

FluidEdge: Expediting Serverless Machine Learning Inference via Bottleneck-Aware Auto-Scaling on Edge SoCs

Borui Li¹, Member, IEEE, Tiange Xia, Weilong Wang², Jingyuan Zhang, Shuai Wang³, Chenhong Cao⁴, Zheng Dong⁵, Member, IEEE, and Shuai Wang⁶, Member, IEEE

Abstract—Mobile applications based on machine learning (ML) are increasingly relying on offloading to the edge devices for low-latency, resource-efficient computation. Applying serverless computing for these ML applications on the edge offers a promising solution for handling dynamic workloads while meeting user-specified latency service-level objectives (SLOs). However, existing serverless frameworks, with their coarse-grained data parallelism and rigid model partitioning, are inadequate for ML inference on widely adopted edge System-on-Chip (SoC) devices. This paper presents FluidEdge, an edge-native serverless inference framework. FluidEdge identifies bottleneck operators in ML models and addresses them through a novel fine-grained intra-function latency-sensitive auto-scaling approach that dynamically scales inference bottlenecks during online serving. Additionally, it employs inter-function scaling to further prevent latency SLO violations and leverages the unified memory of edge SoCs for efficient data sharing during inference. Experimental results demonstrate that FluidEdge achieves a 37.4% latency improvement and 67.3% - 87.6% SLO violation reduction compared to best-performed state-of-the-art serverless inference frameworks.

Index Terms—Serverless, edge intelligence, service-level objective (SLO).

I. INTRODUCTION

NOWADAYS, offloading machine learning (ML) based mobile applications [1], [2] to edge computing devices is a feasible and widely-used way to comply with the latency service-level objectives (SLOs) [3]. For example, 90% of the inference requests are supposed to be completed within 200 ms

to satisfy the interactive mobile environments. Violating latency SLOs may lead to poor user experience or financial loss [4].

To facilitate the simultaneous support of multiple ML applications from different mobile users on the edge, especially under the highly dynamic request patterns of mobile applications, the emerging *serverless edge computing* is preferred [5], [6], [7]. Serving ML applications with a serverless paradigm means running model inference in isolated *containers* called *functions*, which creates a multi-tenant environment for serving multiple mobile users simultaneously. Following the ingress requests and latency SLOs, the serverless platform manages to *scale out* or *scale in* the function by replicating or reducing its serving containers.

Building upon the serverless computing paradigm, existing works further optimize the ML inference performance mainly in two ways: data parallelism [3], [4], [8] and model parallelism [9], [10], [11]. Serverless platforms naturally support data parallelism through auto-scaling. Hence, existing data parallelism optimization approaches [3], [4], [8] focus on prudently allocating the requests to different functions with optimized batch size when the serverless functions are scaled out. Model parallelism, which is also known as pipeline parallelism, partitions large ML models across multiple functions to overcome the resource constraints imposed by individual serverless functions for resource saving [9], [10] or cost reduction [11].

Although these serverless inference frameworks have exhibited great performance gains in the cloud, these works do not cater to the *unique characteristics of edge devices*. Given their power and size constraints, a noticeable trend in the hardware design of edge devices is the integration of heterogeneous yet resource-constrained accelerators, e.g., CPU and GPU, on a single System on Chip (SoC). Among the accelerators, a hardware unified memory is used for direct data sharing between them [12], [13], [14], [15]. These two characteristics cause the current serverless frameworks to underperform on edge SoCs for the following two reasons.

Coarse-grained scaling of data parallelism: Existing model parallelism approaches directly scale out the serverless function containing the *whole model* to serve additional requests. However, opposite to the “infinite capacity” of the cloud, edge devices operate within heterogeneous yet constrained resources. Moreover, our preliminary experiments in Section II-C identify that the unbalanced computation and limited schedulability inside

Received 27 December 2024; revised 8 July 2025; accepted 16 July 2025. Date of publication 24 July 2025; date of current version 5 November 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62302096, Grant 62272098, Grant U24B20152, and Grant 62472391, in part by the Natural Science Foundation of Jiangsu Province under Grant BK20230813, and in part by the Zhishan Young Scholar Program of Southeast University under Grant 2242024RCB0041. Recommended for acceptance by S. Abadal. (Corresponding author: Shuai Wang.)

Borui Li, Tiange Xia, Weilong Wang, Shuai Wang, and Shuai Wang are with the School of Computer Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: shuaiwang_jiot@seu.edu.cn; shuaiwang@seu.edu.cn).

Jingyuan Zhang is with the George Mason University, Fairfax, VA 22030 USA.

Chenhong Cao is with the University of Science and Technology of China, Hefei 230026, China.

Zheng Dong is with the Wayne State University, Detroit, MI 48202 USA.

This article has supplementary downloadable material available at <https://doi.org/10.1109/TMC.2025.3592334>, provided by the authors.

