On Scheduling Early-Exit Layers for Model Pipeline in 6G-Based Edge Inference

Yuxiao Liu, Rui Han, Qinglong Zhang, Haiting Hou, Chi Harold Liu, and Lydia Y. Chen

Abstract

When running edge intelligence applications with 6G networks, model pipeline effectively reduces inference latency via parallelizing layers across multiple edge devices. Today's edge inference systems usually employ static architecture of layers in pipeline parallelism but dynamically skip part of layers in early-exit, which may significantly degrade system throughput. In this paper, we introduce DensePipe, an online layer scheduling approach that optimally allocates early-exit layers to edge devices to maximize their throughput in model pipeline. To this end, DensePipe profiles all network layers' skipping probabilities in early-exit. At run-time, DensePipe maximizes the pipeline throughput by balancing the processing of all unskipped layers among devices according to the current loads and device resource utilizations. We implement DensePipe with Transformer models and demonstrate its effectiveness against state-of-the-art pipeline methods. Comparative experiments show that DensePiple successfully finds the best devices for most of the layers and significantly improves throughput by 3.09x.

Introduction

The rise of 6G communication technology connects edge devices with fast data transmission speeds and facilitates their collaborations in edge intelligence systems. When running large deep learning models (e.g. ViT (Vision Transformer)) on resource-constrained edge devices, *model pipeline* effectively reduces their inference latencies by parallelizing network layers across multiple devices [1], [2]. At the same time, *early exit* [3] allows an inference ends immediately when the required model accuracy is met. Today's edge intelligence systems employ both techniques to improve system performance, but also introduce additional management complexity of edge devices' resources. Specifically, model pipeline techniques usually rely on a static deployment of models' layers [1], [4], [5], but early exit techniques dynamically skip part of these layers according to the characterizes of inference samples [6], thus may cause low resource utilizations for devices with large proportions of skipped layers.

Example. Figure 1(a) illustrates an example scenario where a traffic sign classification transformer of three layers is deployed on three autonomous vehicles. All vehicles are equipped with the same Raspberry Pi 4B board, and the skip probabilities of three layers are 0%, 50%, and 90%, respectively. When using both model pipeline and early exit in inference samples I_1 to I_3, we can see that device 1 has 100% resource utilization while this value is only 33% for device 3. This is because when inferences I_2 and I_3, early exit skips layer 3 on device 3 and thus causes unbalanced resource usages among three devices. Figure 1(b) supports this claim by testing the layers' skipping probabilities when applying early exit in ViT-base model and CIFAR-10 dataset. The result shows more than 60% of layers are skipped with a probability higher than 50%. Successfully applying model pipeline and early exit faces two *key challenges* in practice.

First of all, existing pipeline techniques (e.g. AP2 [4] and PipeEdge [5]) deploy each model layer to a fixed device, based on the best matching between this layer's computational cost and the device's computing capacity [1], [2]. However, such static deployment pre-determines the layers' allocated devices in inference. Using Figure 1(a)'s scenario as an example, existing pipeline techniques equally deploy one layer in one device, because three devices have the same computing capacity. However, with early exit techniques, more than 50% of inference samples skip the processing on devices 2 and 3. An ideal deployment, is to deploy the layers with low skip probabilities on more devices, and deploy the layers with high skip probabilities on fewer devices. For example, layer 1 can be deployed on all devices. As a result, devices 2 and 3 have a higher utilization because they can be used to process layer 1. The **first challenge**, therefore, is *how to understand the layers' probabilities of being skipped in early exit and construct a deployment that supports layers' dynamic allocation in pipeline inference*.

Second, during online inference, early exit techniques (e.g. Deebert [7]) may skip a different number of layers for each inference sample, and each device's load and available resource dynamically change. Such unpredictable changes make load balancing difficult to achieve. For example, given the ideal deployment described above, an inference sample may have two scheduling options: sequentially processed on devices 1,2, and 3, or sequentially processed on devices 3, 2, and 1. If device 1 has three running inferences but device 3 is idle, the second option can alleviate the resource contention on device 1 and thus

Yuxiao Liu, Rui Han (corresponding author), Qinglong Zhang, Haiting Hou, and Chi Harold Liu are with the Beijing Institute of Technology, Beijing 100081, China; Lydia Y. Chen is with the Department of Computer Science, Delft University of Technology (TU Delft), 2628 CD Delft, The Netherlands.

