

EdgeShard: Efficient LLM Inference via Collaborative Edge Computing

Mingjin Zhang^{1b}, Xiaoming Shen, Jiannong Cao^{1b}, *Fellow, IEEE*, Zeyang Cui, and Shan Jiang^{1b}

Abstract—Large language models (LLMs) have shown great success in content generation and intelligent decision making for IoT systems. Traditionally, LLMs are deployed on the cloud, incurring prolonged latency, high bandwidth costs, and privacy concerns. More recently, edge computing has been considered promising in addressing such concerns because the edge devices are closer to data sources. However, edge devices are cursed by their limited resources and can hardly afford LLMs. Existing studies address such a limitation by offloading heavy workloads from edge to cloud or compressing LLMs via model quantization. These methods either still rely heavily on the remote cloud or suffer substantial accuracy loss. This work is the first to deploy LLMs on a collaborative edge computing environment, in which edge devices and cloud servers share resources and collaborate to infer LLMs with high efficiency and no accuracy loss. We design EdgeShard, a novel approach to partition a computation-intensive LLM into affordable shards and deploy them on distributed devices. The partition and distribution are nontrivial, considering device heterogeneity, bandwidth limitations, and model complexity. To this end, we formulate an adaptive joint device selection and model partition problem and design an efficient dynamic programming algorithm to optimize the inference latency and throughput. Extensive experiments of the popular Llama2 serial models on a real-world testbed reveal that EdgeShard achieves up to 50% latency reduction and 2× throughput improvement over the state-of-the-art.

Index Terms—Cloud-edge-end Collaboration, edge artificial intelligence (AI), edge computing, large language models (LLMs).

I. INTRODUCTION

RECENTLY, the emergence of large language models (LLMs) has attracted widespread attention from the public, industry, and academia, representing a significant breakthrough in artificial intelligence (AI). Many players are coming into this field with their advanced models, such as OpenAI’s GPT-4 [1], Meta’s Llama [2], and Google’s PALM [3]. Built on the foundation of transformer architecture [4], LLMs are characterized by their massive scale in terms of the number of parameters and the amount of data they are trained on. The scale of LLMs, often numbering

in hundreds of billions of parameters, enables the models to capture complex patterns in language and context, making them highly effective at generating coherent and contextually appropriate responses. Such a phenomenon is also known as “intelligence emergence.” The outstanding capability of LLMs makes them valuable and well-performed in a wide range of applications, from ChatBot and content generation (e.g., text summation and code generation) to assisting tools of education and research.

However, current LLMs heavily rely on cloud computing, suffering from long response time, high bandwidth cost, and privacy concerns [5]. First, the reliance on the cloud computing hampers the capability for rapid model inference necessary for real-time applications, such as robotics control, navigation, or exploration, where immediate responses are crucial. Second, the transmission of large amounts of data, including texts, video, images, audio, and IoT sensing data, to the cloud data centers leads to substantial bandwidth consumption and immense strain on the network architecture. Third, the cloud-based LLMs raise significant privacy issues, especially when handling sensitive data of hospitals and banks, as well as personal data like text inputs and photos on mobile phones.

Edge computing is a promising solution to address the aforementioned challenges by deploying LLMs on the edge devices (e.g., edge servers, edge gateways, and mobile phones) at the network edge closer to the data sources [6]. However, LLMs are computation-intensive and resource-greedy. For example, the inference of a full-precision Llama2-7B model requires at least 28 GB memory, which may exceed the capacity of most edge devices. Some works leverage model quantization [7], [8], [9], [10], [11], [12] to reduce the model size to fit into the resource-constraint edge devices. However, they often lead to accuracy loss. Other works tend to use the cloud-edge collaboration [13], [14], which partitions the LLMs into two submodels and offloads part of the computation workload to the powerful cloud servers with high-end GPUs. However, the latency between the edge devices and the cloud servers is usually high and unstable.

Alternatively, we have witnessed the continuous growth of the computing power of edge in recent years, and a large number of edge servers and edge clouds have been deployed at the network edge, leaving significant resources to be used. Collaborative edge computing (CEC) [15], [16] is hence proposed recently to integrate the computing resources of geo-distributed edge devices and cloud servers for efficient resource utilization and performance optimization. As shown in Fig. 1, ubiquitous and distributed edge devices and cloud servers are connected and form a shared resource

Received 8 October 2024; revised 7 December 2024; accepted 19 December 2024. Date of publication 31 December 2024; date of current version 9 May 2025. This work was supported by the Research Institute for Artificial Intelligence of Things, The Hong Kong Polytechnic University, HK RGC Grant for Theme-Based Research Scheme under Grant T43-513/23-N, and in part by the Hong Kong RGC Collaborative Research Fund under Grant C5032-23G. (Corresponding author: Shan Jiang.)

The authors are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, SAR, China (e-mail: mingjin.zhang@connect.polyu.hk; xiaoming.shen@polyu.edu.hk; jiannong.cao@polyu.edu.hk; zeyang.cui@polyu.edu.hk; shanjiang@polyu.edu.hk).

Digital Object Identifier 10.1109/IIOT.2024.3524255

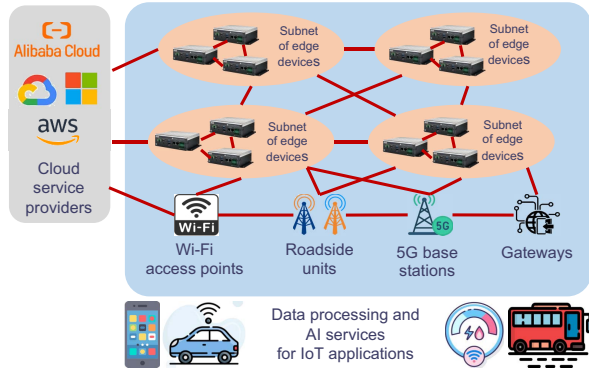


Fig. 1. CEC integrates the computing resources of ubiquitous geo-distributed devices for jointly performing computational tasks, with great benefits of enlarged resource pool, low-latency data processing, flexible device access, and expanded service region.

pool, collaboratively providing instant data processing and AI services. CEC is different from existing edge computing research. Existing edge computing research focuses on the vertical collaboration among cloud, edge, and end devices, while neglecting horizontal edge-to-edge collaborations, suffering from unoptimized resource utilization, restricted service coverage, and uneven performance.

Motivated by the vision of CEC, we propose a general LLM inference framework, named EdgeShard, to support efficient collaborative LLM inference on distributed edge devices and cloud servers. For simplicity, we use computing devices below to refer to the edge devices and cloud servers. Given a network with heterogeneous computing devices, EdgeShard partitions the LLM into multiple shards and allocates them to judicious devices based on the heterogeneous computation and networking resources, as well as the memory budget of devices. To optimize performance, we formulate a joint device selection and model partition problem and design an efficient dynamic programming algorithm to minimize the inference latency and maximize the inference throughput, respectively. Extensive experiments on a practical testbed show that EdgeShard reduces up to 50% latency and achieves $2\times$ throughput over on-device and vertical cloud–edge collaborative inference methods.

This work is distinguished from existing work by partitioning the LLMs and allocating to multiple GPUs in cloud data centers, such as Gpipe [17], PipeDream [18], and Splitwise [19]. First, cloud servers are usually with homogeneous GPUs, while the edge devices are with heterogeneous computation capabilities in nature. Second, modern cloud GPUs for LLMs are usually connected by high-bandwidth networks, such as InfiniBand and Nvlinks, while the edge devices are connected with heterogeneous and low-bandwidth networks. For example, the bandwidth of NVlinks can go up to 600 GB/s, while the bandwidth among edge devices ranges from dozens of Kb/s to 1000 Mb/s. The solutions of LLMs deployment designed for cloud data centers neglect the heterogeneous and resource-constrained edge computing environment. Meantime, EdgeShard also distinguishes from the previous works [20], [21], [22] partitioning small-size DNNs into distributed edge devices for collaborative inference,

as LLM is large in parameter size and its inference is autoregressive, while our work solves the problem. Moreover, they only consider the model partition problem over a fixed number of devices, while EdgeShard goes further, performing joint device selection and model partitioning. The main contributions of this work are as follows.

- 1) We propose a general LLM inference framework for deploying LLMs in the edge computing environment, which enables the collaborative inference among heterogeneous edge devices and cloud servers.
- 2) Further, we quantitatively study how to select computing devices and how to partition the LLM for optimized performance. We mathematically formulate a joint device selection and model partition problem, and propose a dynamic programming algorithm to optimize the latency and throughput, respectively.
- 3) We also evaluate the performance of EdgeShard with state-of-the-art Llama2 serial models on a physical testbed. Experimental results show EdgeShard remarkably outperforms various baseline methods.
- 4) Additionally, we outline several open issues, such as incentive mechanisms, batch-size-aware optimization and privacy-preserving techniques, to guide further advancements in collaborative LLM inference at the edge.

