted in prior works (Kaya et al., 2019). The idea is to encourage the deep neural network to primarily optimize for the full-model output quality from the beginning of the training process, while the skill of early exiting is gradually acquired with minor or no negative impact on the final outputs of the full model.

Another option is *cool-down*, which does the opposite and decreases early-exit loss weights during the training process. This option is inspired by prior works (Lee et al., 2014; Szegedy et al., 2015) that leverage early-exit losses for the purpose of regularizing the training process of deep neural networks. Such "deep supervision" provided by early-exit losses can stabilize and accelerate the convergence of training, and potentially improve the intermediate features learned by the neural network. As early-exit loss weights gradually decay, the regularization gets weaker, so that the network becomes more focused on its primary objective, namely the final-output quality.

C.2. Filling explicit bubbles

To further leverage the explicit bubbles within the 1F1B pipeline schedule, i.e. the light gray areas in Figure 4, we design and implement a novel approach of filling them with *partial* forward/backward computation of additional microbatches. See Figure 4 for a visualization. This is primarily inspired by the idea from (Osawa et al., 2023): instead of designing a new, sophisticated schedule that has a lower bubble ratio, one may seek to fill the bubbles of an existing schedule with useful computation, which leads to better resource utilization and faster training. Note that our method changes the optimization

semantics. From the perspective of stochastic optimization, we can prove formally that with such additional computation and under certain conditions, one obtains an *unbiased* gradient estimate with *reduced variance* for the original training objective. The remaining of this section is dedicated to details of the methodology and analysis.

C.2.1. METHODOLOGY

Our approach is explained below. For notational simplicity, let us call the explicit bubbles between the warm-up and steady phases as Part 1, and the bubbles during the cool-down phase as Part 2. For each part, we fill them with some computation for K additional microbatches. More concretely, for Part 1, the i-th inserted microbatch (where $i \in [K]$) goes through forward computation of the first $K+1-i$ pipeline stages, followed by backward computation of all visited early-exit losses; for Part 2, each inserted microbatch goes through forward computation of all stages, followed by backward computation of the final and early-exit losses (if any) only for the last few stages. In this way, each training iteration can process more data without any time overhead, as long as the number of inserted microbatches and the number of stages for partial forward/backward computation are chosen appropriately.

How many microbatches can be inserted? The maximum number of microbatches that can be inserted into Part 1 or 2 of the explicit bubbles, without increasing training time per iteration, is $\lfloor (p-1)b/(f+b) \rfloor = \lfloor (p-1)/(f/b+1) \rfloor$, where f/b is (an estimate of) the ratio between forward and backward time. To see this for Part 1 (resp. 2), one simply needs to notice that the first (resp. last) pipeline stage has a bubble size $b(p-1)$, while the total forward and backward time for each microbatch is $f+b$. Dividing the first value by the second one concludes the proof.

Details about Part 1. With K inserted microbatches, the i-th microbatch is supposed to go through the forward pass of the first $K+1-i$ stages. However, if there is no early exit on Stage $K+1-i$, then this microbatch only needs to go through the forward pass up to the last stage (among the first $K+1-i$ stages) that has at least one early exit.

Details about Part 2. We can calculate the maximum number of backward stages for each inserted microbatch, while ensuring that no overhead to training time per iteration occurs. Notice that for the i-th microbatch, the remaining bubble size after its forward computation at the last stage is $(p-1)b - f - (i-1)(f+b) = pb - i(f+b)$. Dividing this value by b gives the number of backward stages $\lfloor (pb - i(f+b))/b \rfloor = \lfloor p - i(f/b+1) \rfloor$.

Some remarks. (1) There are some limitations to the proposed approach. For example, it requires that there is no tied/shared model parameter across pipeline stages; otherwise, the gradients from partial forward/backward passes might (in theory) be harmful rather than beneficial for training the model. Another concern is that inserting microbatches into Part 1 of the explicit bubbles can cause additional overhead of activation memory in early stages. (2) While Part 1 of this approach requires the existence of early exits, Part 2 is actually applicable to training standard models without early exits as well. (3) One might raise concerns about the inefficiency of data usage, as some microbatches only undergo partial forward/backward passes. In general, this is not an issue for LLM pre-training, since training data is usually abundant, and the complete training process might not even go through one epoch of the data.