Digital Object Identifier 10.1109/TMC.2025.3592334

the model lead to inference *bottlenecks*. Scaling all operators of an ML model as a whole results in under-provisioned bottleneck areas that hinder performance. To make things worse, with the proliferation of large language models (LLMs), the limited resources of edge devices failed to support scaling the whole model. For example, serving the LLaMA2-7B model with 6-bit quantization requires ~14 GB RAM. While NVIDIA Xavier, a typical edge SoC, is only equipped with 16 GB, which cannot scale the whole model out for bursty requests. Hence, existing scaling policies are *coarse-grained*, which leads to inefficient resource usage and unsatisfied latency SLOs.

Rigidity and inefficiency of model parallelism: Existing serverless frameworks use model parallelism that partitions large ML models across multiple functions to overcome the resource constraints imposed by individual serverless functions. However, their partition points are usually pre-determined offline, which limits the system's ability to adjust to request fluctuations, which is particularly common in edge computing scenarios. The prevalent edge SoC design complicates model parallelism due to the possibility that a partitioned model executes on heterogeneous accelerators within the same SoC. It necessitates the design of an edge-native serverless ML framework to dynamically partition the model and run efficiently on edge SoCs.

To address these unique characteristics of edge computing, we introduce **FluidEdge**, a holistic system designed for expediting serverless ML inference on edge SoCs. First, FluidEdge leverages qualitative and quantitative analysis to identify the root cause of the ML inference bottlenecks in Section II. Based on the observation, we propose a new parallelism space named "fluid model parallelism" which dynamically scales bottleneck components of an ML model during execution to enhance inference performance. FluidEdge operates in two stages: offline and online. During the offline stage, FluidEdge analyzes serverless functions and ML models to perform a *bottleneck-aware pre-partition* for online auto-scaling. Furthermore, FluidEdge designs a *context-based FluidModel composition* approach to facilitate flexible partition composition with minimized overhead. In the online serving stage, FluidEdge employs a hierarchical auto-scaling framework that combines *intra-function scaling through bottleneck detection* and the *request-oriented inter-function scaling* to meet latency SLOs while optimizing resource utilization. Additionally, FluidEdge incorporates a *conflict-free tensor sharing* mechanism to facilitate seamless data sharing between functions scaled out and running on different processing units on edge SoCs.

Our contributions are summarized as follows:

- We identify inference bottlenecks in ML models and their impact on inference performance, introducing the FluidModel abstraction that includes potential bottleneck information for online optimization.
- We propose a hierarchical auto-scaling technology based on the FluidModel, leveraging both intra- and inter-function parallelism to optimize resource use on edge SoCs for better inference performance.
- We implement FluidEdge on the widely-adopted open-source serverless framework, Knative [16], and conduct extensive evaluation with real-world workloads. Results

show that FluidEdge reduces the inference latency by 37.4% and SLO violation by 67.3% -87.6% compared to state-of-the-art serverless inference systems.

II. BACKGROUND AND MOTIVATION

In this section, we briefly introduce the application scenario of FluidEdge and the hardware architecture of edge SoCs. Then, we demonstrate our findings regarding ML inference bottlenecks on edge SoCs.

A. Target Application Scenario of FluidEdge

Nowadays, edge devices are equipped with heterogeneous accelerators like GPUs to form a System-on-Chip (SoC) to support real-time and energy-efficient ML inference. FluidEdge aims to efficiently serve multi-tenant DNNs in an edge computing scenario, which is prevalent in various real-world edge computing use cases.

Specifically, FluidEdge targets two real-world application scenarios. First, various real-world applications *leverage edge SoCs to serve multiple DNN tasks simultaneously* [17], [18] for resource efficiency, such as Industrial IoT and smart cities. For example, smart city [19] deploys edge SoCs at traffic intersections to run multiple video analytics applications concurrently, such as facial recognition and anomaly detection. Second, another recent prevalent scenario is *leveraging multiple edge SoCs to form a physical edge SoC-cluster server* [15], [20]. Taking advantage of the enhanced energy efficiency of edge SoCs and the affinity between edge SoCs and mobile phones, this new form of edge cluster is widely used for multi-tenant cloud gaming as well as efficient DNN training and inference.

In such scenarios, serverless computing is well-suited because (1) the auto-scaling allows it to scale to zero when there are no input tasks to conserve precious edge resources, and (2) it handles bursty and dynamic workloads of edge [21]. However, due to the limited resources on edge SoCs, especially considering multi-tenancy, it is necessary to carefully allocate resources for ML applications to achieve optimized performance. Therefore, we propose FluidEdge to expedite the serverless ML inference.

B. Hardware Architecture of Edge SoCs

The most prevalent edge SoCs are NVIDIA Jetson series, which are widely used in autonomous driving and VR/AR. The architecture of edge devices comprises a CPU and an integrated GPU located on the *same SoC*, which offers a promising starting point for zero-copy data sharing between scaled serverless functions on CPU and GPU. The integrated GPU includes several *Stream Multiprocessor (SM)* and each containing multiple cores for parallel processing. Applications for these edge SoCs are developed using the CUDA programming model, where the GPU executes code through *GPU streams*. A stream contains a series of CUDA kernels and executes them in a *FIFO manner*. During execution, a kernel is instantiated into a different number of CUDA blocks according to its computation complexity. Each block comprises a group of threads that are executed simultaneously within an SM.