II. PRELIMINARIES AND MOTIVATIONS

Generative LLM Inference: LLMs generally refer to decoder-based transformer models with billions of parameters. Different from encoder-based architecture like BERT [23], whose inference process is single phase, the process of LLM inference is iterative and typically involves two phases: 1) the prompt processing phase and 2) the autoregressive generation. The prompt processing phase is also known as prefill.

In the prompt processing phase, the model takes the user initial token (x_1, \dots, x_n) as input and generates the first new token x_{n+1} by computing the probability $P(x_{n+1} | x_1, \dots, x_n)$.

In the autoregressive generation phase, the model generates one token at a time, based on both the initial input and the tokens it has generated so far. This phase generates tokens sequentially for multiple iterations until a stopping criterion is met, i.e., either when generating an end-of-sequence (EOS) token or reaching the maximum number of tokens specified by user or constrained by the LLM.

A decoder-based LLM model usually consists of an embedding layer, followed by repetitive linearly-connected transformer layers, and an output layer. As shown in Fig. 2, suppose the LLM model has N layers, which will take a sequence of input tokens and run all layers to generate a token in a one-by-one manner. In the prefill phase, the model takes the input (“Today is a”) at once, and the first generated token is “good.” In the autoregressive generation phase, the model first takes (“Today is a good”) as input and generates the next token (“day”). It then takes (“Today is a good day”) as input and generates the next token (“EOS”), which indicates the end of the generation. Since a token generated is determined by all its previous token in a

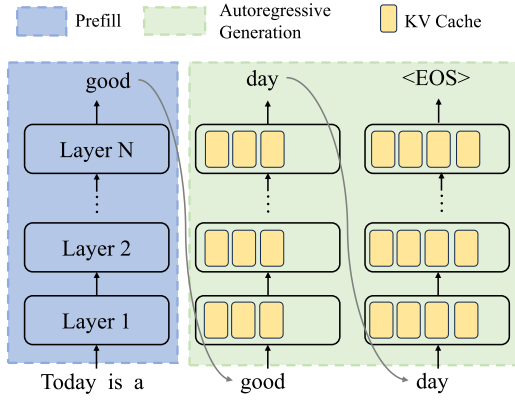


Fig. 2. LLM inference has an autoregressive nature.

TABLE I
MINIMUM MEMORY USAGE OF LLMs INFERENCE AND
MEMORY CAPACITY OF EDGE DEVICES

Model	Full Precision	8-bit	4-bit	Edge Devices
Llama2-7B	28GB	7GB	3.5GB	Smartphone(6-12GB)
Llama2-13B	52GB	13GB	6.5GB	Jetson Orin(8-16GB)
Llama2-70B	280GB	70GB	35GB	Jetson AGX(32-64GB)

sequence, LLMs utilize key-value caching (KV caching) to avoid repetitive computation. Each transformer layer stores past computations to expedite responses, thereby reducing computational workload and improving response times. The time to generate a token in the prefill stage is much higher (usually 10 \times) than that of in the autoregressive stage, as the prefill stage needs to calculate the KV cache of all input tokens as initialization.

LLMs Are Memory Consuming: A single edge device may not have sufficient memory to accommodate an LLM model. Take one of the most popular LLM models, i.e., Llama2, as an example. As shown in Table I, Llama2 has three different versions, i.e., 7B, 13B, and 70B. We can see from the table that the full precision inference of Llama2-7B requires at least 28 GB memory, but the smartphones usually only have 6–12 GB memory, and the Jetson Orin NX has 8–16 GB memory. They are unable to burden the on-device LLM inference. Some works try to use low-precision quantization, e.g., 8- and 4-bit. However, it may still exceed the memory capacity of the edge devices. For example, the 4-bit inference of Llama2-70B requires at least 35 GB memory, which cannot be accommodated on most edge devices. Moreover, low-precision inference leads to performance degradation.

In this work, we leverage CEC, a computing paradigm where geo-distributed edge devices and cloud servers collaborate to perform computational tasks. Based on that idea, we propose EdgeShard, a general LLM inference framework that allows adaptive device selection and LLM partition over distributed computing devices, to address the high memory requirements and leverage heterogeneous resources to optimize LLM inference.

III. COLLABORATIVE EDGE COMPUTING FOR LLMs

There are three stages of the framework, including profiling, task scheduling optimization, and collaborative inference. The workflow is shown in Fig. 3.

Profiling is an offline step that profiles the necessary run-time traces for the optimization step and only needs to be done once. Those traces include: 1) the execution time of each layer on different devices; 2) the size of activations and memory consumption for each layer of the LLM model; and 3) available memory of each device and the bandwidth among devices. For the execution time of each layer, we profile the time to generate a token in the prefill stage and autoregressive stage, respectively, and take the average. For those devices that may not have efficient memory to hold the full model for performing the profiling, we utilize a dynamic model loading technology, where the model layers are consecutively loaded to fit the constrained memory. More specifically, we first calculate the memory consumption of each layer based on the model architecture. According to the layer's memory consumption and the memory budget of a given device, we can estimate the maximum layers that the device can hold, based on which we partition the model into several shards and sequentially load the shards into the device. Moreover, during the dynamic loading, the output/activations of the previous shard are stored and act as the input for the next shard. The profiling information will then be used to support intelligent task scheduling strategies.

Scheduling Optimization: At the task scheduling optimization stage, the scheduler generates a deployment strategy by determining which device to participate in the inference, how to partition the LLM layer-wisely, and which device should the model shard be allocated to. The strategy comprehensively considers the heterogeneous resources, memory budget of devices, and privacy constraints and can later be applied to selected devices for efficient LLM inference. More details are described in Section IV.

Collaborative Inference: After getting the LLM model partition and allocation strategy, the selected devices will perform the collaborative inference. A critical issue is the KV-cache management, as the size of KV-cache will increase with length of the generated tokens. If the KV cache of previous tokens is overwritten by a new one, it can severely impact the model's output. To address this issue, we preallocate memory space for KV cache on each participating device. More specifically, considering the fact that each transformer layer will have its own KV cache and layer is the basic unit of the partition strategy, each device could maintain the KV cache of its allocated transformer layer. We adopted the greedy principle, where each device will preallocate the space to fit the maximum memory consumption of the KV cache. Suppose the number of allocated transformer layers is N_a , the batch size of the input tokens is B_b , the maximum length of input tokens is I_m , and the maximum length of generated tokens is G_m , then the size of preallocated KV cache space can be calculated as $N_a \cdot B_b \cdot (I_m + G_m)$. For the collaborative inference, we consider two cases, i.e., sequential inference and pipeline parallel inference.

In sequential inference, devices take turns to perform the computation with the allocated model shards. As shown in Fig. 4(a), suppose the LLM model is Llama2-7b, which has 32 layers in total, including 30 transformer layers, the model is partitioned into three shards, i.e., layer 1–16, layer 17–26, and layer 27–32, are allocated to device 1–3, respectively.

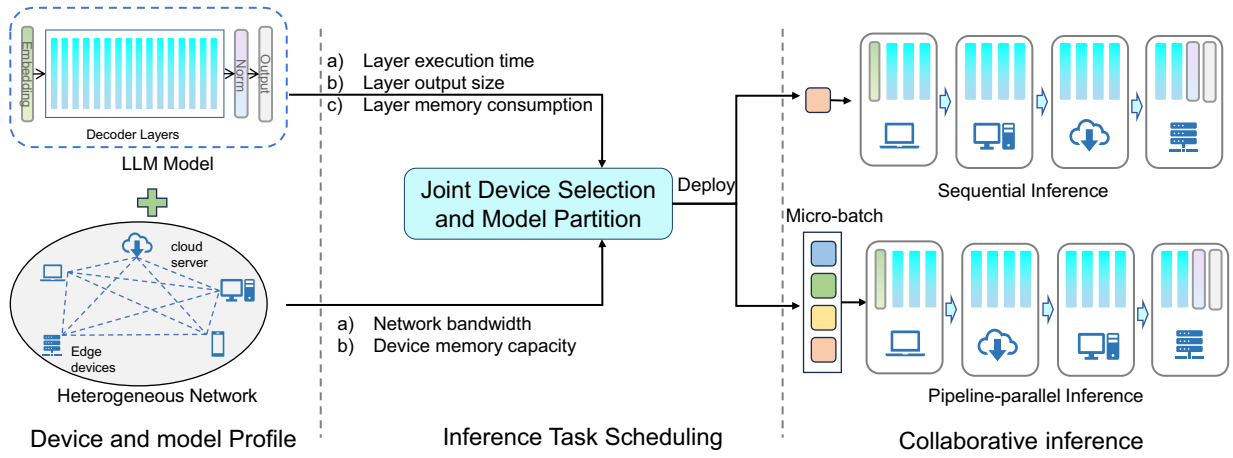


Fig. 3. Framework of EdgeLLM. It consists of three stages: 1) offline profiling; 2) task scheduling optimization; and 3) online collaborative LLM inference.