A different perspective. The actual implementation of this method in `EE-LLM` takes a different perspective. Instead of inserting additional microbatches into the original schedule, we keep the number of microbatches per training iteration unchanged, and do the following: (1) we replace the full forward/backward computation of a few microbatches with partial forward/backward computation, which can be placed in the bubble between the warm-up and steady phases; (2) we truncate the backward computation of the last few microbatches. These two steps correspond exactly to Parts 1 and 2 introduced earlier, and the visualization in Figure 4(b) remains valid. Such an implementation reduces the training time per iteration, at the cost of lower data utilization.

C.2.2. ANALYSIS

We argue that such partial forward/backward computation offers useful gradient information for training, under the assumption that there is no tied model parameter across pipeline stages. For a theoretical justification, let us take the perspective of stochastic optimization, and assume that each microbatch is sampled independently from some data distribution.

Claim C.1 (Informal). Under the above assumptions, the accumulated gradient of one training iteration, with extra updates from additional microbatches inserted into Part 2, remains (after some appropriate entrywise scaling) an *unbiased* estimate, but with *reduced variance*, of the gradient for the targeted population risk. A similar claim holds true for Part 1 as well,

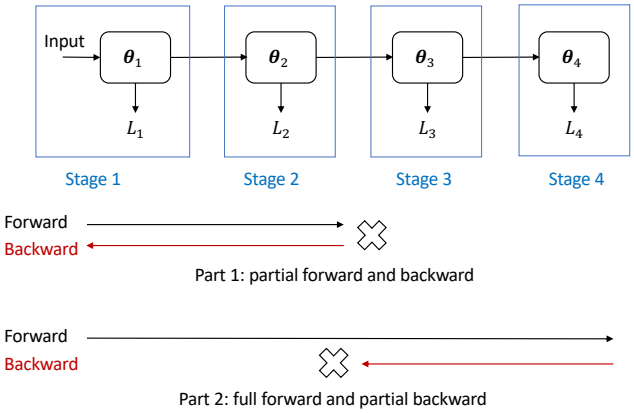


Figure 12: An illustration of the partial forward/backward passes when filling pipeline bubbles with additional microbatches.

except for certain extreme (and unlikely) situations where gradients of different early/final losses have a strong negative correlation.

In the remaining of this section, we provide an informal proof for this claim, in a concrete example with four pipeline stages and one additional microbatch inserted into either Part 1 or Part 2; a visualization can be found in Figure 12. Recall that the training objective is defined as $\mathcal{L} = \sum_{i\in[4]} \mathcal{L}_i$, where each $\mathcal{L}_i$ is a weighted sum of all losses on Stage i. We denote the model parameters as $\boldsymbol{\theta} = [\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \boldsymbol{\theta}_4]$, with correspondence to how the model is partitioned into four pipeline stages. One key observation that will be useful is that $\partial \mathcal{L}_i / \partial \boldsymbol{\theta}_j = \mathbf{0}$ for any $1 \le i < j \le 4$, due to the sequential nature of the model.

Analysis of Part 2. We start with an analysis for Part 2, which is slightly simpler. Suppose that an additional microbatch goes through the full forward pass, followed by a partial backward pass covering only the last two stages, namely $\boldsymbol{\theta}_4$ and $\boldsymbol{\theta}_3$. Then, the gradient from this microbatch is

$$\text{gradient} = \left[\mathbf{0}, \mathbf{0}, \frac{\partial(\mathcal{L}_3 + \mathcal{L}_4)}{\partial \boldsymbol{\theta}_3}, \frac{\partial(\mathcal{L}_3 + \mathcal{L}_4)}{\partial \boldsymbol{\theta}_4}\right] = \left[\mathbf{0}, \mathbf{0}, \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_3}, \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_4}\right];$$

here, the first equality is due to Proposition 3.1 and the partial backward pass, while the second equality follows from the observation that $\partial \mathcal{L}_i / \partial \boldsymbol{\theta}_j = \mathbf{0}$ for any $i < j$.