Digital Object Identifier: 10.1109/MNET.2024.3520555
Date of Current Version: 16 September 2025
Date of Publication: 19 December 2024

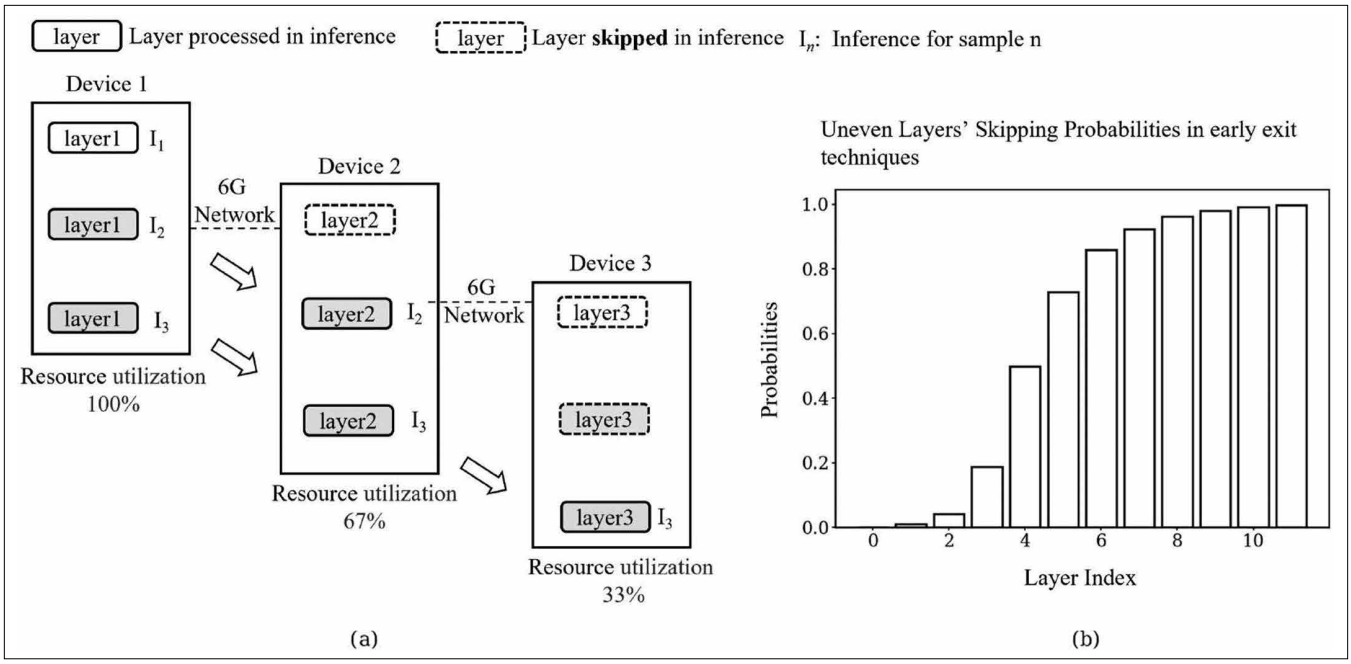

FIGURE 1. Introduction of pipeline model parallelism and early exit. a) An example scenario of applying pipeline techniques on early-exit model inference. b) Skipping probabilities of layers.

improve the throughput. This gives rise to the **second challenge** about how to *efficiently search the optimal device for each layer under dynamically changing skipping probability and system status at run-time*.

This paper proposes DensePipe, an online scheduling approach that dynamically allocates each early exit's unskipped layer to an optimal device in model pipeline. The key idea of DensePipe is to offline profile all layers' information related to early exit, and use it to guide the layers' deployment on devices. At run-time, this deployment supports the dynamic scheduling of unskipped layers according to the system's latest status. In doing so, DensePipe balances the resource usages among different edge devices and maximizes the throughput of model pipeline in inference. In particular, the contributions of this paper are as follows:

- **Offline layer deployment planner** profiles the skip probabilities, memory footprint, and inference latency of all model layers in early exit, and outputs a deployment of layers such that each device has layers of both high and low skipping probabilities.
- **Online layer scheduler** takes the current layers' deployment, system load and resource utilizations as inputs, and allocates each layer to an optimal device to maximize the overall throughput of all inferences
- We implement a system prototype of DensePipe and conduct evaluation against the state-of-the-art techniques, i.e. Pico, PipeEdge, and AP2 using typical early-exit transformer models (ViT-Base and ViT-Large). Comparative experiments show that our approach achieves 3.46 x higher throughput on homogeneous devices and 2.72x higher throughput on heterogeneous devices, and works well within 6G networks.

Background

6G networks feature high bandwidth and low latency, promoting collaborative inference across multiple devices. When deploying large deep learning models (e.g. transformers) on resource-constrained edge devices, **model parallelism** splits a model into multiple layers and distributes them across multiple devices for collaborative inference. This technique improves inference throughput via model parallelism and it can be divided into two categories: tensor parallelism and pipeline parallelism.

Tensor parallelism splits the tensors of large models into multiple devices for parallel inference. For example, MegatronLM [8] introduces a one-dimensional tensor parallelism method for Transformer models based on matrix partitioning rules. It divides the weight matrices of the self-attention layer and the feed-forward network layer along the column dimension, and distributes them to different devices. However, such fine-grained tensor parallelism requires numerous all-reduce communication operations, leading to significant communication costs in collaborative inference between edge devices.

Pipeline parallelism partitions the model into multiple stages (each stage is a set of consecutive layers) and deploys each stage to a device. On each device, pipeline parallelism ensures that the forward operation of one inference is overlapped with the communications of another inference. It significantly reduces the frequency and volume of communication between devices and brings lower latency compared to tensor parallelism. Existing pipeline parallelism techniques has three types.