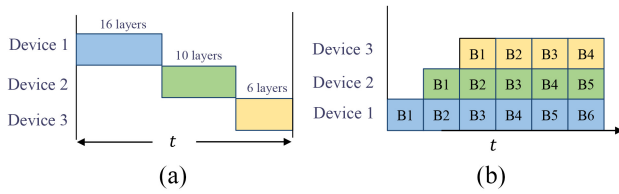


Fig. 4. Collaborative LLM inference. (a) Sequential inference. (b) Pipeline parallel inference.

Device-1 will first process the input tokens and then send the activations/outputs to device-2, which will process the data and then transmit to device-3. Device-3 will send back the generated token to device-1, which will then continue to process the new input tokens. This loop will repeat for multiple iterations until generating an EOS token or reaching the maximum number of generated tokens. Sequential inference is suitable for serving a single user, such as in smart home scenario, where users' personal devices (e.g., tablet, phones, and smart speaker) collaborate to perform LLM inference. In such scenario, user inputs a prompt and gets the response and then input another prompt. We aim to minimize the latency of sequential inference.

However, sequential inference is not resource-efficient from the system's perspective. When device-1 is performing computation, devices-2 and -3 are idle. We thus take pipeline parallelism to improve resource utilization. For the pipeline parallel inference as taken in the previous work Gpipe [17] and PipeDream [18] for the cloud servers, the input data will first be split into micro-batch and subsequently feed into the system. As depicted in Fig. 4(b), device-1 first handles data B1 and then transmits intermediate data to device-2. After handling data B1, device-1 immediately goes to handle data B2. In such a pipeline manner, every device is busy with high system resource utilization.

IV. OPTIMIZE LLM INFERENCE

We consider a general collaborative edge network with heterogeneous devices and bandwidth connection. More specifically, given a set of heterogeneous devices connected

TABLE II
LIST OF NOTATIONS

Symbol	Descriptions
$X_{i,j}$	binary variable, whether layer i of a model is allocated to device j
$t_{comp}^{i,j}$	computation time of layer i on device j
$t_{comp}^{i \rightarrow m,j}$	computation time of layer i to layer m on device j
$t_{comm}^{i-1,k,j}$	communication time to transmit activations of layer $i-1$ from device k to device j
$DP(i,j)$	minimal total execution time of the first i layers if layer i is allocated to device j
$g(i,S,k)$	processing time of the slowest node to process the first i layers with device set S

with heterogeneous bandwidth, EdgeShard aims to select a subset of devices and partition the LLM into shards, which will be allocated to the selected devices to minimize the inference latency or maximize the throughput.

System Model: LLMs usually have a layered architecture, which consists of an embedding layer, multiple decoder layer, and an output layer. Sizes of parameters and activations (i.e., the output of a layer) vary across layers. We assume the model is with N layers. O_i represents the size of activations of layer i , $0 \leq i \leq N-1$. The memory consumption of a layer i is denoted by Req_i .

We consider a network consisting of M edge devices and cloud servers. The devices have heterogeneous computation and memory capabilities, and the cloud servers are much more powerful than the edge devices in terms of computation capability. The memory budget of a device j is Mem_j . The computing devices are interconnected. Bandwidth between a device k and a device j is $B_{k,j}$, $0 \leq k \leq M-1$, and $0 \leq j \leq M-1$. There is a source node where the input tokens reside. Without loss of generality, we set the source node as node 0. The main notations used in this article are shown in Table II.

A. Optimize LLM Inference Latency

Problem Formulation: We use a binary variable $X_{i,j}$ to denote the LLM allocation strategy. $X_{i,j}$ equals to 1 if layer

i is allocated to node j . Otherwise, $X_{i,j}$ equals to zero. A layer will be and only be allocated to one node. Hence, we have $\sum_{j=0}^{M-1} X_{i,j} = 1 \forall i$. Let $t_{\text{comp}}^{i,j}$ denotes the computation time of layer i on node j . Suppose layer $i-1$ and layer i are allocated to nodes k and j , respectively. We use $t_{\text{comm}}^{i-1,k,j}$ to denote the communication time to transmit the activations of layer $i-1$ from node k to node j . The data transmission time is determined by the output size of a layer and the bandwidth between two nodes. If layer $i-1$ and layer i are on the same node, we assume the transmission time is zero. Hence, we have

$$t_{\text{comm}}^{i-1,k,j} = \begin{cases} \frac{O_{i-1}}{B_{k,j}}, & \text{if } k \neq j \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

The total inference time can thus be calculated by the following equation:

$$T_{\text{tol}}(X) = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} X_{i,j} \cdot t_{\text{comp}}^{i,j} + \sum_{i=1}^{N-1} \sum_{j=0}^{M-1} \sum_{k=0}^{M-1} X_{i-1,k} \cdot X_{i,j} \cdot t_{\text{comm}}^{i-1,k,j}. \quad (2)$$

Hence, the problem of minimizing the LLM inference latency can be formulated as follows, where (4) is the privacy constraint. It shows that the first layer of the LLM model should always be allocated to node 0, which is set to be the source node with input tokens. In such a case, the raw input data resides on the source node and avoids to be transmitted among computing devices. Equation (5) shows that the memory requirements of all the layers allocated to node j cannot exceed its memory budget. Equation (6) shows that a layer can be and only be allocated to one node

$$\min_{X_{i,j}} T_{\text{tol}}(X) \quad (3)$$

$$\text{s.t. } X_{0,0} = 1 \quad (4)$$

$$\sum_{i=0}^{N-1} X_{i,j} \cdot \text{Req}_i \leq \text{Mem}_j \quad \forall j \quad (5)$$

$$\sum_{j=0}^{M-1} X_{i,j} = 1 \quad \forall i. \quad (6)$$

Solution: To minimize the inference latency, we design a dynamic programming algorithm. The intuition is that the minimal execution time of the first i layer is determined by the first $i-1$ layer, which means the optimal solution can be constructed from the optimal results of the subproblems. It has the optimal subproblem property, which motivates us to use dynamic programming.

Let $DP(i, j)$ denote the minimal total execution time of the first i layers after the layer i is allocated to the node j . The state transition equation is formulated as

$$DP(i, j) = \min \left(DP(i-1, k) + t_{\text{comp}}^{i,j} + t_{\text{comm}}^{i-1,k,j} + \mathbf{1}(i) \cdot t_{\text{comm}}^{i,j,0} \right) \quad (7)$$

where $DP(i-1, k)$ indicates the minimal execution time of the first $i-1$ layers if layer $i-1$ is allocated to device k ,

and $\mathbf{1}(i)$ is an indicator function, where its value is 1 when $i = N-1$, or 0 otherwise. Equation (7) shows that $DP(i, j)$ is determined by traversing at all possible nodes of the previous layer and choosing the one that minimizes the execution time of the first i layers. Moreover, due to the autoregressive nature of LLM, the generated token needs to be sent back to the source node for next iteration of generation. Hence, for the final layer $N-1$, the communication time not only includes the data transmission time from the $N-2$ layer but also the transmission time to the source node $t_{\text{comm}}^{N-1,j,0}$. Additionally, we initialize $DP(0, 0)$ as shown in Equation (8), by considering the privacy constraint

$$DP(0, 0) = t_{\text{comp}}^{0,0}. \quad (8)$$

By traversing each layer and each node based on (7), we can fill in the dynamic programming table $DP(i, j)$ to track the minimum total execution time to reach each layer. Finally, the minimal total execution time at the final layer can be calculated by (9). We can then get the optimal node allocation for each layer by backtracking $DP(i, j)$

$$\min_{j=0, \dots, M-1} (DP(N-1, j)). \quad (9)$$

This method is simple and effective. With dynamic programming, we can quickly traverse the solution space and find the best LLM partition and allocation strategy. The algorithm to find the optimal LLM partition and allocation strategy for minimizing inference latency is shown in Algorithm 1.

In Algorithm 1, we first initialize the dynamic programming table $DP(i, j)$ and choice table $\text{choice}(i, j)$ (lines 1 and 2). We initialize $DP(0, 0)$ according to (8). The $\text{choice}(i, j)$ records the node k to host the $i-1$ layer. It is the optimal variable of (7). We then traverse the layers of the large language model from layer 1. For each layer, we traverse all the computing devices with sufficient memory and calculate the inference time (lines 3–19). After filling the DP table, we can find the minimal $DP(N-1, j)$, which represents the minimal time for executing the LLM model, and the final node to host layer $N-1$. Finally, by backtracing $\text{choice}(i, j)$, we get the model partition and allocation strategy R (lines 20–28). The computational complexity of Algorithm 1 is $O(N \times M \times M)$, where N is the number of layers of the LLM model and M is the number of devices in the network.