Now, suppose that a total of B microbatches was originally used for one training iteration, and we let $\mathcal{L}^{(k)}$ denote the loss of the k-th microbatch. Then, with the additional $(B+1)$-th microbatch inserted into Part 2 of the explicit bubbles, the accumulated gradient of one training iteration (normalized by $1/B$) becomes:

$$\begin{aligned}\text{accumulated gradient} &= \frac{1}{B}\left(\sum_{k\in[B]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}} + \left[\mathbf{0}, \mathbf{0}, \frac{\partial \mathcal{L}^{(B+1)}}{\partial \boldsymbol{\theta}_3}, \frac{\partial \mathcal{L}^{(B+1)}}{\partial \boldsymbol{\theta}_4}\right]\right) \\ &= \left[\frac{1}{B}\sum_{k\in[B]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_1}, \frac{1}{B}\sum_{k\in[B]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_2}, \frac{1}{B}\sum_{k\in[B+1]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_3}, \frac{1}{B}\sum_{k\in[B+1]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_4}\right]\end{aligned}$$

Taking the expectation over the data distribution, we realize that the additional microbatch essentially scales up gradients for $\boldsymbol{\theta}_3$ and $\boldsymbol{\theta}_4$ by a factor of $(B+1)/B$. Or, with a simple entrywise scaling on the gradients for $\boldsymbol{\theta}_3$ and $\boldsymbol{\theta}_4$ by a factor of $B/(B+1)$, we recover an unbiased estimate of the gradient for the targeted population risk, with reduced variance:

$$\text{accumulated gradient} = \left[\frac{1}{B}\sum_{k\in[B]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_1}, \frac{1}{B}\sum_{k\in[B]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_2}, \frac{1}{B+1}\sum_{k\in[B+1]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_3}, \frac{1}{B+1}\sum_{k\in[B+1]} \frac{\partial \mathcal{L}^{(k)}}{\partial \boldsymbol{\theta}_4}\right].$$

Analysis of Part 1. Suppose that an additional microbatch goes through the forward and backward computation only for the first two stages, corresponding to model parameters $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$. Then, the gradient from this microbatch is

$$\text{gradient} = \left[\frac{\partial(\mathcal{L}_1+\mathcal{L}_2)}{\partial\boldsymbol{\theta}_1}, \frac{\partial(\mathcal{L}_1+\mathcal{L}_2)}{\partial\boldsymbol{\theta}_2}, \mathbf{0}, \mathbf{0}\right] = \frac{\partial(\mathcal{L}_1+\mathcal{L}_2)}{\partial\boldsymbol{\theta}};$$

here, the first equality is due to the partial forward and backward passes, and the second equality follows again from the observation that $\partial\mathcal{L}_i/\partial\boldsymbol{\theta}_j = \mathbf{0}$ for any $i < j$.

Now, suppose that a total of B microbatches was originally used for one training iteration. We denote the loss of the k-th microbatch as $\mathcal{L}^{(k)} = \sum_{i\in[4]} \mathcal{L}_i^{(k)}$. Then, with the additional $(B+1)$-th microbatch inserted into Part 1 of the explicit bubbles, the accumulated gradient of one training iteration (normalized by $1/B$) becomes:

$$\begin{aligned}
\text{accumulated gradient} &= \frac{1}{B}\left(\sum_{k\in[B]} \frac{\partial\mathcal{L}^{(k)}}{\partial\boldsymbol{\theta}} + \frac{\partial(\mathcal{L}_1^{(B+1)}+\mathcal{L}_2^{(B+1)})}{\partial\boldsymbol{\theta}}\right) \\
&= \frac{1}{B}\left(\sum_{k\in[B]} \frac{\partial(\mathcal{L}_1^{(k)}+\mathcal{L}_2^{(k)}+\mathcal{L}_3^{(k)}+\mathcal{L}_4^{(k)})}{\partial\boldsymbol{\theta}} + \frac{\partial(\mathcal{L}_1^{(B+1)}+\mathcal{L}_2^{(B+1)})}{\partial\boldsymbol{\theta}}\right) \\
&= \frac{1}{B}\left(\sum_{k\in[B+1]} \frac{\partial(\mathcal{L}_1^{(k)}+\mathcal{L}_2^{(k)})}{\partial\boldsymbol{\theta}} + \sum_{k\in[B]} \frac{\partial(\mathcal{L}_3^{(k)}+\mathcal{L}_4^{(k)})}{\partial\boldsymbol{\theta}}\right).
\end{aligned}$$

Taking the expectation over the data distribution, we see that the additional microbatch essentially scales up the weights of $\mathcal{L}_1$ and $\mathcal{L}_2$ in the gradient by a factor of $(B+1)/B$. Or, if the weights of $\mathcal{L}_1$ and $\mathcal{L}_2$ are manually scaled by a factor of $B/(B+1)$ during training, then we recover an unbiased gradient estimate for the original risk $\mathcal{L} = \sum_{i\in[4]} \mathcal{L}_i$:

$$\text{accumulated gradient} = \frac{1}{B+1}\sum_{k\in[B+1]} \frac{\partial(\mathcal{L}_1^{(k)}+\mathcal{L}_2^{(k)})}{\partial\boldsymbol{\theta}} + \frac{1}{B}\sum_{k\in[B]} \frac{\partial(\mathcal{L}_3^{(k)}+\mathcal{L}_4^{(k)})}{\partial\boldsymbol{\theta}}$$

We claim that this leads to reduced gradient variance, *unless* gradients of early and later losses have a strong negative correlation. In the formal analysis of variance below, we consider scalars for notational simplicity, though it can be easily generalized to vectors or tensors by replacing scalar multiplication with inner product.

Proposition C.2. *Consider two distributions $\mathcal{A}$ and $\mathcal{B}$ with means $a^\star$ and $b^\star$, respectively. Let $a_1, \ldots, a_N, a_{N+1}$ be independent and identically distributed (i.i.d.) samples of $\mathcal{A}$, and $b_1, \ldots, b_N$ be i.i.d. samples of $\mathcal{B}$, but for each i, a_i can be correlated with b_i. Consider two estimates of $a^\star + b^\star$, defined as follows:*

$$\widehat{e} \coloneqq \widehat{a}_N + \widehat{b}_N, \quad \widehat{e}_+ \coloneqq \widehat{a}_{N+1} + \widehat{b}_N, \quad \textit{where}$$

$$\widehat{a}_k \coloneqq \frac{1}{k}\sum_{i\in[k]} a_i, \quad \widehat{b}_k \coloneqq \frac{1}{k}\sum_{i\in[k]} b_i, \quad k \in \{N, N+1\}.$$

Then it holds that

$$\mathbb{E}[\widehat{e}] = \mathbb{E}[\widehat{e}_+] = a^\star + b^\star, \quad \mathit{var}(\widehat{e}) - \mathit{var}(\widehat{e}_+) = \frac{1}{N(N+1)}\mathit{var}(a_1) + \frac{2}{N(N+1)}\mathit{cov}(a_1, b_1).$$

In this proposition, $\widehat{a}_N, \widehat{a}_{N+1}$ and $\widehat{b}_N$ correspond to $\frac{1}{B}\sum_{k\in[B]} \frac{\partial(\mathcal{L}_1^{(k)}+\mathcal{L}_2^{(k)})}{\partial\boldsymbol{\theta}}$, $\frac{1}{B+1}\sum_{k\in[B+1]} \frac{\partial(\mathcal{L}_1^{(k)}+\mathcal{L}_2^{(k)})}{\partial\boldsymbol{\theta}}$ and $\frac{1}{B}\sum_{k\in[B]} \frac{\partial(\mathcal{L}_3^{(k)}+\mathcal{L}_4^{(k)})}{\partial\boldsymbol{\theta}}$ in our previous analysis of accumulated gradients, respectively. Hence our claim on gradient variance follows immediately from the conclusion of this proposition.