The first type is designed for data centers with homogeneous machines [4], [5], [9], [10], such as GPipe [4] and PipeDream [5]. GPipe [4] equally partitions the model layers into devices, and

PipeDream [5] further considers the topology of homogeneous devices in partitioning.

The second type further considers heterogeneous devices, where each device has several heterogeneous GPUs. For example, HetPipe [11] first aggregates multiple heterogeneous GPUs into a single virtual worker, and then uses pipeline model parallelism within the virtual worker and data parallelism among different virtual workers.

The third type focuses on heterogeneous edge devices. PipeEdge [1] considers the computing capacity, communication, and memory of devices during partitioning layers. AP2 [2] views the inference process of each micro-batch in each stage as a separate sub-inference, and uses a genetic algorithm to optimize the allocation and scheduling of all sub-inferences across devices. PICO [12] views a CNN as a directed acyclic graph, and partitions it using a graph partitioning algorithm. Specifically, it first partitions the stages according to the average resources of devices and then fine-tunes the partitions.

Moreover, **early exit** [3] allows inference to terminate once the desired model accuracy is achieved. It adds early-exit branches into the middle of the model, and allows simple samples to early exit from these branches during inference. when applying in Transformers, it adds early-exit branches between each two adjacent Transformer layers [13], [14], [15].

When applying both techniques in resource-constrained edge devices, the static deployment of layers in pipeline and the unpredictable skips of these layers in early exit may cause unbalanced loads and resource utilizations across devices, leading to considerable performance degradation.

DensePipe

Overview

DensePipe is designed with two objectives: (1) how to deploy early-exit layers into devices according to layers' skipping probabilities (the section "Offline Layer Deployment Planner"); and (2) how to online schedule early-exit layers based on the inference sample's unskipped layers and current loads of devices (the section "Online Layer Scheduler"). Figure 2 illustrates the architecture of DensePipe, which is divided into an offline stage and an online stage.

At the offline stage, DensePipe consists of two modules: the *layer profiler* and the *layer deployment planner*. Given an early-exit model, the *layer profiler* first analyzes the memory footprint and the inference latency of layer on edge device. Existing pipeline methods only consider this information when performing pipeline stage partitioning, leading to low utilization of devices with less used layers. In contrast, DensePipe's profiler further collects the layers' skipping probabilities by performing inference on all samples in the training dataset. Based on these profiles and the available resource of each device, the *layer deployment planner* outputs a deployment solution, which deploys layers with higher skipping probabilities to more powerful devices and vice versa.

Formally, let P_i be layer i's skipping probability, C_j be device j's computing capacity, and $O_{i,j}$ be a binary variable representing whether layer i is deployed on device j($O_{i,j} = 1$) or not ($O_{i,j} = 0$). The planner aims to find a solution $O_{i,j}$ that minimizes the sum of the product of each layer's skipping probabilities P_i and each device's computing capacity C_j (where j satisfies $O_{i,j} = 1$), as shown in Figure 2.

At the online stage, the *online layer scheduler* operates on each unskipped layer of an inference sample. Specifically, the scheduler takes three information as input: the layers' offline deployment, an inference table that maintains all devices' current inference tasks, and all the information of the current inference. Formally, the scheduler selects the device d with the smallest