B. Optimize LLM Inference Throughput

Problem Formulation: For optimizing throughput, pipeline parallelism is adopted to avoid device idleness. As illustrated before, the computation time of layer i on node j is $t_{\text{comp}}^{i,j}$, and if layers $i-m$ are all allocated to node j , the computation time is indicated by $t_{\text{comp}}^{i \rightarrow m, j}$. The data transmission time of the activations of layer $i-1$ from node k to node j is $t_{\text{comm}}^{i-1,k,j}$. In pipeline parallel inference, the computation time and communication time can be overlapped to maximize the throughput. Thus, for the inference task, the maximum latency for the device j can be calculated as

$$T_{\text{latency}}^j = \max \left\{ \begin{matrix} t_{\text{comp}}^{i \rightarrow m, j} \\ t_{\text{comm}}^{i-1,k,j} \end{matrix} \right\}. \quad (10)$$

Algorithm 1: Joint Device Selection and LLM Partition for Optimizing Latency

Input: A LLM model; Computing device M ; Profiled traces; bandwidth $B_{k,j}$;
Output: the device selection and LLM partition strategy R

```

// initialization
1 Initialize DP table  $DP(i,j) \leftarrow INF$ , and choice table  $choice(i,j) \leftarrow NULL$  to record the strategy;
2 Enforce first layer to be allocated to node 0 by  $DP(0,0) \leftarrow t_{comp}^{0,0}$  and  $choice(0,0) \leftarrow 0$ ;
// fill in the DP table
3 for  $i \leftarrow 1$  to  $N-1$  do
4   for  $j \leftarrow 0$  to  $M-1$  do
5     if  $Mem_j \leq Req_i$  then
6       Continue;
7     end
8     else
9       for  $k \leftarrow 0$  to  $M-1$  do
10        Calculate the total execution time by Eq. (7) and assign it to  $t_{total}$ ;
11        if  $t_{total} \leq DP(i,j)$  then
12          Update  $DP(i,j)$  by assigning  $DP(i,j) \leftarrow t_{total}$ ;
13          Update memory  $Mem_j$ ;
14          Record allocation plan  $choice(i,j) \leftarrow k$ ;
15        end
16      end
17    end
18  end
19 end
// backtrace for allocation strategy
20 Initialize optimal strategy  $R$ ;
21 Find the last selected node  $N_{last} \leftarrow argmin_j(DP(N-1,j))$ ;
22 Add  $N_{last}$  to  $R$ ;
23 for  $i \leftarrow N-1$  to 0 do
24   Find the previous node  $N_{last} \leftarrow choice(i, N_{last})$ ;
25   Add  $N_{last}$  to  $R$ ;
26 end
27 Reverse  $R$ ;
28 return  $R$ ;

```

Ideally, for the selected devices, achieving the maximal throughput is equivalent to minimizing the latency of the slowest device. We use S to denote the selected devices, and then the problem of maximizing the inference throughput can thus be formulated as (11), where $j \in S$

$$\min_{X_{i,j}} \{T_{latency}^j | j \in S\}. \quad (11)$$

Solution: Similar to minimizing the inference latency, the problem of maximizing the throughput also has an optimal subproblem property. Maximizing the throughput of the first i layer can be deduced from solving the problem of allocating the first $i-1$ layer, which indicates that the optimal solution of

the whole problem can be constructed from the subproblems. We also use dynamic programming to solve the problem.

Let $g(i, S, k)$ denote the minimum time to process the first i layers with the set of used devices S , and the device k is the final node to be used, $k \in S$. We use $g(m, S', j)$ to denote the next state to process the first m layers with the set of used devices S' , and the device j is the final node to be used, where $0 \leq i < m \leq N-1$, $j \in M \setminus S$, $S' = S \cup \{j\}$.

The state transition equation is formulated in (12), where $g(m, S', j)$ is determined by the previous state $g(i, S, k)$, and the maximum latency of device j , i.e., the computation time $t_{comm}^{i-1,k,j}$ and the communication time $t_{comp}^{i \rightarrow m,j}$. The final optimal solution T_{thru}^{opt} is the minimum $g(N-1, S', j)$, where $S' \subseteq M$

$$g(m, S', j) = \min_{\substack{S' = S \cup \{j\} \\ 0 \leq i < m \leq N-1 \\ j \in M \setminus S}} \max \begin{cases} g(i, S, k) \\ t_{comm}^{i-1,k,j} \\ t_{comp}^{i \rightarrow m,j} \end{cases}. \quad (12)$$

Additionally, we have constraints when performing state transition. They are the memory constraint shown in (13), and privacy constraint

$$Req_{i \rightarrow m} \leq Mem_j \quad (13)$$

$$g(1, 1, 0) = t_{comp}^{0,0}. \quad (14)$$

Algorithm 2 describes the pseudocode to find the optimal solution T_{thru}^{opt} and the corresponding model partition and allocation strategy. In Algorithm 2, we first initialize the dynamic programming table $g(m, S', j)$ and choice table $choice(m, S, j)$, and assign $t_{comp}^{0,0}$ to $g(1, 1, 0)$ (lines 1 and 2). We then traverse the layers of the large language model from layer 1. For each layer, we traverse all the computing devices with sufficient memory and calculate the maximum latency (lines 3–23). After filling the DP table, we can find the maximum latency, based on which we then backtrace the choice table and finally get the model partition and allocation strategy (lines 24–32). The computational complexity of Algorithm 2 is $O(N^2 \times 2^M \times M^2)$, where N is the number of layers of the LLM model and M is the number devices in the network.

Pipeline Execution Optimization: Note that the above problem formulation and solution are based on the ideal case, where there is no idle device at any time. A device processes a batch of data and continues to handle another batch of data without waiting. However, it is impractical for LLM inference in real-world cases.

As shown in Fig. 5(a), different from those one-phase computation applications, the decoder-based LLM application has an autoregressive nature, where there will be multiple iterations and each iteration generates a token. The calculation of the current token relies on all the previous tokens. Hence, an iteration cannot start until the previous iteration ends and it gets the previously generated token. For pipeline parallel inference, this kind of scheme will lead to bubbles, representing idle status of devices. As shown in Fig. 5(a), suppose there are four micro-batches, $P1-P4$ represents the first iteration (prefill phase) of those micro-batches, respectively. G_{1A} and G_{1B} indicate the second and third iteration, respectively. They are also the first and second iteration in the generation phase. In this case, G_{1A} cannot start until finish of the first iteration,

Algorithm 2: Joint Device Selection and LLM Partition for Optimizing Throughput

Input: A LLM model; Computing devices M ; Profiled traces; bandwidth $B_{k,j}$;
Output: the device selection and LLM partition strategy R

```

// initialization
1 Initialize DP table  $g(i, S, k) \leftarrow INF$ , and choice table  $choice(m, S, j) \leftarrow NULL$  to record the strategy;
2 Enforce first layer to be allocated to node 0 by  $g(1, 1, 0) \leftarrow t_{comp}^{0,0}$  and  $choice(1, 1, 0) \leftarrow (0, 0, 0)$ ;
// fill in DP table
3 for  $i \leftarrow 1$  to  $N - 1$  do
4   for each subset  $S \subseteq M$  do
5     for last node  $k \in S$  do
6       for  $m \leftarrow i + 1$  to  $N - 1$  do
7         for  $j \in M \setminus S$  do
8           if  $Mem_j \leq \sum_i^m Req_i$  then
9             Continue;
10          end
11         else
12           Get  $S'$  by adding node  $j$  to the selected device set  $S$ ;
13           Calculate current maximum execution time  $T_{max}$  via Eq. (12) for the maximum execution time in all stages;
14         end
15         if  $T_{max} \leq g(i, S, k)$  then
16            $g(m, S', j) \leftarrow T_{max}$ ;
17           Record the current strategy  $choice(m, S', j) \leftarrow (i, j, k)$ ;
18         end
19       end
20     end
21   end
22 end
// backtrack for optimal allocation
24 Initialize optimal strategy  $R$ ;
25 Find selected device set  $S$  and the last selected node  $N_{last}$  by  $S, N_{last} \leftarrow argmin_{S,k}(g(N - 1, S, k))$ ;
26 Initialize  $layer \leftarrow N - 1$ ;
27 while  $layer > 0$  do
28    $(i, j, k) \leftarrow choice(layer, S, N_{last})$ ;
29   Add  $(i \rightarrow layer, j)$  to  $R$ ;
30   Update  $layer, S$  and  $N_{last}$ ;
31 end
32 return  $R$ ;

```

i.e., the ends of $P4$, which leads to bubbles, where devices 1–3 are in idle state.