Proof of Proposition C.2. First, unbiasedness is obvious. As for variance, we have the following elementary calculation:

$$\begin{aligned}
\mathrm{var}(\widehat{e}) &= \mathbb{E}\big[(\widehat{a}_N + \widehat{b}_N - a^\star - b^\star)^2\big] = \mathbb{E}\big[(\widehat{a}_N - a^\star)^2\big] + \mathbb{E}\big[(\widehat{b}_N - b^\star)^2\big] + 2\mathbb{E}\big[(\widehat{a}_N - a^\star)(\widehat{b}_N - b^\star)\big], \\
\mathrm{var}(\widehat{e}_+) &= \mathbb{E}\big[(\widehat{a}_{N+1} + \widehat{b}_N - a^\star - b^\star)^2\big] = \mathbb{E}\big[(\widehat{a}_{N+1} - a^\star)^2\big] + \mathbb{E}\big[(\widehat{b}_N - b^\star)^2\big] + 2\mathbb{E}\big[(\widehat{a}_{N+1} - a^\star)(\widehat{b}_N - b^\star)\big].
\end{aligned}$$

Moreover,

$$\mathbb{E}\big[(\widehat{a}_N - a^\star)^2\big] = \frac{1}{N}\mathrm{var}(a_1), \quad \mathbb{E}\big[(\widehat{a}_{N+1} - a^\star)^2\big] = \frac{1}{N+1}\mathrm{var}(a_1),$$

and

$$\mathbb{E}\big[(\widehat{a}_N - a^\star)(\widehat{b}_N - b^\star)\big] = \frac{1}{N^2}\sum_{i\in[N]}\mathbb{E}\big[(a_i - a^\star)(b_i - b^\star)\big] = \frac{1}{N}\mathrm{cov}(a_1, b_1),$$
$$\mathbb{E}\big[(\widehat{a}_{N+1} - a^\star)(\widehat{b}_N - b^\star)\big] = \frac{1}{N(N+1)}\sum_{i\in[N]}\mathbb{E}\big[(a_i - a^\star)(b_i - b^\star)\big] = \frac{1}{N+1}\mathrm{cov}(a_1, b_1).$$

Putting things together,

$$\begin{aligned}\mathrm{var}(\widehat{e}) - \mathrm{var}(\widehat{e}_+) &= \mathbb{E}\big[(\widehat{a}_N - a^\star)^2\big] - \mathbb{E}\big[(\widehat{a}_{N+1} - a^\star)^2\big] \\ &\quad + 2\Big(\mathbb{E}\big[(\widehat{a}_N - a^\star)(\widehat{b}_N - b^\star)\big] - \mathbb{E}\big[(\widehat{a}_{N+1} - a^\star)(\widehat{b}_N - b^\star)\big]\Big) \\ &= \frac{1}{N(N+1)}\mathrm{var}(a_1) + \frac{2}{N(N+1)}\mathrm{cov}(a_1, b_1),\end{aligned}$$

which concludes our proof. □

D. Implementations

The implementation of `EE-LLM` is based on Megatron-LM (Narayanan et al., 2021b), primarily extending Megatron-LM's model architectures, pipeline scheduling, and inference service to support the training and inference of early-exit LLMs. We introduce each of these aspects in more detail below.