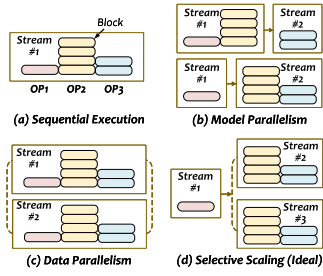


Fig. 1. An example DNN with different parallelism levels.

In modern ML developing frameworks such as PyTorch, ML developers define a set of layers to form a DNN model. These layers are sequential or parallel to each other. Generally, multiple ML operators exist for each layer, and CUDA kernels are used to implement these ML operators. In typical DNN inference scenarios, PyTorch automatically launches the sequence of DNN operators onto a *single default CUDA stream* in the correct topological order. Therefore, by default, the ML operators are executed sequentially, even though they are defined in parallel in the DNN model.

C. Inference Bottlenecks of ML Models

Characterizing serverless inference on edge SoCs: In state-of-the-practice inference engines such as TensorRT and ONNX, an ML model is represented using a direct acyclic graph (DAG) of operators. Typical operators include average pooling, convolution, etc. These operators are implemented with CUDA kernels. In Fig. 1, we exemplify using an ML model with three operators, denoted as OP1-3. Due to the different computation complexity, each operator owns a different number of CUDA blocks for execution.

Before execution, a topological sort is conducted to convert the operator DAG into a *single CUDA stream* by default. Operators in a stream have *strict dependencies* to execute sequentially, e.g., OP2 is supposed to be executed after OP1 completely finishes, as Fig. 1(a). Existing works [9], [10] leverage model parallelism, shown in Fig. 1(b), to partition the model for serving. Other works [3], [8] exploit data parallelism, shown in Fig. 1(c), that splits the ingress request to serve with different serverless functions. We argue that an ideal way for ML inference in edge computing is to selectively scale out a proportion of operators, i.e., OP2 and 3, to obtain better performance with optimized resource efficiency as Fig. 1(d).

To verify the above arguments, we implement the four parallelisms using serverless functions on NVIDIA Orin AGX. An SM in Orin AGX is limited to allocating at most 1,536 threads at once [22]. Inspired by [12], we configure each block in Fig. 1 to have 500 CUDA threads consuming 100 MB memory that execute for 100 ms, and the latency SLO is set to 800 ms. Under this setup, the maximum number of blocks that are concurrently executed on an SM is 3. We then investigate the latency against different ingress throughput configurations as shown in Fig. 2. For model parallelism, we measure both the partition plans in Fig. 1(b) and choose splitting after OP1 for its better

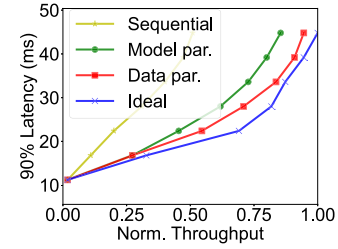


Fig. 2. Latency comparison under varied ingress throughputs.

performance. We observe that selective scaling achieves a lower latency than its counterparts, especially in a request-intensive situation.

Insight 1: Scaling out the judiciously selected proportion of an ML model enhances the inference performance with the same resources.

ML inference bottlenecks: We conduct a more in-depth investigation by collecting block invocation traces and the SM allocation with NVIDIA Nsight profiling toolbox [23]. We excerpt a short period as a representative example in which requests come in at $t = 0, 100, 200$ ms as Fig. 3, and the requests are marked as R1, R2, and R3.

We define the ML “inference bottleneck” as *portions of the ML model’s operators limit the overall throughput and increase the end-to-end latency according to the intrinsic characteristics of the operators*. This phenomenon occurs when operators have limited schedulability or are provisioned with insufficient resources. For example, in Fig. 3(a), operators are sequentially packed into a single CUDA stream by default. Because the *strict dependency* between operators within a stream limits the schedulability of the model, hardware resource under-utilization emerges, as shown in the red frames in Fig. 3(a). Hence, we refer to the group of operators (OP1, 2, and 3) as the *inference bottleneck* under sequential execution.

Merely partitioning the model does not always solve the bottleneck issue. As depicted in Fig. 3(b), the model parallelism approach partitions OP1 to a new stream to decouple the dependency between OP1 and OP2-3 for better schedulability, which facilitates the instant processing of OP1 of each request. However, this partition does not completely resolve the bottleneck because of the *unbalanced computational complexity* between the two partitions. OP1 handles all three requests in a short time while OP2-3 queues the requests due to the lack of computational resources, i.e., under-provisioned. Therefore, we identify OP2-3 as bottlenecks in Fig. 3(b). If we provision more resources to the bottleneck OP2-3 by scaling it to more instances, as shown in Fig. 3(d), the bottlenecks are removed and the hardware resources are fully utilized for optimal performance.

Similarly, the coarse-grained scaling of existing data parallelism, shown in Fig. 3(c), also has limited schedulability due to the dependency inherent in a complete instance. For example, the under-utilization during 200-300 ms is not filled with R3:OP1 because all the inference instances are occupied. The model parallelism also improves by selectively scaling out OP2-3.