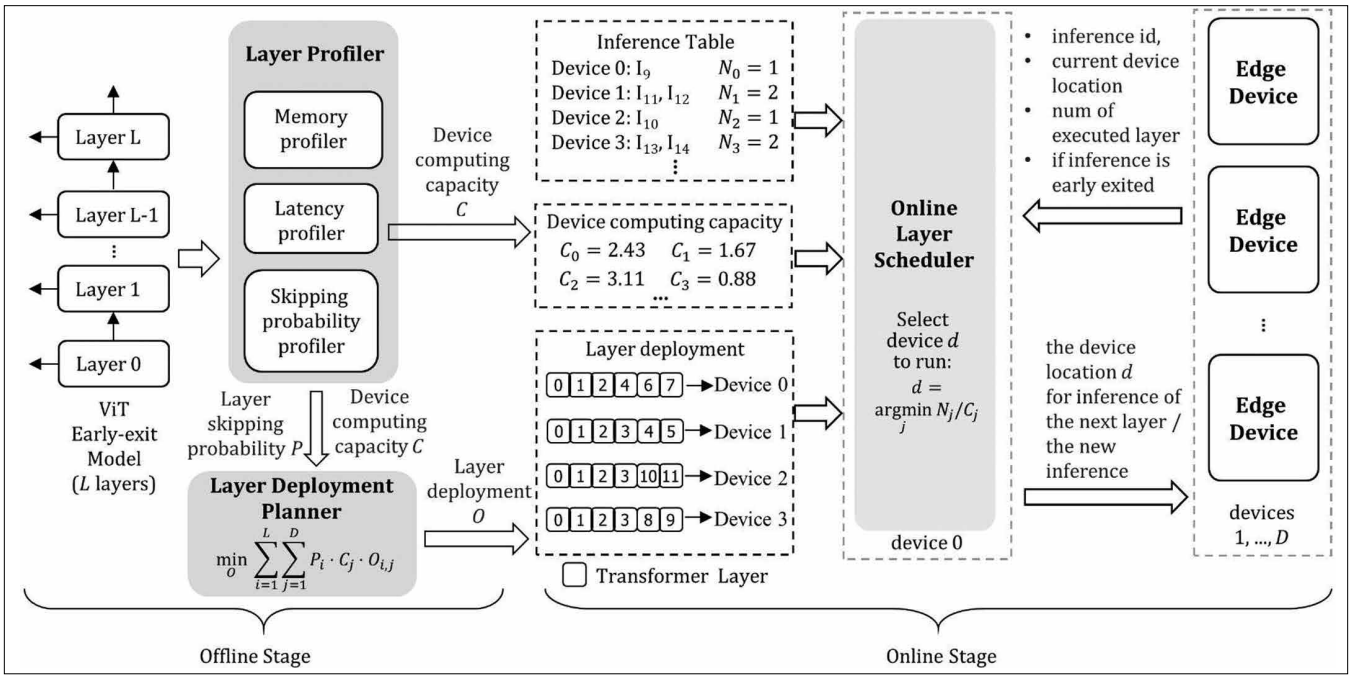

FIGURE 2. DensePipe architecture.

computational load N_d/C_d to process the scheduled layer i, where N_d is the number of running inferences on device d.

Offline Layer Deployment Planner

1) Layer Profiler: DensePipe has three types of layer profiler: memory profiler, latency profiler, and skipping probability profiler. The *memory profiler* obtains the memory usage of each layer by sequentially loading the layers of the given model. The *latency profiler* measures the inference latency of each layer on each device, and calculates the computational capability of each device as the inverse of model inference latency on this device. Finally, the *skipping probability profiler* includes two steps: step 1 lets the early-exit model perform inference on all samples in the pre-training dataset; step 2 computes the skipping probability of a layer, which is defined as the ratio of the number of times this layer is skipped to the number of samples. Note that at the online stage, the profiling result can be calibrated according to the latest collected information.

2) Layer Deployment Planner: With profiling information, the planner decides how to allocate model layers to different devices. The planner treats this as an integer linear programming problem with an objective and three constraints. The *objective* is to minimize the sum of the product of each layer's skipping probabilities and each device's computing capacity. This ensures that more powerful devices are deployed with lower skip probability layers. The *constraints* are that: (1) for each device, the memory footprint of deployed layers must not exceed the available memory; (2) each layer is deployed on at least one device; and (3) for any two adjacent layers with similar skipping probabilities, the planner limits the difference between their numbers of deployed devices. This ensures the computational load from the previous layer does not congest in the next layer. The planner solves this optimization problem to output the deployment of layers.

The feasibility of the planner is based on two assumptions. First, it assumes that the throughput is indeed higher if the layers with lower skip probabilities are deployed to more powerful devices. Second, it assumes that the optimization problem can be solved if the sum of all devices' available memories is larger than the sum of all layers' memory footprint. The planner employs the branch-and-bound algorithm to solve the optimal solution.

Analysis of Overhead. The planner's overhead can be divided into three parts: (1) the layer profiler sequentially loads each layer and performs one inference to test memory and latency. Its time usage is the sum of the whole model's load time and inference latency at edge; (2) the layer profiler tests skipping probability on the powerful server. Its time usage is the inference latency on cloud multiplied by the number of training samples; (3) for the layer deployment planner, the number of candidate solutions is up to $2^{D \cdot L}$, where D is the number of devices and L is the number of layers. The planner adopts the branch-and-bound algorithm to find the best one, and it has a polynomial time complexity with respect to D · L. For instance, given a model ViT-base (L = 12), three Raspberry Pi 4 B boards (D = 3), and a server with NVIDIA Quadro RTX 8000, the above three parts take 100ms, 3 minutes and 200 ms to complete, respectively. Note that both the profiler and the planner are executed at the offline stage once.

Online Layer Scheduler

At run-time, the online layer scheduler determines the optimal device to process each inference's unskipped layers, so as to balance load among all devices to maximize the overall pipeline throughput. Once the processing of a layer is completed, the scheduler selects a device for processing this inference's next unskipped layer. The scheduler maintains an inference table to record currently running inferences on each device. In each scheduling, it accesses this table to calculate the ratio of the number of running inferences on each device divided by the computational capacity of the corresponding device. The scheduler then selects the device with the lightest load as the execution device for that layer, regardless of whether the selected device is already occupied by some running inferences or not. The scheduler is feasible and the selected device is always optimal. This is because the calculated load is proportional to the that layer's waiting time before processing. By selecting the device with the lightest load, the layer's waiting time is minimized and brings higher throughput.

The scheduler is designed to handle three conditions. First, if some running inferences already occupy the selected device, the newly scheduled inference waits until they complete. Second, if no device has sufficient memory to run the new inference, the scheduler waits until some running inferences are completed and release enough memory. Finally, two devices will encounter a deadlock when they need to send information to each other but do not have sufficient memory to store the received information. In this case, the scheduler first selects some layers not used in inference at present and then offloads them to hard disk, thus releasing some memory to resolve the deadlock.