D.1. Model architectures

We have introduced a new class of models called `EarlyExitGPTModel`, which is the early-exit counterpart of `GPTModel` in the original model library of Megatron-LM. The model is constructed with a few other classes, including `EarlyExitTransformerLayer`, `EarlyExitTransformer`, and `EarlyExitLanguageModel`. `EarlyExitTransformerLayer` is a replacement for the original `ParallelTransformerLayer` in Megatron-LM. It adds an early-exit structure on top of the standard Transformer layer, which allows it to generate outputs for both the main network backbone and the early exit; for the latter, it returns a lazy loss function during training, or tokens during inference. This module supports various customizations of the early-exit structure; besides the minimalistic structure with an output embedding matrix and an optional output normalization layer, one might add e.g. a MLP or a complete Transformer layer. These additional structures can be combined in any desired manner and can be placed before or after the backbone part of this layer. On the other hand, `EarlyExitTransformer` and `EarlyExitLanguageModel` are mainly used to propagate the early-exit outputs to the top-level model. They are capable of stopping the forward computation at the early-exit layer and returning the intermediate outputs, which facilitates accelerated inference.

D.2. Pipeline scheduling

We have adjusted the existing 1F1B schedule for early-exit LLMs, as shown in Figure 3. To fill implicit bubbles and reduce GPU memory overhead, lazy loss functions of early-exit layers are returned together with outputs of the backbone network during forward steps. These lazy functions are not actually called until their corresponding auxiliary losses (cf. Section 3.1.1) are calculated in the backward steps. For the method of filling explicit bubbles proposed in Section 3.3, we have inserted partial forward/backward computation of additional microbatches into warm-up and cool-down phases of the 1F1B schedule. The number of inserted microbatches and partial forward/backward stages can be automatically calculated through the user-specified (estimate of) ratio between backward and forward time.

D.3. Inference service

To support inference of early-exit LLMs, we have refactored the text-generation module of Megatron-LM.

For inference with pipeline parallelism, i.e. the *pipeline-based* approach proposed in Section 4, we have re-implemented the forward process. With our implementation, the first pipeline stage will wait for an exit signal from the early/final exits

of all subsequent stages after its forward computation is completed. Each subsequent stage will send an exit signal and the output token to the first stage, if there is an exit within the stage that satisfies the exit condition. Upon receiving the signal and generated token, the first stage will immediately start the forward pass for generating the next token. With this implementation, regardless of the early-exit layers' positions in subsequent stages, the inference service can immediately generate a token whenever early exiting happens on some stage, without waiting for the completion of the entire stage (except for the first stage).

For inference without pipeline parallelism, we have implemented a mechanism of *KV recomputation*, which is a variant of synchronized parallel decoding proposed recently in (Bae et al., 2023). In this approach, we maintain a list of the most recent tokens that have missing KV caches in deep layers due to early exiting. During each forward pass, we include these early-exit tokens in the current forward pass, which allows for direct recomputation of the KV caches for these tokens and thus avoids the issue of missing KV caches. Acceleration of sequence generation is still achieved, thanks to the batching effects of GPU computation. To avoid the endless accumulation of early-exit tokens, we enforce a full-model forward pass whenever the number of early-exit tokens reaches a pre-specified value.

E. Preliminaries

Transformers. The Transformer architecture (Vaswani et al., 2017; Tay et al., 2022) has been playing a dominant role in natural language processing (NLP) and large language models (LLMs) (Bommasani et al., 2021; Zhao et al., 2023a). It is typically composed of an input embedding layer, a stack of Transformer layers, and finally an output layer. Each Transformer layer consists of cross-attention and/or self-attention modules (Bahdanau et al., 2015; Kim et al., 2017; Parikh et al., 2016), a multi-layer perceptron (MLP), and layer normalization (LayerNorm (Ba et al., 2016) or RMSNorm (Zhang & Sennrich, 2019)). Transformers can be categorized into three types: encoder-only, encoder-decoder, and decoder-only. For the latter two, there is an output embedding matrix in the output layer, which transforms hidden states into logits on a (typically large) vocabulary that can be used for generating tokens. An LLM can be learned by unsupervised pre-training, e.g. minimizing the negative log-likelihood of next-token prediction on a large corpus (Radford et al., 2018; 2019). In this work, we focus on the decoder-only generative pre-training (GPT) Transformer architecture (Radford et al., 2018; 2019), though many of our ideas are widely applicable to other Transformer architectures or generic deep neural networks.