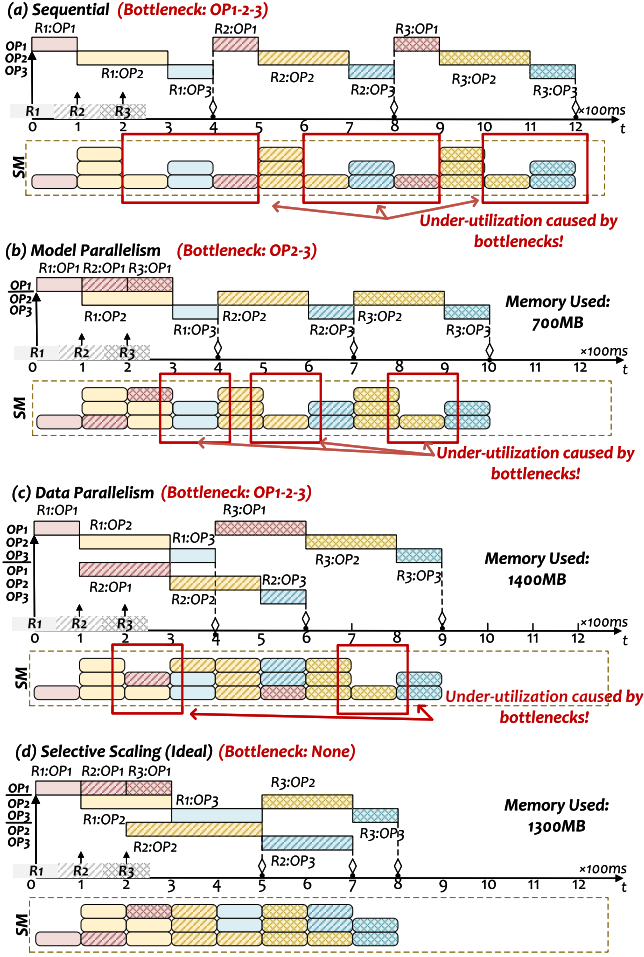


Fig. 3. Inference handling timeline and resource usage of different scheduling policies.

Insight 2: ML models have inherent *bottlenecks* that hinder the inference performance. Bottlenecks come from the *dependency* between ML operators and *under-provisioned* resources.

Based on the insights, FluidEdge offline analyzes the ML model and pre-partitions it according to the dependency. During the online stage, FluidEdge dynamically scales the model partitions to provision appropriate resources for inference bottlenecks to improve the overall performance.

III. FLUIDEDGE OVERVIEW

A. Key Abstraction: FluidModel and FluidOP

Before diving into the design of FluidEdge, we first introduce the key abstraction we use throughout this paper.

As we stated in the motivation, it is crucial to address the bottleneck of models to enhance the performance of ML inference. Traditional model parallelism approaches used by serverless inference divide the ML model before deployment, resulting in sub-optimal performance, particularly under highly dynamic workloads in edge computing environments. In FluidEdge, we

propose a new class of parallelism named *fluid model parallelism*. Its fundamental idea is to eliminate the rigid partition boundaries, dynamically choose proportions of the ML model causing inference bottlenecks during execution, and scale them out for better performance. Towards this, FluidEdge advocates a new abstraction named FluidModel, which consists of FluidOP. Specifically, the original ML model is judiciously partitioned to FluidOPs which consist of one or more ML operators. The term “fluid” means FluidOPs are combined or separated arbitrarily from each other to constitute a FluidModel during execution without incurring significant data passing overhead (cf. Section IV-B and Section V-D).

With this FluidOP and FluidModel abstraction, FluidEdge selectively scales out the proportions of the ML model once it detects one or multiple FluidOPs are the bottleneck that delays the inference. Furthermore, FluidEdge also agilely reorganizes the scaled FluidOPs and adjusts the scaling strategy in response to the bottleneck change due to different incoming request patterns.

B. FluidEdge in a Nutshell

Fig. 4 depicts the system overview of FluidEdge framework. FluidEdge first generates the FluidOPs offline. At runtime, FluidEdge selectively scales out the FluidOPs to mitigate the SLO violation caused by inference bottlenecks.

Offline FluidModel generation: In the offline phase, users submit their serverless application along with the associated ML model for deployment. Taking the serverless function and ML model as input, FluidEdge obtains the corresponding FluidModel by partitioning and generating the FluidOP for online inference serving and scaling. It is worth noting that FluidEdge generates FluidModels for both CPU and GPU execution, depicted as M_{cpu} and M_{gpu} in Fig. 4, to make full use of the computing resources on edge SoCs.

In this generation process, FluidEdge takes the overhead as a vital consideration. Hence, based on the FluidOP abstraction, FluidEdge further reduces the potential overhead by two methods: ① Reducing the number of generated FluidOPs by only selecting the critical operators that are supposed to be inference bottleneck as partitioning points (Section IV-A). ② Optimizing the data transfer between FluidOPs that form a FluidModel with the help of inference contexts and zero-copy mechanism (Section IV-B).