Discussion of communication overhead with 6G networks. The scheduler minimizes communication overhead from two aspects. First, it compactly stores the transferred information (e.g., device statuses and layers' outputs) in hash tables and tensors. For ViT-base model, they take only 0.78 MB of memory and can be quickly transmitted within the 6G network (using 10 ms under 1 Gbps bandwidth). Second, the scheduler concurrently executes data transmission and inference computation. By doing so, most of the communication time can be hidden behind inference latency and does not reduce throughput. Moreover, the scheduler can further reduce communication time by being aware of the network bandwidth. For example, the scheduler can prioritize the device with the highest bandwidth and light/medium load, rather than the device with the lightest load but low bandwidth.

Running Example

Figure 3 illustrates an example where DensePipe first deploys a model's layers to three devices offline, and then schedules inference I_3's layer 2 to device 2 after its layer 1 is processed on device 3.

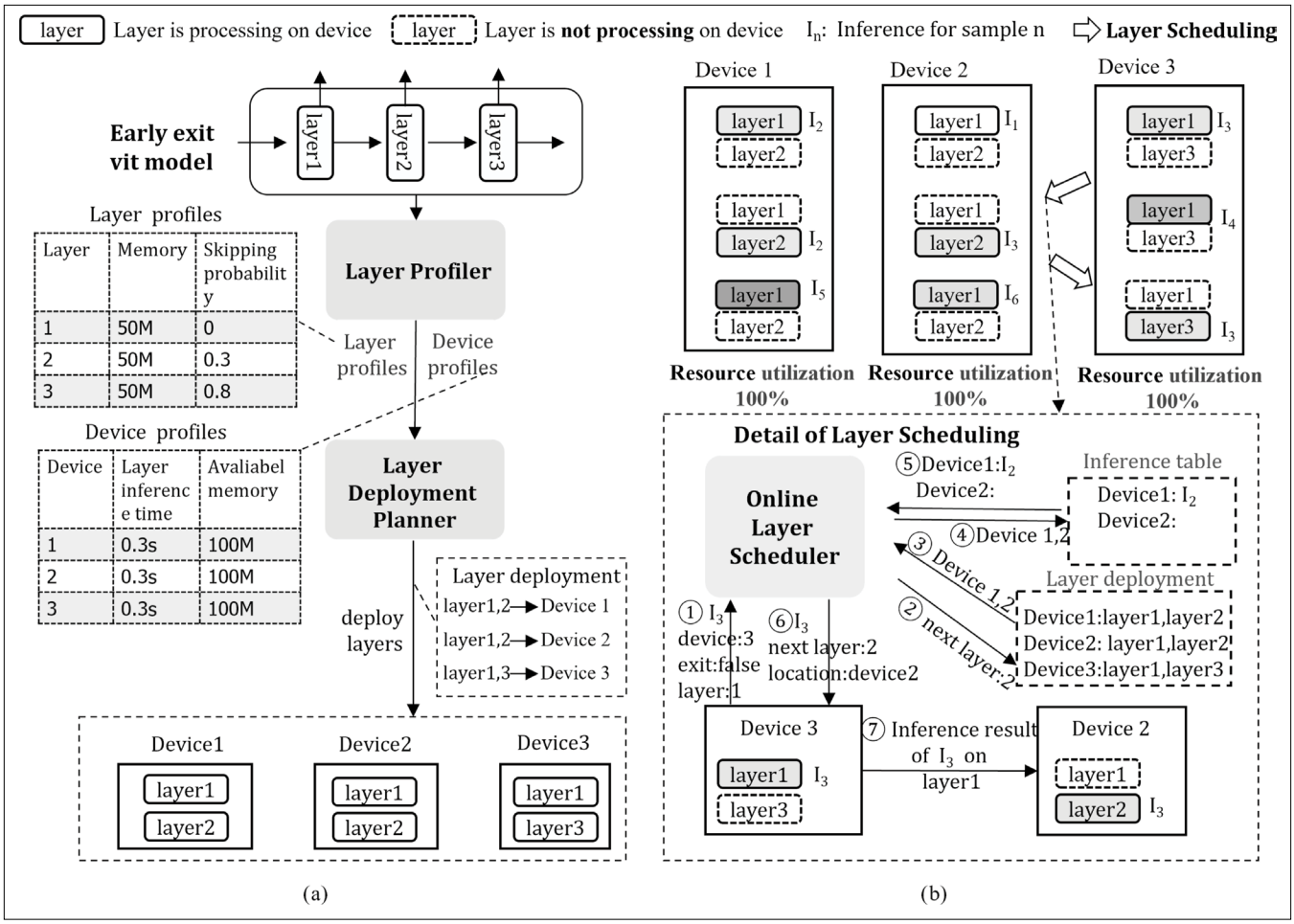

FIGURE 3. A example of scheduling layer with DensePipe. a) An example in offline stage. b) An example in offline stage.