To approximate the ideal case and enhance the resource utilization for improving throughput, we tend to reduce the bubbles in the pipeline execution. We propose EdgeShard-No-Bubbles, which optimizes the token generation order

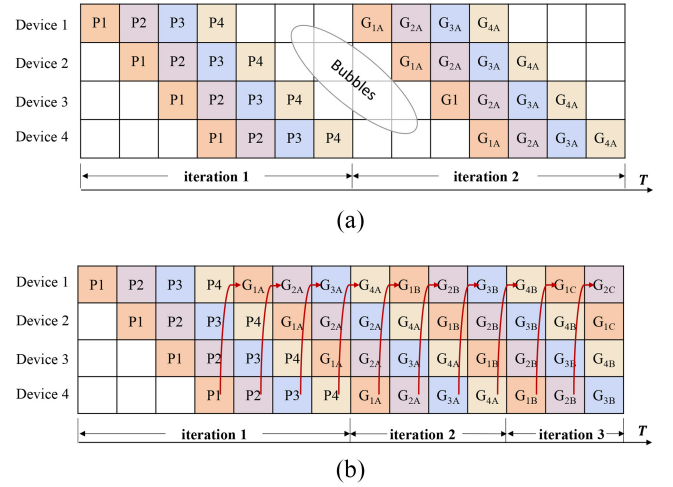


Fig. 5. Different pipeline execution strategies of EdgeShard. EdgeShard-No-Bubbles reduces device idle time to improve throughput by allowing immediate token generation of a micro-batch without waiting for other micro-batches. (a) EdgeShard-Bubbles. (b) EdgeShard-No-Bubbles.

TABLE III
SPECIFICATIONS OF HETEROGENEOUS PHYSICAL DEVICES

Category	Device	Memory	AI Performance
Edge Device	Jetson AGX Orin	32GB	3.33 TFLOPS
Edge Device	Jetson Orin NX	16GB	1.88 TFLOPS
Cloud Server	RTX 3090	24GB	36 TFLOPS

across micro batches, allowing for immediate token generation without waiting for the ending of all micro-batches in an iteration. As shown in Fig. 5(b), after the prefill stage $P1$ ends of the first batch, device-1 immediately executes the token generation of the first micro-batch as indicated by G_{1A} . Similarly, when G_{1A} ends, device-1 goes to the next iteration of token generation of the first micro-batch, indicated by G_{1B} . Same operations is applied for $P2$ – $P4$. Compared to EdgeShard-Bubbles, EdgeShard-No-Bubbles reduces bubbles by mitigating device idle time and is expected to improve throughput. From the pipeline execution graph in Fig. 5, we can see that EdgeShard-No-Bubbles generates more tokens at the same time.

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

Testbed: We use various edge devices and cloud servers to act as the heterogeneous computation devices in CEC. The specifications of those devices are listed in Table III. We use 15 devices, including 12 Jetson AGX Orin, two Jetson Orin NX, and one cloud server to configure the collaborative edge network. The physical testbed is shown in Fig. 6. Those devices are connected with a route and a switch. The bandwidth between any two devices is 1000 Mb/s. We use the Linux TC tool [24] to vary network bandwidth and communication latency between devices.

Benchmarks: We test the performance of EdgeShard with a series of Llama2 models [2], including Llama2-7B, Llama2-13B, and Llama2-70B. Llama2 is released by Meta in July

TABLE IV
PERFORMANCE OF LLM INFERENCE (AVERAGE LATENCY: MILLISECONDS/TOKEN; THROUGHPUT: TOKENS/SECOND)

	Llama2-7B		Llama2-13B		Llama2-70B	
	Latency	Throughput	Latency	Throughput	Latency	Throughput
Edge-Solo	140.34	24.36	OOM	OOM	OOM	OOM
Cloud-Edge-Even	227.35	7.56	319.44	4.68	OOM	OOM
Cloud-Edge-Opt	140.34	24.36	243.45	4.74	OOM	OOM
EdgeShard	75.88	52.45	173.43	10.45	3086.43	1.25



Fig. 6. Our testbed has heterogeneous edge devices and cloud server. Their specifications are shown in Table III.

2023 and is one of the most popular and powerful open-source LLMs, representing a groundbreaking leap in the field of AI and natural language processing. For the model inference, we adopt the text generation task to test the performance. We use the WikiText-2 dataset [25] from HuggingFace. We extract a subset of samples with the length of input tokens as 32 and generate 96 tokens. We use full-precision inference in all the following experiments.

Baselines: We compare the performance in terms of latency and throughput of EdgeShard with various baselines. (We do not use the cloud-only as a baseline because it requires the input token to be transmitted to the cloud server, which may lead to privacy concerns).

- 1) *Edge-Solo*: In this case, the LLMs are deployed locally on an edge device without model partition.
- 2) *Cloud-Edge-Even*. In this case, the LLMs are evenly partitioned into two parts. One is allocated to the edge device, and another is allocated to the cloud server.
- 3) *Cloud-Edge-Opt*: In this case, the LLMs are partitioned into two shards. One is allocated to the edge device, and another is allocated to the cloud server. For the partition strategy of LLMs, we also use the proposed dynamic programming algorithms. The difference is that there is only two devices as the algorithm input.

B. Overall Evaluation

We set AGX Orin as the source node and the bandwidth between the source node and the cloud server as 1 Mb/s. The bandwidth between other computing devices is set to be 50 Mb/s with a variance of 20%. To test the throughput, we set the batch size as the maximum batch size that the participating devices can support. The latency and throughput of LLM inference are shown in Table IV.

We have the following observations. First, EdgeShard is potential and beneficial for large language model deployment. For the Llama2-70B model, the memory requirement is about 280 GB, which far exceeds the memory capacity of solo edge deployment and cloud-edge collaborative deployment. They will have the out-of-memory issue (OOM). However, EdgeShard tackles this challenge by splitting the large model into shards and allocating them to multiple devices, enabling collaborative model inference. Second, EdgeShard achieves obviously lower inference latency and higher inference throughput than baseline methods. For the Llama2-7B model, EdgeShard achieves 75.88 ms latency, which is about $1.85\times$ faster than the Edge-Solo and Cloud-Edge-Opt, and about $3\times$ faster than the Cloud-Edge-Even. For the inference throughput, EdgeShard achieves 52.45 tokens per second with a maximum batch size of 8, which is around $2.2\times$ larger than the Edge-Solo and Cloud-Edge-Opt, and about $7\times$ larger than the Cloud-Edge-Even. Similar performance improvement is also observed for the Llama2-13B model, where EdgeShard achieves 45.7% and 28.8% lower latency than the Cloud-Edge-Even and Cloud-Edge-Opt, respectively. Also, EdgeShard has $2.23\times$ and $2.2\times$ higher throughput than the Cloud-Edge-Even and Cloud-Edge-Opt. Third, we can also see that, for Llama2-7B, the Cloud-Edge-Opt tends to have the same performance in terms of both inference latency and throughput as the Edge-Solo. This is because the bandwidth between the source node and the cloud server is very limited in this experimental setting, i.e., 1 Mb/s. The optimal deployment strategy of the cloud-edge-collaboration is local execution, which is the same as the Edge-Solo.

C. Effects of Bandwidth

We set the source node as AGX Orin and vary the bandwidth between the cloud server and the source node from 1 to 50 Mb/s. The performance of the latency and throughput of LLM inference are shown in Figs. 7 and 8, respectively.

For Llama2-13B, a single AGX Orin cannot accommodate the full model. We only compare the performance among Cloud-Edge-Even, Cloud-Edge-Opt, and EdgeShard. Similarly, due to the memory constraint, the three baseline methods are not able to deploy the Llama2-70B model. Instead, we compare the performance of EdgeShard with its variant, i.e., EdgeShard-Even, where the model is equally partitioned and deployed to all the participating computing devices. It selects 11 AGX Orin and one RTX 3090 to deploy the 70B model.

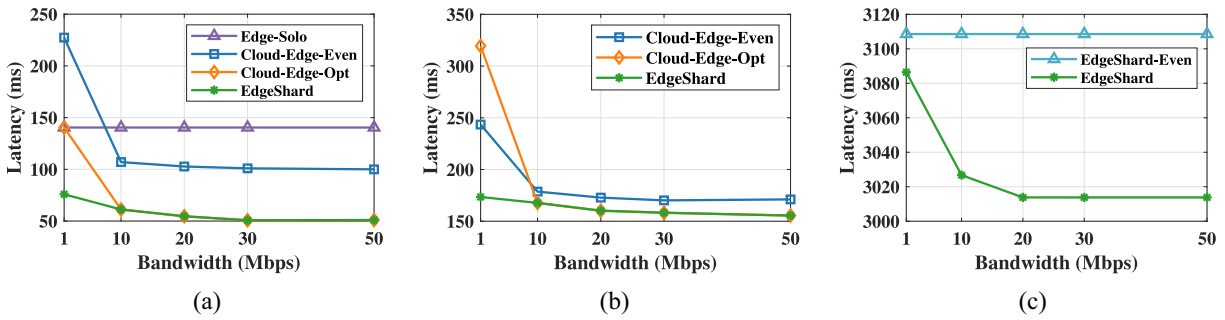


Fig. 7. Impact of network bandwidth to latency of collaborative LLMs inference. (a) Llama2-7B. (b) Llama2-13B. (c) Llama2-70B.