Online hierarchical auto-scaling: In the online phase, FluidEdge leverages a hierarchical auto-scaling mechanism to adapt to the ever-changing edge requests and satisfy user-specified SLOs. Hierarchical here means FluidEdge leverages an ③ intra-function scaling to allocate more resources to the FluidOPs that are considered to be inference bottleneck (Section V-B) and an ④ inter-function scaling to scale out the whole function. The rationale behind this hierarchical auto-scaling is that functions in serverless computing are limited to a certain resource usage [24], [25]. Our intra-function scaling, which is tailored to address bottlenecks, dynamically adjusts resource allocation for different parts of the ML model to enhance resource efficiency based on incoming requests. However, this optimized resource allocation

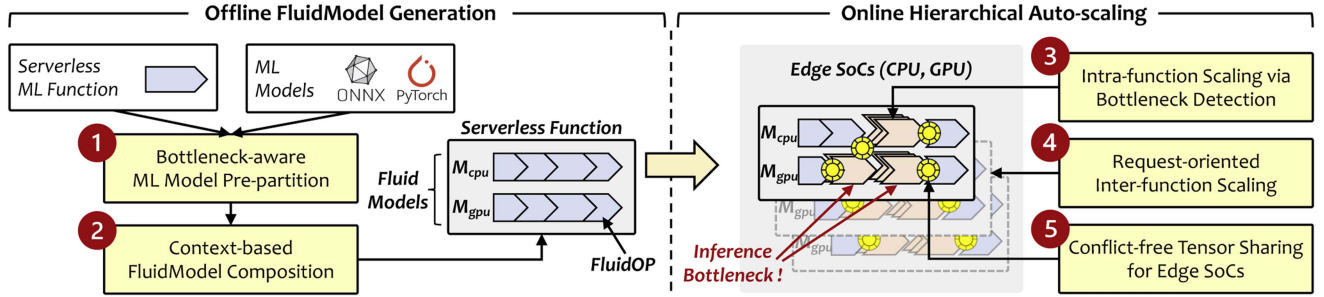


Fig. 4. Overview of FluidEdge framework.

may not suffice when the input requests surpass the maximum bearable rate, at which point the inter-function auto-scaling approach becomes necessary.

Furthermore, when the FluidModel is re-organized for scaling, the tensor sharing between consecutive FluidOPs has a non-trivial overhead. Especially, the heterogeneous processing units (CPU, GPU) on edge SoCs and different batch sizes further worsen this problem. To improve the overall performance, FluidEdge proposes a ⑤ Scaling-oriented Tensor Sharing approach which takes advantage of the hardware unified memory on edge SoCs (Section V-D).

IV. OFFLINE FLUIDMODEL GENERATION

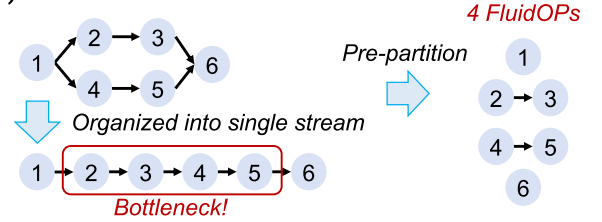
At first glance, the fluid model parallelism requires partitioning the model during execution. A naive solution is to pause inference serving, generate the partitioned models, and solve data dependencies between partitions, which leads to service suspension and SLO violation. Even if we prepare another partitioned model without interrupting other models from inference using techniques such as blue-green deployment, it will consume more computing resources. Hence, FluidEdge propose an offline FluidModel generation approach for the agile yet low-overhead online dynamic scaling. Specifically, considering the tradeoff between the schedulability and runtime overhead, FluidEdge strategically pre-partitions the ML model to FluidOPs with full awareness of the potential bottlenecks instead of separating each operator individually (Section IV-A). To further minimize the communication overhead between FluidOPs, FluidEdge introduces a lightweight context-based FluidModel composition (Section IV-B).

A. Bottleneck-Aware ML Model Pre-Partition

As Insight 2 in Section II-C suggests, dependencies and the under-utilization of parallelism lead to the inference bottleneck. During this offline stage, FluidEdge first investigates the model for potential bottlenecks and pre-partitions the model accordingly to break the dependencies and facilitate the online bottleneck detection and scaling. We identify two kinds of potential bottlenecks that occur due to different dependencies between operators: *architectural bottleneck* and *resource-bounded bottleneck*.

Architectural bottleneck: As we discussed in the background, the operator DAG of an ML model goes through a topological

(a) Architectural Bottleneck



(b) Resource-bounded Bottleneck

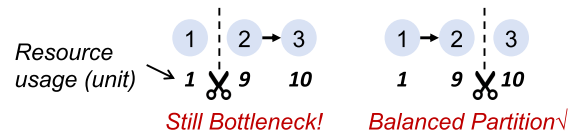


Fig. 5. Illustration of two kinds of potential bottlenecks.

sort and is organized in a single stream. As shown in Fig. 5(a), we use a DNN model including six operators as an example to illustrate the architectural bottleneck. By default, the operators are packed into a single CUDA stream, leading to a sequential execution of the parallel operators such as OP2-3 and OP4-5. This organization limits the schedulability between parallelizable operators and incurs resource underutilization. We recognize this phenomenon as the architectural bottleneck. To facilitate the online mitigation of this kind of bottleneck, FluidEdge pre-partitions different branches of the operator graph to distinct FluidOPs. With this partitioning, FluidOPs are able to be separated into different CUDA streams during the online auto-scaling, providing a feasible way to mitigate the bottleneck.