At the offline stage, as shown in Figure 3(a), the *layer profiler* profiles an early exit model with three identical layers, and outputs both model profiles and device profiles. The model profiles show that each layer has a memory usage of 50 MB and skipping probabilities ranging from 0 to 0.8. The device profiles show that the computation time for each layer on each device is 0.3 s, and the available memory for each device is 100 MB. Using these profiles, the *layer deployment planner* solves the optimization problem to generate a deployment solution: layer 1 with the lowest skipping probability is deployed on all three devices, and layer 3 with the highest probability is only deployed on device 3.

At the online stage, after device 3 processes layer 1 of inference I_3's and layer 2 is not skipped in early exit, the *online layer scheduler* searches the optimal device for layer 2 using 7 steps, as illustrated in Figure 3(b). Step 1 sends the following information to the scheduler: the ID of I_3 (3), the early exit status (false), and the current execution count (1). Step 2 queries the Layer deployment obtained during the offline stage to identify the devices with layer 2. Step 3 finds that layer 2 is deployed on devices 1 and 2. Step 4 queries the Inference table for the current inferences on these two devices. Step 5 finds that device 1 is processing inference I_2 and device 2 is idle. Step 6 instructs device 3 to perform layer 2 of inference I_3. Finally, step 7 sends the processing results of I_3's layer 1 to device 2. In this example, the scheduler successfully balances the loads among three devices and hence improves the overall pipeline throughput in inference.

Evaluation

In this section, we evaluate DensePipe against the state-of-the-art methods in three scenarios: homogeneous edge devices (the section "Homogeneous Edge Devices"), heterogeneous edge Devices (the section "Heterogeneous Edge Devices") and simulated 6G environment (the section "Simulation of 6G Scenario").

Experimental Settings

Testbeds. We select Raspberry Pi 4B devices with 8GB memory and up to 1.8 Ghz CPU frequency, and test both homogeneous and heterogeneous edge devices by adjusting their CPU frequencies and available memories using *cgroup* tools.

Models and Dataset. We use the early-exit ViT-Base and ViT-Large pre-trained on CIFAR-10 image classification dataset. We add the early-exit branches between each two consecutive layers [13].

Compared Baselines. We implement and compare six state-of-the-art pipeline parallel methods: (1) AP2 is specifically designed for 6G scenarios; (2) *PICO* first performs homogeneous pipeline partitioning and then adapts to heterogeneous scenarios; (3) *PipeDream* and *GPipe* equally

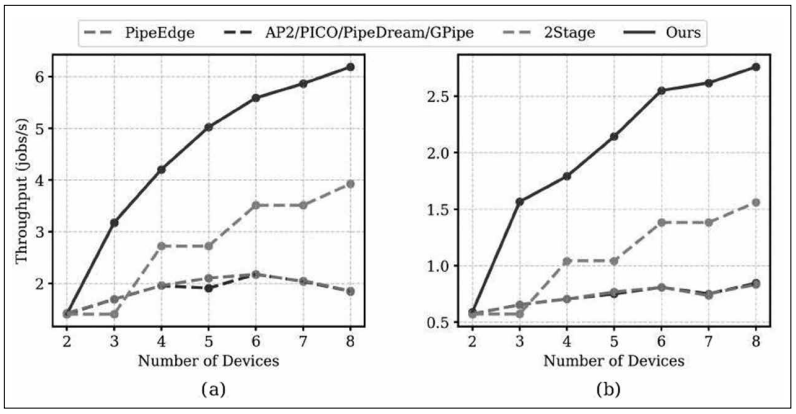

FIGURE 4. Throughput versus Number of Devices in Homogeneous Scenarios. a) ViT-Base with early exit on Cifar10. b) ViT-Large with early exit on Cifar10.

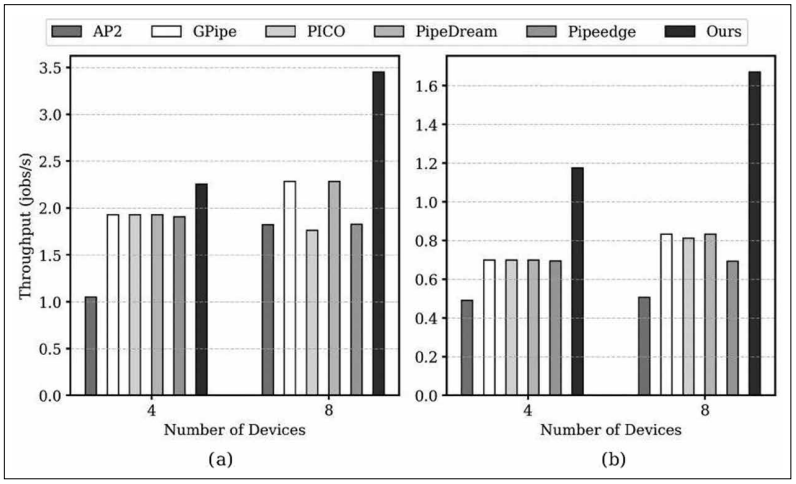

FIGURE 5. Throughput versus Number of Devices in Heterogeneous Scenarios. a) ViT-Base with early exit on Cifar10. b) ViT-Large with early exit on Cifar10.