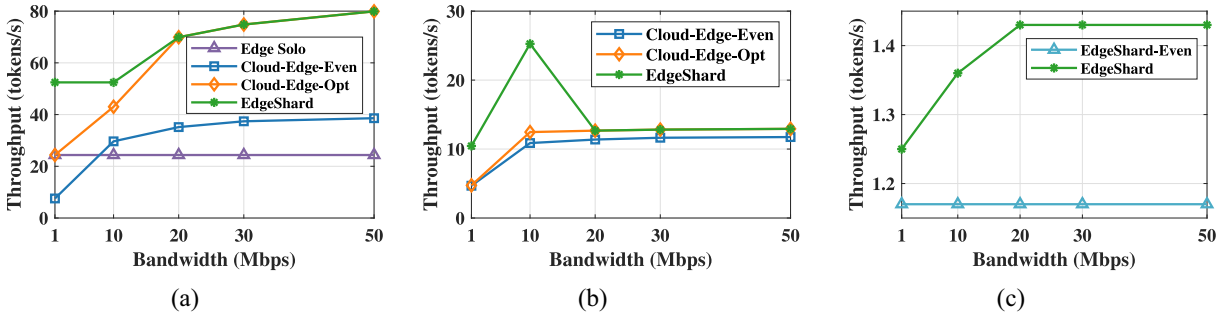


Fig. 8. Impact of bandwidth to throughput of collaborative LLMs inference. (a) Llama2-7B. (b) Llama2-13B. (c) Llama2-70B.

In terms of latency, except for the Edge-Solo, the latency of the other three methods decreases with the increasing bandwidth. This is because the three methods are collaboration-based, and the latency is influenced by the data transmission time. The increasing bandwidth leads to reduced communication time. We can also see that for the collaboration methods, there is a dramatically latency reduction when the cloud-source bandwidth changes from 1 to 10 Mb/s and a minor variance from 10 to 50 Mb/s. This is because the bandwidth is gradually saturated at that time, and the computation time becomes the bottleneck.

Moreover, we can see that when the bandwidth is greater than 10 Mb/s, the cloud-edge collaboration methods outperform the Edge-Solo method, as the cloud-edge collaboration methods introduce the powerful cloud server for computation acceleration. However, when the bandwidth is 1 Mb/s, the Cloud-Edge-Even performs worse than EdgeSolo. This is because the data transmission cost is high in this case. The Cloud-Edge-Opt method tends to deploy the LLM model locally, which is the same as the Edge-Solo method. Interestingly, the latency of the Cloud-Edge-Opt and EdgeShard is nearly the same when the bandwidth is greater than 10 Mb/s. We found that EdgeShard generates the same model partition and allocation policies as the Cloud-Edge-Opt method. The variance comes from the small fluctuations in model execution. It shows that the performance of EdgeShard will not be worse than that of Cloud-Edge-Opt, and the Cloud-Edge-Opt method is a special case of EdgeShard. A similar pattern is also observed for Llama2-13B. For Llama2-70B, EdgeShard performs better than its variant EdgeShard-Even, as there is resource heterogeneity among the cloud server and edge devices, and EdgeShard adaptively

partitions the LLMs among computing devices. However, the performance improvement is not so obvious as there are 11 AGX with the same capability and only one RTX 3090.

In terms of throughput, similar patterns are also found for the Llama2-7B model. Differently and interestingly, for Llama2-13B, EdgeShard does not show a closing performance with the Cloud-Edge-Opt method when the bandwidth is 10 Mb/s, but with a great improvement about $2\times$ higher throughput than the Cloud-Edge-Opt method. This is because of the high memory consumption of the RTX 3090 and the source node, i.e., AGX Orin. We observed that for the Cloud-Edge-Opt, the memory consumption of the two devices goes up to 95% and 98%, respectively, which only allows for a maximum batch size of 4. Otherwise, there will not be enough memory to hold the KV cache. However, when the bandwidth is 10 Mb/s, EdgeShard involves several edge devices where the memory consumption of an individual device becomes dramatically decreased, allowing for a larger batch size, i.e., 8 in this case. When the bandwidth is higher than 10 Mb/s, EdgeShards tends to have the same model partition and allocation strategy as the Cloud-Edge-Opt, which yields a closing performance, as shown in Llama2-7B. For Llama2-70B, there is a slight throughput improvement of EdgeShard, and EdgeShard-Even shows a steady throughput as the evenly partition strategy will not change with the cloud-source bandwidth.

D. Effects of Source Node

We also test the influence of the source node on the inference latency and throughput, as the source node may have different computation and memory capacities, and EdgeShard

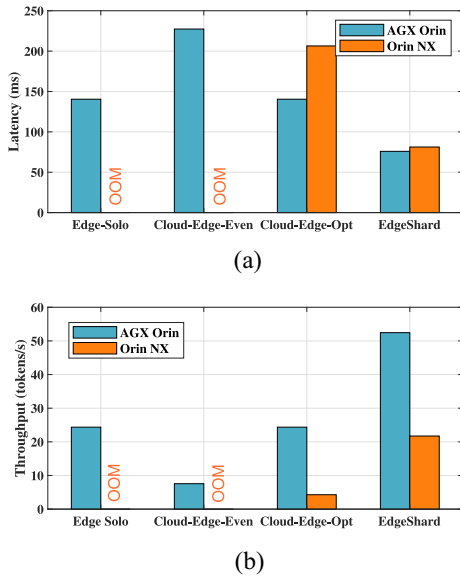


Fig. 9. Impact of source node. (a) Llama2-7B-latency. (b) Llama2-7B-throughput.

enforces the first layer of LLM models residing on the source node to avoid raw data transmission. We set the source node as AGX Orin and Orin NX, respectively, and compare their performance. We set the bandwidth between the source node and the cloud server as 1 Mb/s. The results of Llama2-7B inference are shown in Fig. 9.

We find that when the source node is Orin NX, the Edge-Solo and Cloud-Edge-Even methods encounter the OOM error. This is due to the relatively lower memory of Orin NX, which cannot accommodate the Llama2-7B model, even for half part of the model. The difference between the two cases under the Cloud-Edge-Opt method is much more obvious than that of EdgeShard. For Cloud-Edge-Opt, there is about a 60 ms gap, and for EdgeShard, the gap is about 5 ms. This is because there are only two devices in the Cloud-Edge-Opt case, and it tends to put more layers on the source node. However, AGX Orin is much more powerful than Orin NX in terms of computation capacity. EdgeShard tends to involve more devices and put fewer model layers on the source node, which can fill in the gap in computation capacity between the source nodes. A similar phenomenon is also observed for the throughput, where AGX Orin has 6 \times higher throughput than Orin NX for the Cloud-Edge-Opt method and only 2 \times higher throughput under the EdgeShard method. It shows that EdgeShard can make full use of the computation resources in the network.

E. Effects of Profiling Strategy

The first step of Edgeshard is to do the profiling, which requires each device to run the LLM and record the execution time. However, for LLM, the execution time changes significantly in the prefill and generation phase. In this work, we take the average of prefill and generation phase as the stable value in the profiling step. We also study the performance of EdgeShard under different profiling strategies.

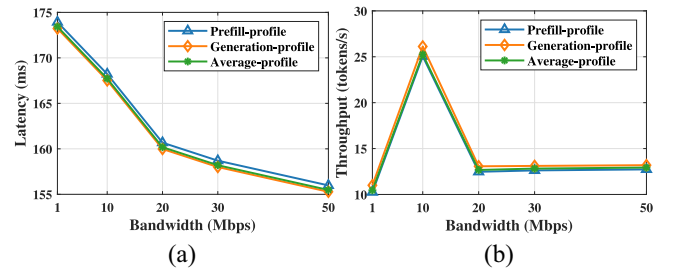


Fig. 10. Performance of LLama-2-13B under different profiling methods. (a) LLama2-13B latency. (b) LLama2-13B throughput.

We take the LLama2-13B as an example. The performance of EdgeShard for LLama2-13B inference under different profiling methods are shown in Fig. 10(a) and (b), where prefill-profile, generation-profile, and average-Profile represent taking the execution time in prefill phase, generation phase, and average as stable value, respectively. We find that there are minor difference and small fluctuations among the performance of the three methods. This is because through there are 10 \times gap between the prefill and generation phase, the relative computing capabilities of different computing devices keep unchanged, i.e., the inference time of RTX3090 (most powerful) in prefill and generation phase are both much more lower than that of AGX Orin (the weakest). The heterogeneity of computing capability has a great impact on the device selection and model partition strategy.

F. Effects of Pipeline Execution Strategy

We evaluate the two pipeline execution strategies. We set the bandwidth between the cloud server and the source node as 1 Mb/s. The results are shown in Fig. 11.

We can see that for all methods, EdgeShard-No-Bubble outperforms EdgeShard-Bubble. Specifically, for Llama2-7b, EdgeShard-No-Bubble achieves an improvement of about 0.36 and 6.96 tokens per second than Edgeshard-Bubble for Cloud-Edge-Even and EdgeShard, respectively. For the Cloud-Edge-Opt method, it selects local execution in this case. There is no pipeline execution, so the throughput for the two methods is the same. For Llama2-13b, EdgeShard-No-Bubble achieves an improvement of about 1.69, 1.89, and 5.21 tokens per second than Edgeshard-Bubble for the Cloud-Edge-Even, Cloud-Edge, and EdgeShard, respectively. Compared to EdgeShard-Bubble, EdgeShard-No-bubble does not need to wait for the completion of all micro-batches in an iteration and can effectively reduce the devices' idle time, thus leading to a higher throughput.