Early-exit LLMs. An early-exit neural network can be obtained by adding to a standard neural network some early-exit layers that turn intermediate hidden states into early outputs (Xin et al., 2020; Schwartz et al., 2020). During inference for a given input, the model starts a forward pass and decides (at each early exit) whether to return an output or continue forwarding via certain rules, e.g. to return an output whenever the confidence of prediction is above a pre-defined threshold (Schwartz et al., 2020; Schuster et al., 2022).

The standard way of training an early-exit model is to minimize a weighted sum of early-exit and final-exit training losses (Schwartz et al., 2020; Schuster et al., 2022). Note that early-exit layers bring additional computational overhead to training. This is especially the case for LLMs, primarily due to the large output embedding matrix of size $h \times V$ within each early-exit layer, where h is the hidden dimension and V is the vocabulary size. We call an early-exit layer *minimalistic* if it has the same structure as the final output layer of the GPT model architecture (Radford et al., 2018; 2019), which includes an output embedding matrix, plus an optional LayerNorm/RMSNorm in front of it. Additional modules can be added to early-exit layers for increased expressivity and adaptivity of early exits.

3D parallelism. 3D parallelism refers to the combination of data, tensor, sequence and pipeline parallelism, and has been implemented in state-of-the-art LLM frameworks such as Megatron-LM (Shoeybi et al., 2019; Narayanan et al., 2021b; Korthikanti et al., 2022; Smith et al., 2022), DeepSpeed (Rasley et al., 2020; Smith et al., 2022), Mesh-TensorFlow (Shazeer et al., 2018), Alpa (Zheng et al., 2022), InternLM (Team, 2023), among others. With *data parallelism*, each GPU handles the forward and backward computation for one part of the data batch, and then the results are aggregated at the end of the training iteration. When the model is too large to fit in a single GPU, model partitioning becomes necessary and can be used in conjunction with data parallelism. With *tensor (and sequence) parallelism*, each large module (e.g. a linear layer) is divided into multiple pieces that are assigned to different GPUs, so that each computational task related to it (e.g. large matrix multiplication) can be divided into smaller tasks and solved in parallel. One major limitation of tensor (and sequence) parallelism is that it requires expensive collective communication such as all-reduce operations, and thus is only viable for high-end GPUs within the same computing node, with high-bandwidth communication among them.

Pipeline parallelism (Narayanan et al., 2019; 2021a; Fan et al., 2021; Li & Hoefler, 2021), on the other hand, partitions a

deep model along the depth dimension into multiple pipeline stages. Moreover, each data batch is divided into multiple microbatches, and their forward/backward computational tasks are scheduled among those multiple pipeline stages. More specifically, each stage performs the forward computation for each microbatch and sends the resulting hidden states to another stage; later on, it performs the backward computation for the same microbatch after receiving the gradients of the training objective with respect to the sent hidden states. Pipeline parallelism only requires sparse and inexpensive point-to-point (P2P) communication between pipeline stages, which makes it applicable and oftentimes must-have in much broader scenarios when tensor (and sequence) parallelism is infeasible or insufficient, whether in GPU clusters or in decentralized settings (Yuan et al., 2022; Yang et al., 2023). The main concern with pipeline parallelism is its low utilization rate of computational resources, due to pipeline bubbles and load imbalance across pipeline stages (Narayanan et al., 2021b).