Resource-bounded bottleneck: Partitioning the operators in a model into several independent FluidOPs breaks the strict dependency between them and reduces the bottleneck. More partitions lead to better schedulability, but also higher coordination overhead. Therefore, FluidEdge groups consecutive operators to a single FluidOP which is the minimum online scaling unit – but how to select an optimal grouping strategy?

As illustrated in Fig. 5(b), each operator has its resource usage, such as CUDA threads and memory. When partitioning between OP1 and OP2 to create two CUDA streams, the bottleneck persists between OP2 and OP3. This unbalanced partition leads to a situation where a significant number of requests complete OP1 quickly but queue for OP2-3 due to their computational

Algorithm 1: Resource-Bounded Pre-Partition.

Input: Set of ML operators $G_{OP} = \{g_1, \dots, g_{|G_{OP}|}\}$.
Output: Set of FluidOPs G_{FOP} .

- 1 $V_{i,j}$ denotes the variance from operator i to j ;
- 2 $DP_{i,j}$ is sum of variance when i partitions is applied for the first j operators, V_{max} is the variance limit ;
- 3 $i \leftarrow 1, G_{FOP} \leftarrow \emptyset$;
- 4 **while** $DP_{i,|G_{OP}|} > V_{max}$ **do**
- 5 $i \leftarrow i + 1$;
- 6 **for** $j \leftarrow i \sim |G_{OP}|$ **do**
- 7 $k \leftarrow i - 1$,
- 7 $k^* \leftarrow \arg \min_k^{j-1} (DP_{i-1,k} + V_{k+1,j})$;
- 8 $DP_{i,j} \leftarrow DP_{i-1,k^*} + V_{k^*+1,j}$;
- 9 $G_{FOP} \leftarrow G_{FOP} \cup \{g_{k^*}, \dots, g_j\}$;

complexity. The online scaling technique scales out OP1 with multiple instances to balance the computation load. However, managing too many instances results in significant overhead. A promising solution lies in balancing the resources of each partition, as depicted on the right side of Fig. 5(b). This strategy effectively decouples the dependency of operators and enhances scheduling efficiency without introducing substantial management overhead.

The next question is to determine how many FluidOPs to partition into. The main goal of this pre-partitioning is to facilitate SLO-aware online scheduling. We derive a theorem¹ related to the SLO requirement T_{SLO} , e.g., the 90th percentile inference latency should not exceed 500ms, and the sum of the variance of resource demand between FluidOPs is denoted as V_{sum} . For a given SLO requirement T_{SLO} of an ML model, there exists a variance threshold V_{max} such that if V_{sum} less than V_{max} , then T_{SLO} is able to be satisfied through online scaling. Consequently, for resource-bounded bottlenecks, FluidEdge pre-partitions the model into a minimal number of FluidOPs that conform to $V_{sum} \leq V_{max}$.

FluidEdge solves this partitioning problem via dynamic programming, as shown in Algorithm 1. Lines 6-9 search for the optimal partition point k^* , and line 4 constrains the generation with the variance threshold V_{max} . Before pre-partition, FluidEdge profiles the ML model for necessary data, e.g., the execution time and resource usage of each operator. To reduce the search space, FluidEdge performs an operator fusion to combine the correlated operators as [26]. Finally, we combine the architectural and resource-bounded bottlenecks as partition points and perform the pre-partition for the online fine-grained auto-scaling.

B. Context-Based FluidModel Composition

After we obtain a set of FluidOPs by identifying the potential bottleneck of an ML model, the next question is how to compose

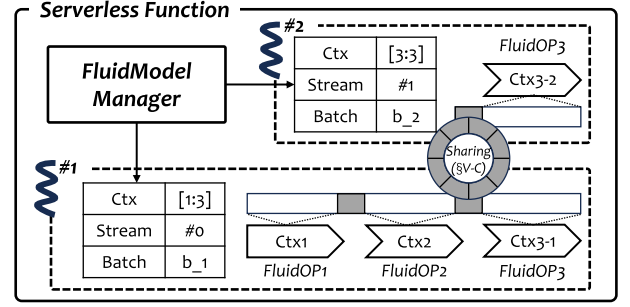


Fig. 6. Lightweight context-based FluidModel composition approach.

a FluidModel with these operators. A satisfying composition mechanism should (1) allow agile decoupling when decided to be individually scaled without sacrificing the serving performance, and (2) exhibit negligible overhead when FluidOPs are not partitioned with each other.