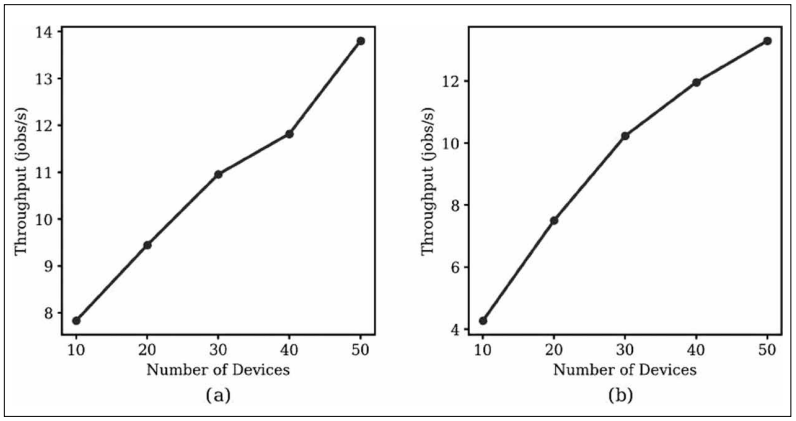

FIGURE 6. Throughput versus Number of Devices in Large-Scale 6G Device Interconnection Scenarios. a) ViT-Base with early exit on Cifar10. b) ViT-Large with early exit on Cifar10.

divides the pipeline based on the number of devices; (4) 2Stage incorporates data parallelism and forms a pipeline between each two devices; and (5) *PipeEdge* considers the device heterogeneity and optimizes the pipeline planning for such scenario.

Metrics. We evaluate the *throughput* of each method in our experiments, which is defined as the number of processed images per second.

Homogeneous Edge Devices

Setting. We conduct this evaluation using 2 to 8 devices with the same maximal CPU frequency and available memory.

Results. In Figure 4, we can see DensePipe achieves the highest throughput compared to all baselines. This is because DensePipe deploys layers with higher computation probabilities on as many devices as possible and thus fully utilizes the computational resources on these devices. The results show that DensePipe's throughput increases linearly when the number of devices. In contrast, AP2, PICO, PipeDream, and GPipe equally divide the pipeline into several stages/layers and deploy each stage to a device. Hence with early-exit layers, the devices deployed the former layers undertake most of the computations, and other idle devices lower the overall system throughput. In conclusion, DensePipe achieves 3.35 x to 3.56 x (3.46x in average) higher throughput compared to baselines.

Heterogeneous Edge Devices

Setting. We conduct this experiment on 4 or 8 devices. For 4 devices, we set their CPU frequencies to 1.8GHz/1.2GHz/1.2GHz/0.6GHz, available memory to 100MB/200MB/300MB/150MB for ViT-Base, 300MB/600M/800M/400M for ViT-Large, respectively. For 8 devices, we set CPU frequencies of three devices to 1.8 GHz, three devices to 1.2 GHz, and one device to 0.6 GHz, respectively. We set three different available memories: (i) two devices: 150MB for ViT-Base, and 400M for ViT-Large; (ii) three devices: 200MB for ViT-Base, and 600M for ViT-Large; and (iii) three devices: 250MB for ViT-Base, and 800M for ViT-Large.

Results. Figure 5 shows that DensePipe still consistently achieves the highest throughput in heterogeneous device scenario. This is because DensePipe's planner considers layers' skipping probabilities in early exit and dynamically distributes unskipped layers to devices according to their distinct computing capacities and available resources at run-time. In contrast, although PipeEdge, PICO, and AP2 consider the device heterogeneity, their pipeline planning is still based on the pre-specified deployment of layers and overlooks the dynamic load and resource changes in the system. That is, these techniques always assign more inferences to faster devices, regardless of these devices' current load. This exacerbates the resource contention of these devices and also lowers the resource utilization of slower devices. As a result, their throughput can be even lower than the equal partitioning methods (e.g. PipeDream and GPipe). In conclusion, DensePipe achieves 2.15 x to 3.30 x (2.72x in average) higher throughput compared to baselines.

Simulation of 6G Scenario

Setting. We simulate 10 to 50 devices connected with the theoretical 6G connection speed, and test the throughput of DensePipe.

Results. As shown in Figure 6, the throughput of DensePipe increases linearly with the number of devices, indicating that DensePipe remains effective when deployed on more devices. This

is because DensePipe's heuristic online scheduling is near-optimal and has negligible overhead, hence works well with 6G network environment. In contrast, baseline techniques' throughput is still limited by the conflict between pre-specified layer deployment and dynamically changing resources.

Conclusion

This paper presents the design and evaluation of DensePipe, an approach that efficiently schedules early-exit layers in model pipeline parallelism. DensePipe maximizes the pipeline throughput by balancing the processing of all inferences' layers among edge devices according to the latest system status at run-time. We tested our method on typical transformer models and edge environment to validate its effectiveness in improving system throughput.