F. Related works

Early exiting. As introduced in Section 1, early exiting has been widely applied for accelerating inference of deep neural networks in the literature (Graves, 2016; Liu et al., 2020; Hou et al., 2020; Zhou et al., 2020; Elbayad et al., 2020; Schwartz et al., 2020; Xin et al., 2020; Schuster et al., 2021; Xin et al., 2021; Li et al., 2021; Hu et al., 2023; Teerapittayanon et al., 2016; Huang et al., 2018; Kaya et al., 2019; Laskaridis et al., 2021; Scardapane et al., 2020; Han et al., 2022; Xu & McAuley, 2023; Dai et al., 2023). In terms of early-exit Transformers in the NLP domain, the majority of prior works are focused on BERT (Devlin et al., 2019) or other encoder-only models for classification tasks (Liu et al., 2020; Hou et al., 2020; Zhou et al., 2020; Elbayad et al., 2020; Schwartz et al., 2020; Xin et al., 2020; Schuster et al., 2021; Xin et al., 2021; Li et al., 2021; Hu et al., 2023). Recent works have begun to study token-wise early exiting for accelerating inference of encoder-decoder or decoder-only LLMs in autoregressive sequence generation (Schuster et al., 2021; Corro et al., 2023; Bae et al., 2023; Varshney et al., 2023). While they are largely focused on designing *inference* mechanisms, the lack of support in prior works for *training* early-exit models with massive 3D parallelism inevitably poses an obstacle to truly scaling up early-exit LLMs. Our work on EE-LLM is one important step towards eliminating this obstacle, and we also contribute a novel pipeline-based inference mechanism along the way.

Early-exit inference with KV caching. Several approaches have been recently proposed to resolve the conflict between early exiting and KV caching in autoregressive generation. One approach (Elbayad et al., 2020; Li et al., 2021; Schuster et al., 2022) is to copy the hidden states of the current token at the exiting layer to all later layers, which will be used to compute the keys and values at later attention layers. Despite its efficiency, this method causes a deviation from the inference process that the model is trained to excel at, which can harm the output quality. Another solution (Corro et al., 2023) is to pre-specify the exiting layer for each token, while ensuring that KV missing in previous tokens will not hinder generation of later tokens; with this approach, the ability of token-wise adaptive selection of exits is inevitably lost. The third method (Bae et al., 2023; Tang et al., 2023) is to store the hidden states of recent tokens that were generated with early exiting, and whenever KV missing happens, run a batch forward pass with the current and recent tokens to fulfill the KV caches. Acceleration by this method relies on the batching effect of GPU computation. In comparison, our proposed pipeline-based method achieves early-exit acceleration in theoretical time complexity.

Other applications of early-exit models. Interestingly, early exiting has been used for many other purposes in the literature, besides accelerating inference. For example, early-exit losses can provide deep supervision and regularization for training deep networks with enhanced stability of convergence (Szegedy et al., 2015; Lee et al., 2014), which motivates our implementation of the cool-down option for non-constant early-exit loss weights, as explained in Appendix C.1. Other works use early exits for interpreting the intermediate features learned by each layer of a Transformer (Langedijk et al., 2023), for improving outputs of the final exit (Gera et al., 2023), or as the draft models in speculative decoding (Kim et al., 2023b). Early exiting has also found applications in long-tailed classification (Duggal et al., 2020), federated learning with heterogeneous clients (Ilhan et al., 2023; Zhong et al., 2022), token reduction (Liu et al., 2024), layer-wise training of deep networks (Scardapane et al., 2020), among others. We believe that our progress in this work can be beneficial for these various purposes.

Other methods of accelerating LLM inference. It is worth mentioning that there are other types of dynamic neural networks (Han et al., 2022; Xu & McAuley, 2023) that facilitate conditional computation and elastic inference, such as layer skipping (Bengio et al., 2013; 2015; Wang et al., 2022; Corro et al., 2023; Din et al., 2023; Zeng et al., 2023; Wang et al., 2023) and mixtures of experts (Jacobs et al., 1991; Shazeer et al., 2017; Fedus et al., 2021). Another line of work aims to accelerate LLM inference by designing new decoding methods rather than model architectures (Leviathan et al., 2022;

Santilli et al., 2023; Zhao et al., 2023b). Each of these approaches has its own pros and cons, and some of them can be complemented by early exiting. A detailed comparison between these methods is beyond the scope of the current work.