Hence, we propose a lightweight FluidModel composition based on the context mechanism of existing ML inference frameworks. A context is an instantiated model ready for execution with all the necessary parameters and a reserved memory region for model I/O. By default, a context is run by a single thread. However, to facilitate the agile decoupling, FluidOPs are instantiated to individual inference contexts as Fig. 6. When multiple FluidOPs serve as a whole, they are mapped to a thread (e.g., thread #1) inside the serverless function. FluidEdge leverages zero-copy data sharing between contexts to minimize the overhead.

Once FluidEdge detects a FluidOP as bottleneck and decides to scale it out separately, it notifies the FluidModel Manager inside the function to spawn new inference threads (e.g., thread #2 in Fig. 6) and contexts. We also employ a zero-copy ring buffer between the scaled threads (cf. Section V-D). Each thread has its configuration table that records its context, CUDA stream, and batch size information. FluidModel Manager asynchronously modifies the table to adjust the behavior without disturbing the current execution.

V. ONLINE HIERARCHICAL AUTO-SCALING

A. Design Principles of FluidEdge Auto-Scaling

Before we dive into the details of the auto-scaling algorithms of FluidEdge, we first present its design principles. Commercial-off-the-shelf serverless platforms pose resource constraints for each function. Therefore, during online scaling, FluidEdge scales FluidOPs and functions to *meet the latency SLO while minimizing resource usage*. Based on offline pre-partitioning of FluidEdge, we design a hierarchical auto-scaling mechanism that consists of both inter- and intra-function scaling to meet the SLOs. The intra-function auto-scaling detects the inference bottleneck and scales out the FluidOPs within a function, confronting the user-specified SLO under resource constraints of a single function. This intra-function auto-scaling finds the optimal resource allocation under dynamic request

¹We provide the proof in supplemental materials due to page limit.

patterns. The inter-function scaling aims to increase resources on a function-by-function basis to achieve further performance improvements when incoming requests surpass the capacity of a single function. Furthermore, we also introduce an efficient tensor sharing mechanism between the scaled FluidOPs in Section V-D.

B. Intra-Function Scaling Via Bottleneck Detection

The intra-function scaling first finds the bottleneck FluidOP, then models the scaling decision process as a Markovian Decision Process (MDP) and obtains optimal scaling action using reinforcement learning (RL).

Bottleneck detection: To detect the bottleneck occurrence online, we introduce *congestion intensity* of each FluidOP to express the degree of dispersion of the corresponding FluidOP's processing latency, which is defined using the ratio of the 99th percentile latency to the median latency. A high congestion intensity means that the execution latency of the FluidOP fluctuates in a large range, which is usually due to the congestion and stacking of requests in addition to the FluidOP. A typical case is OP2/3 in Fig. 3(c). It also means that scaling out the FluidOP brings about a larger change in the execution latency.

Why RL? The rationale for FluidEdge to use RL for intra-function auto-scaling decisions is two-fold: (1) RL is well-suited for learning sequential decision policies with complex dynamics, and the actions are well-defined. Deciding the auto-scaling operations on the detected bottleneck FluidOP is such a problem. (2) After each scaling decision, the connectivity structure of FluidOPs alters. This topology change may fluctuate or even invalidate traditional policy- or modeling-based scaling approaches [3], [16]. Therefore, FluidEdge leverages a model-free, actor-critic RL framework with a deep deterministic policy gradient (DDPG) algorithm to handle the bottleneck-aware auto-scaling. This model-free RL algorithm is well-suited for our problem as it adapts to the environment dynamics caused by auto-scaling.

Problem formulation: The goal of intra-function scaling is to obtain the optimal scaling factor of each FluidOP. We use $k_{i,j}^{FOP}$ to denote the scaling number of i -th FluidOP in function j . Hence, we formulate the decision problem as an MDP whose basic elements are states, actions, and rewards.

1) **States:** At t -th round, the state of our intra-function auto-scaling model s^t mainly contains the latency such as the P90 latency of timestep t , i.e., T_{P90}^t . Furthermore, FluidEdge also considers the utilization of different kinds of resources RU_m^t , $m \in \{CPU, GPU, Mem\}$ and the request arrival rate TP_{in} for decision. Therefore, $s^t = \{T_{P90}^t, RU_m^t, TP_{in}\}$.

2) **Actions:** After the observation of state s^t , the RL agent selects the action a^t to perform. In FluidEdge, we design three kinds of actions that are applied on the bottleneck FluidOP: $\{k_{i,j}^{FOP}, p_i^t, NoOp\}$. $k_{i,j}^{FOP}$ controls scaling out and scaling in the specific FluidOP i . A binary variable p_i^t denotes whether the FluidOP i is split or merged. If ingress requests stably achieve the latency SLO for a long time, and there is a long period of resource utilization redundancy, the bottleneck FluidOP needs to be scaled down or merged.