VI. RELATED WORK

A. Model Partition for Distributed Model Inference at Edge

Partitioning the resource-greedy model and accommodating multiple devices is a major solution to address the resource-greedy issue of large AI models. Moreover, leveraging the heterogeneous resources of multiple devices could achieve inference acceleration without sacrificing accuracy. Several studies [26], [27], [28], [29] explore the distribution of DNN

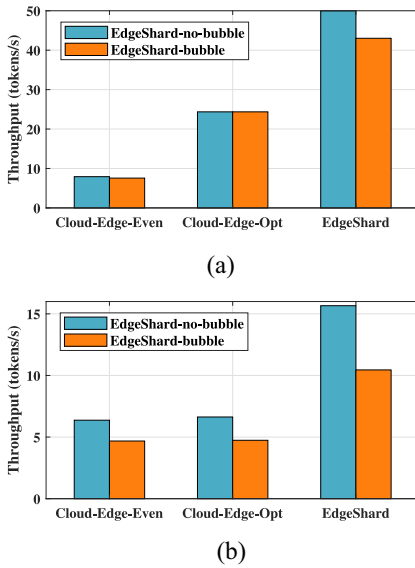


Fig. 11. Impact of pipeline execution strategy. (a) Llama2-7B-throughput. (b) Llama2-13B-throughput.

model inference by splitting tasks between the edge devices and the cloud servers. This approach helps reduce both latency and the computational load on the edge devices, but may suffer from limited bandwidth and unpredictable delays between the edge and the cloud. Some works try to leverage edge-edge collaboration to address the issue by distributing the workload to several edge devices. DeepThings [30] proposes a method that divides CNN model computations into finer tasks for parallel inference across the edge devices by partitioning the feature maps. DeepHome [20] distributes inference tasks across various heterogeneous devices within the home environment. The follow-up works like DeepSlicing [21] and [22] further optimize the network and device heterogeneity for performance improvement. However, those works only consider the model allocation problem, which studies how to partition the model and allocate the model shards to preset devices. EdgeShard goes further, performing joint device selection and model partitioning. Moreover, they are particularly designed for CNN models and not applicable for transformer-based LLMs.

B. Edge Computing for Efficient LLM Deployment

LLM is computation-intensive and memory consuming. To address the issue of memory wall, quantization is widely adopted [7], [8], [9], [10], [11], [12]. GPTQ [8] quantizes LLM with hundreds of billions of parameters to 3–4 bits based on approximate second-order information. Lin et al. [10] reduce quantization error by optimizing channel scaling to preserve the salient important weights, which uses weight-only quantization. SmoothQuant [11] and Agile-Quant [7] take a further step, which quantizes not only the model weights but also the model activations. However, the computation capacity and memory of a single device are still limited even for quantized LLM. Moreover, the performance of quantized LLM usually cannot be compared to that of its full-size model.

Other works [13], [14] tend to leverage the cloud-edge collaboration to partition and distribute the massive computation workload of LLM inference and finetuning. Wang et al. [13] increase the throughput by distributing the computation between the cloud servers and the edge devices, and reducing the communication overhead of transmitting the activations between the central cloud and edge devices by leveraging the low-rank property of residual activations. Chen et al. [14] efficiently leveraged location-based information of edge devices for personalized prompt completion during collaborative edge-cloud LLM serving. Edge-LLM [31] also proposed a cloud-edge collaborative network which implements an adaptive quantization strategy, caching feature maps and adopting the value density first scheduling algorithm, resulting in reduced GPU overhead and accelerated LLM computation. However, the latency between the edge devices and the central cloud is usually high and unstable, which will affect the inference and fine tuning performance of LLM.

EdgeShard is different from those works. We propose a general framework to integrate the computation resources of heterogeneous and ubiquitous cloud servers and edge devices. The framework allows the adaptive selection of computation devices and partitions of the computation workload of LLM inference for optimized latency and throughput.

C. LLM for Optimizing Edge Computing

LLMs also have great potential in making complex and coherent decisions. There are also some works that leverage LLM to optimize resource utilization in edge computing, such as resource allocation and task offloading, network management, and intelligent IoT control. Dong et al. [32] proposed LAMBO, an LLM-based task offloading framework for mobile edge computing, to address the challenging issues of heterogeneous constraints, partial status perception, diverse optimization objectives, and dynamic environment that are not well addressed in traditional task offloading research. LAMBO shows that LLM is more effective compared to traditional DNN and deep reinforcement learning-based methods in complex and dynamic edge computing environments. They further design an LLM-based multiagent system and incorporate communication knowledge and tools into the system, empowering it with the ability to optimize semantic communication in a 6G network [33]. Apart from optimization of resource utilization, Shen et al. [34] leveraged the outstanding abilities of GPT in language understanding and code generation to train new models among federated edge devices. Rong and Rutagemwa [35] leveraged LLMs to generate adaptive control algorithms for addressing the diverse, dynamic, and decentralized network conditions in 6G integrated terrestrial network (TN) and nonterrestrial network (NTN). Though LLMs have shown great potential in making intelligent decisions, especially in complex and dynamic edge computing systems, the related research is still in the early stages. Challenges, such as significant resource consumption, latency of decision making, and uncertainty of generated decisions need further study.

VII. DISCUSSION AND FUTURE WORK

This section discusses some open issues and future works that may appeal to readers.

Where to Use EdgeShard: EdgeShard aligns well with technology trend and can be applied to various application scenarios. Regarding technology trend, computing power network [36] has been put high agenda on many governments' technology innovation plan, which envisions integrating distributed computing resources for jointly performing computational tasks. EdgeShard provides a framework and intelligent task scheduling algorithms for LLM inference over geo-distributed devices. Regarding practical usage, decentralized physical infrastructure network (DePIN) [37] attracts great attentions recent days from both academic and industry. It advocates a network, where everyone can contribute their resources (e.g., GPU, storage, and power) and get rewards back. For example, Yandex [38] performs collaborative LLM inference over distributed consumers' GPUs. But they do not study how to intelligently select devices and partition LLM over heterogeneous resources. Moreover, in smart home area, there are several edge devices, such as TVs, phones, and smart speakers. Those devices can collaborate to perform LLM inference without sending user personal data to cloud.

Incentive Mechanisms: In this work, we partition the LLM into multiple shards and allocate them to heterogeneous devices. For edge computing scenarios, such as smart home and smart factory, there is a set of trusted devices owned by a single stakeholder. They may be able to use those devices for collaborative inference. However, if the devices belong to different stakeholders, they may not be willing to share devices' computation resources. Further incentive mechanisms are needed to reward resource sharing.

Batch Size-Aware Optimization: Large batch size will increase memory usage and affect the inference throughput. As shown in the experiments, by partitioning the workload of LLM inference to multiple devices, the memory usage of participating devices can be reduced and thus allows for a larger batch size, leading to increased throughput. However, the designed dynamic programming algorithm does not consider this factor, remaining space for further optimization.

Security Issue: In EdgeShard, the activations of the partition layer will be transmitted across collaborative devices and may contain sensitive information. An honest but curious device may infer features of raw data through the shared activations, leading to data leakage risk. Researchers and commercial vendors have proposed various approaches to address this challenge, such as differential privacy and confidential computing. The former has been extensively studied in federated learning [39]. Confidential computing [40] is becoming popular and widely adopted by cloud platforms. It leverages trusted execution environments (TEE) to create an isolated runtime environment that provides security and integrity protection by using specialized secure hardware. A thorough discussion of the threat model and privacy-preserving solutions may be out of the scope of this article.

VIII. CONCLUSION

In this work, we proposed EdgeShard to enable the efficient deployment and distributed inference of LLM on collaborative edge devices and cloud servers. We formulated a joint device selection and model partition problem to optimize inference latency and throughput, respectively, and solve it using dynamic programming algorithms. Experimental results show that EdgeShard can adaptively determine the LLM partition and deployment strategy under various heterogeneous network conditions for optimizing inference performance. EdgeShard is not designed to replace cloud-based LLM inference, but to provide a flexible and adaptive LLM serving methods by utilizing ubiquitous computing devices. Experiments also shows that EdgeShard outperforms the cloud-edge collaborative inference method when the cloud bandwidth is insufficient and tends to yield the same deployment strategy as the cloud-edge collaborative inference method when facing relatively abundant cloud bandwidth.

This is a pioneering work of deploying LLM in CEC environment. We hope this work can stimulate more ideas and further research in this promising area.