Our future work focuses on extending DensePipe in two directions. First, the current implementation of DensePipe is designed for encoder-based transformers and we will extend it to support decoder-based transformers (e.g. GPT and LLaMA) in natural language processing applications. These models have two stages of inferences, which require separate optimizations in layer deployment and scheduling. Second, DensePipe sequentially executes multiple inferences that are scheduled to the same device, incorporating it with batching techniques can further improve the throughput.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFE0209100; in part by the National Natural Science Foundation of China under Grant 62272046, Grant 62132019, Grant 61872337, and Grant U21A20519; and in part by the Open Project of Ministry of Education's Key Laboratory of Computing Power Network and Information Security under Grant 2023PY002.

References

[1] Y. Hu et al., "PipeEdge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices," in *Proc. 25th Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2022, pp. 298–307.

[2] H. Shi et al., "Automatic pipeline parallelism: A parallel inference framework for deep learning applications in 6G mobile communication systems," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 7, pp. 2041–2056, Jul. 2023.

[3] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit. (ICPR)*, 2016, pp. 2464–2469.

[4] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–10.

[5] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 1–15.

[6] A. Bakhtiarnia, Q. Zhang, and A. Iosifidis, "Single-layer vision transformers for more accurate early exits with less overhead," *Neural Netw.*, vol. 153, pp. 461–473, Sep. 2022.

[7] J. Xin et al., "DeeBERT: Dynamic early exiting for accelerating BERT inference," 2020, *arXiv:2004.12993*.

[8] M. Shoeybi et al., "Megatron-LM: Training multi-billion parameter language models using model parallelism," 2019, *arXiv:1909.08053*.

[9] W. Liu et al., "AutoPipe: A fast pipeline parallelism approach with balanced partitioning and micro-batch slicing," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2022, pp. 301–312. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9912711

[10] S. Zhao et al., "NASPipe: High performance and reproducible pipeline parallel supernet training via causal synchronous parallelism," in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Feb. 2022, pp. 374–387. [Online]. Available: https://dl.acm.org/doi/10.1145/3503222

[11] J. Park et al., "HetPipe: Enabling large DNN training on (Whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *Proc. USENIX Annu. Tech. Conf.*, Jan. 2020, pp. 307–321. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/park

[12] X. Yang et al., "PICO: Pipeline inference framework for versatile CNNs on diverse mobile devices," *IEEE Trans. Mobile Comput.*, vol. 23, no. 4, pp. 2712–2730, Apr. 2023.

[13] W. Liu et al., "FastBERT: A self-distilling BERT with adaptive inference time," 2020, *arXiv:2004.02178*.

[14] G. Xu et al., "LGViT: Dynamic early exiting for accelerating vision transformer," in *Proc. 31st ACM Int. Conf. Multimedia*, Oct. 2023, pp. 9103–9114.

[15] W. Zhu, "LeeBERT: Learned early exit for BERT with cross-level optimization," in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics 11th Int. Joint Conf. Natural Lang. Process.*, 2021, pp. 2968–2980.

Biographies

Yuxiao Liu (3220215156@bit.edu.cn) is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include edge intelligence and deep learning applications.

Rui Han (hanrui@bit.edu.cn) received the M.Sc. degree (Hons.) from Tsinghua University, China, in 2010, and the Ph.D. degree from Imperial College London, U.K., in 2014. He is currently an Associate Professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. His research interests are system optimization for cloud data center workloads (in particular highly parallel services and deep learning applications). He has over 40 publications in these areas, including papers at MobiCOM, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Knowledge and Data Engineering, INFOCOM, and ICDCS.

Qinglong Zhang (3120211050@bit.edu.cn) is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include edge intelligence and deep learning applications.

Haiting Hou is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology. His research interests include edge intelligence and deep learning applications.

Chi Harold Liu (Senior Member, IEEE) (chiliu@bit.edu.cn) received the B.Eng. degree from Tsinghua University, Beijing, China, and the Ph.D. degree from Imperial College London, London, U.K. He was at IBM Research China and Deutsche Telekom Laboratories, Berlin, Germany, and the IBM T. J. Watson Research Center, USA. He is currently a Full Professor and the Vice Dean of the School of Computer Science and Technology, Beijing Institute of Technology, Beijing. His current research interests include the big data analytics, mobile computing, and deep learning. He is a fellow of IET and the Royal Society of the Arts. He is also an Associate Editor of IEEE Transactions on Network Science and Engineering.

Lydia Y. Chen (Senior Member, IEEE) (lydiaychen@ieee.org) received the B.A. degree from National Taiwan University and the Ph.D. degree from Pennsylvania State University. She was a Research Staff Member at the IBM Zurich Research Laboratory from 2007 to 2018. She is currently an Associate Professor with the Department of Computer Science, Technology University Delft. She has published more than 80 articles in journals, such as IEEE Transactions on Distributed Systems, IEEE Transactions on Service Computing, and conference proceedings, such as INFOCOM, Sigmetrics, DSN, and Eurosys. Her research interests center around dependability management, resource allocation, and privacy enhancement for large scale data processing systems and services. She was a co-recipient of the Best Paper Awards at the CCgrid'15 and the eEnergy'15.