REFERENCES

- [1] J. Achiam et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.
- [2] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, *arXiv:2307.09288*.
- [3] R. Anil et al., "PaLM 2 technical report," 2023, *arXiv:2305.10403*.
- [4] A. Vaswani et al., "Attention is all you need," in *Proc. 31st Conf. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.
- [5] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
- [6] X. Chen, J. Cao, Y. Sahni, M. Zhang, Z. Liang, and L. Yang, "Mobility-aware dependent task offloading in edge computing: A digital twin-assisted reinforcement learning approach," early access, Nov. 25, 2024, doi: [10.1109/TMC.2024.3506221](https://doi.org/10.1109/TMC.2024.3506221).
- [7] X. Shen et al., "Agile-quant: Activation-guided quantization for faster inference of LLMs on the edge," 2023, *arXiv:2312.05693*.
- [8] E. Frantar, S. Ashkboos, T. Hoeffer, and D. Alistarh, "GPTQ: Accurate post-training quantization for generative pretrained transformers," 2022, *arXiv:2210.17323*.
- [9] E. Frantar, S. Ashkboos, T. Hoeffer, and D. Alistarh, "OPTQ: Accurate quantization for generative pretrained transformers," in *Proc. 11th Int. Conf. Learn. Represent.*, 2022, pp. 1–16.
- [10] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "AWQ: Activation-aware weight quantization for LLM compression and acceleration," 2023, *arXiv:2306.00978*.
- [11] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in *Proc. Int. Conf. Mach. Learn.*, 2023, pp. 38087–38099.
- [12] X. Shen et al., "EdgeQAT: Entropy and distribution guided quantization-aware training for the acceleration of lightweight LLMs on the edge," 2024, *arXiv:2402.10787*.
- [13] Y. Wang, Y. Lin, X. Zeng, and G. Zhang, "PrivateLoRA for efficient privacy preserving LLM," 2023, *arXiv:2311.14030*.
- [14] Y. Chen et al., "NetGPT: A native-AI network architecture beyond provisioning personalized generative services," 2023, *arXiv:2307.06148*.
- [15] M. Zhang, J. Cao, Y. Sahni, Q. Chen, S. Jiang, and T. Wu, "EaaS: A service-oriented edge computing framework toward distributed intelligence," in *Proc. IEEE Int. Conf. Service-Orient. Syst. Eng. (SOSE)*, 2022, pp. 165–175.
- [16] M. Zhang, J. Cao, L. Yang, L. Zhang, Y. Sahni, and S. Jiang, "ENTS: An edge-native task scheduling system for collaborative edge computing," in *Proc. IEEE/ACM 7th Symp. Edge Comput. (SEC)*, 2022, pp. 149–161.

- [17] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. 33rd Conf. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 103–112.
- [18] D. Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Oper. Syst. Princ.*, 2019, pp. 1–15.
- [19] P. Patel et al., "Splitwise: Efficient generative LLM inference using phase splitting," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit. (ISCA)*, 2024, pp. 118–132.
- [20] Z. Hu, A. B. Tarakji, V. Raheja, C. Phillips, T. Wang, and I. Mohamed, "DeepHome: Distributed inference with heterogeneous devices in the edge," in *Proc. 3rd Int. Workshop Deep Learn. Mobile Syst. Appl.*, 2019, pp. 13–18.
- [21] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "DeepSlicing: Collaborative and adaptive CNN inference with low latency," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2175–2187, Sep. 2021.
- [22] C. Hu and B. Li, "Distributed inference with deep learning models across heterogeneous edge devices," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2022, pp. 330–339.
- [23] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pretraining of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.
- [24] B. Hubert et al., "Linux advanced routing & traffic control HOWTO," *Netherlabs BV*, vol. 1, pp. 99–107, Dec. 2002.
- [25] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," in *Proc. Int. Conf. Learn. Represent.*, 2017, pp. 1–15.
- [26] Y. Kang et al., "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 1, pp. 615–629, 2017.
- [27] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Trans. Mob. Comput.*, vol. 20, no. 2, pp. 565–576, Feb. 2021.
- [28] P. Ren, X. Qiao, Y. Huang, L. Liu, C. Pu, and S. Dustdar, "Fine-grained elastic partitioning for distributed DNN toward mobile Web AR services in the 5G era," *IEEE Trans. Serv. Comput.*, vol. 15, no. 6, pp. 3260–3274, Nov/Dec. 2022.
- [29] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. INFOCOM IEEE Conf. Comput. Commun.*, 2019, pp. 1423–1431.
- [30] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.
- [31] F. Cai, D. Yuan, Z. Yang, and L. Cui, "Edge-LLM: A collaborative framework for large language model serving in edge computing," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2024, pp. 799–809.
- [32] L. Dong et al., "LAMBO: Large AI model empowered edge intelligence," 2023, *arXiv:2308.15078*.
- [33] F. Jiang et al., "Large language model enhanced multiagent systems for 6G communications," 2023, *arXiv:2312.07850*.
- [34] Y. Shen et al., "Large language models empowered autonomous edge AI for connected intelligence," *IEEE Commun. Mag.*, vol. 62, no. 10, pp. 140–146, Oct. 2024.
- [35] B. Rong and H. Rutagemwa, "Leveraging large language models for intelligent control of 6G integrated TN-NTN with IoT service," *IEEE Netw.*, vol. 38, no. 4, pp. 136–142, Jul. 2024.
- [36] X. Tang et al., "Computing power network: The architecture of convergence of computing and networking toward 6G requirement," *China Commun.*, vol. 18, no. 2, pp. 175–185, Feb. 2021.
- [37] Z. Lin, T. Wang, L. Shi, S. Zhang, and B. Cao, "Decentralized physical infrastructure network (DePIN): Challenges and opportunities," 2024, *arXiv:2406.02239*.
- [38] A. Borzunov et al., "Petals: Collaborative inference and fine-tuning of large models," 2022, *arXiv:2209.01188*.
- [39] A. El Ouadrhiri and A. Abdelhadi, "Differential privacy for deep and federated learning: A survey," *IEEE Access*, vol. 10, pp. 22359–22380, 2022.
- [40] D. P. Mulligan, G. Petri, N. Spinale, G. Stockwell, and H. J. Vincent, "Confidential computing—A brave new world," in *Proc. Int. Symp. Secure Private Execut. Environ. Design (SEED)*, 2021, pp. 132–138.



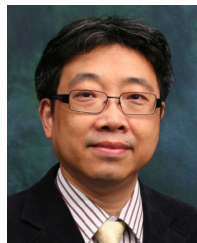
Mingjin Zhang received the B.Eng. degree in communication engineering from Wuhan University of Technology, Wuhan, China, in 2019. He is currently pursuing the Ph.D. degree with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, SAR, China.

He has been a Visiting Ph.D. Student with the Department of Computer Science and Technology, University of Cambridge, Cambridge, U.K., from February 2023 to September 2023. His research interests include edge computing, edge AI, and distributed machine learning.



Xiaoming Shen received the B.Eng. degree in Internet of Things engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2022, and the M.Sc. degree from the University of Hong Kong, Hong Kong, in 2023.

He is a Research Assistant with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, SAR, China. His research interests include edge computing, deep learning, and AIoT.



Jiannong Cao (Fellow, IEEE) received the B.Sc. degree from Nanjing University, Nanjing, China, and the M.Sc. and Ph.D. from Washington State University, Pullman, WA, USA.

He is currently the Otto Poon Charitable Foundation Professor of Data Science and the Chair Professor of Distributed and Mobile Computing with the Department of Computing, Hong Kong Polytechnic University (PolyU), Hong Kong, where he is also the Dean of the Graduate School, the Director of the Research Institute for Artificial

Intelligence of Things, and the Director of the Internet and Mobile Computing Laboratory. He was the Founding Director and currently an Associate Director of Research Facility in Big Data Analytics, PolyU. He served as the Department Head from 2011 to 2017. His research interests include distributed systems and blockchain, wireless sensing and networking, big data and machine learning, and mobile cloud and edge computing.

Prof. Cao is a Member of Academia Europaea, a Fellow of the Hong Kong Academy of Engineering Science and the China Computer Federation, and an ACM Distinguished Member.



Zeyang Cui received the B.Eng. degree in computer science and technology from Lanzhou University, Lanzhou, China, in 2022, and the M.Sc. degree from the University of Hong Kong, Hong Kong, in 2023.

He is currently a Research Assistant with the Department of Computing, Hong Kong Polytechnic University, Hong Kong, SAR, China. His research interests include edge computing, deep learning, graph representation learning, bioinformatics, AIoT, and metaverse.



Shan Jiang received the B.Sc. degree in computer science and technology from Sun Yat-sen University, Guangzhou, China, in 2015, and the Ph.D. degree in computer science from Hong Kong Polytechnic University, Hong Kong, SAR, China, in 2021.

He is currently a Research Assistant Professor with the Department of Computing, Hong Kong Polytechnic University. Before that, he visited Imperial College London, London, U.K., from November 2018 to March 2019. His research interests include distributed systems and blockchain,

edge computing and mobile blockchain, and blockchain-based Web3 data infrastructure.