3) **Rewards:** Given the action a^t , the RL agent will receive a reward r^t . As we mentioned in the design principles, FluidEdge jointly considers the latency and resource usage. Additionally, it is essential to incorporate penalties to ensure that the agent adheres to the SLO constraint. Therefore, the reward r^t is defined as:

$$r^t = \begin{cases} T_{SLO}/T_{P90}^t + \beta \left(1 - \frac{1}{m} \sum_{i=1}^m RU_m^t\right), & T_{P90} < T_{SLO} \\ -T_{P90}/T_{SLO}, & T_{P90} \geq T_{SLO} \end{cases}, \quad (1)$$

where β is a coefficient to control the preference of optimization goals. The first equation of r^t means that if the observed P90 latency satisfies the SLO, the RL agent will receive a reward according to how low the latency it achieves and how many resources it uses. The second equation indicates a negative reward when the SLO is violated. With the above problem formulation, we train RL agents using the DDPG algorithm to obtain optimal intra-function scaling decisions.

C. Request-Oriented Inter-Function Scaling

When the ingress request exceeds the limit of a function instance, FluidEdge scales out the whole function with more instances. During online serving, FluidEdge records the throughput of each function instance TP_f when the requests are processed in accordance with the user-specified latency SLOs. Inspired by [10], FluidEdge leverages a request-oriented, threshold-based approach to reduce scaling overhead and maintain the adaptivity to existing serverless frameworks. According to the request pattern, FluidEdge proposes two thresholds, Q_{out} and Q_{in} , to determine whether to scale out/in the functions.

$$Q_{out} = \sum_{f \in F} TP_f / RU_f, \quad Q_{in} = \sum_{f \in F} TP_f - \min_{f \in F} TP_f. \quad (2)$$

With the request arrival rate $TP_{ingress}$, we discuss the decisions of the scaling strategies in three cases.

- 1) $TP_{ingress} > Q_{out}$: It indicates that the existing functions cannot satisfy the ingress requests. FluidEdge will scale out new functions for latency-sensitive inference.
- 2) $Q_{in} < TP_{ingress} \leq Q_{out}$: It means the existing function instances meet the demand for steady service requests. FluidEdge will continue to invoke intra-function scaling to tune the intrinsic configurations under the current requests.
- 3) $TP_{ingress} \leq Q_{in}$: It implies that the existing functions are already redundant for handling the requests. Hence, FluidEdge will scale down the function instances to save precious resources for multi-tenancy on edge SoCs.

D. Conflict-Free Tensor Sharing for Edge SoCs

In the intra-function auto-scaling process, sharing intermediate tensors across the partitioned FluidModel (Section IV-B) has a significant influence on the overall performance. In FluidEdge, designing such a tensor-sharing mechanism should minimize the sharing overhead while taking the following unique problems into consideration. (1) The sharing involves FluidOPs that run on heterogeneous accelerators on edge SoCs. (2) The sharing is a multi-producer, multi-consumer (MPMC) sharing problem. This occurs due to the presence of auto-scaling. It generates

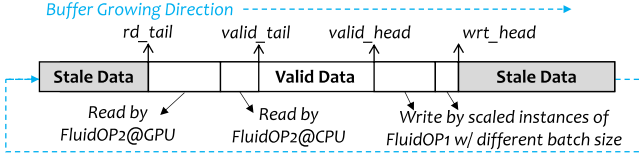


Fig. 7. Illustration of the scaling-oriented tensor sharing of FluidEdge.

multiple instances of the FluidOP that write tensors to the buffer, i.e., producer, and the FluidOP reads the buffer, i.e., consumer.

Taking advantage of the hardware unified memory on edge SoCs, we propose a conflict-free ring buffer to support zero-copy sharing between scaled FluidOPs on CPUs and GPUs. A ring buffer facilitates asynchronous data exchange and is suitable for MPMC problems. However, in FluidEdge, the consumer FluidOPs who read the data in a zero-copy manner must retain the data until its computation is complete, which is time-consuming. This problem breaks the asynchronism of the ring buffer and leads to data conflict and overwriting, which is unacceptable in FluidEdge. Towards this problem, we propose a conflict-free ring buffer with the additional pointer that records the data being processed by FluidOPs instead of naive read/write index pointers of the traditional ring buffer. In Fig. 7, each time a succeeding FluidOP reads the data, it modifies `valid_tail` with its batch size to declare its ownership of the data. Then it processes the data directly on the memory region and updates the `rd_tail` to release the buffer. The procedure of the write operation is similar. This specialized design facilitates conflict-free tensor sharing between various FluidOPs and processors.

VI. EVALUATION

In this section, we compare and analyze FluidEdge with other serverless ML frameworks.

A. Experimental Setup

Implementation: During the design and implementation of FluidEdge, we take the adaptivity to existing serverless frameworks as a vital rule for further research in serverless edge intelligence. Hence, FluidEdge is built on top of a widely used, open-source serverless system, Knative [16], and TensorRT inference engine. The hierarchical auto-scaling mechanism is implemented with KEDA, the event-driven scaling framework for Kubernetes. The intra-function RL scheduler is implemented as an add-on of KEDA, and the RL hyperparameter β in Equ. (1) is set to 1 for equally optimizing the latency and resource usage, which is both important in the edge scenario.

Device setup: We evaluate FluidEdge on two commonly-used edge SoCs with varying performance, i.e., the NVIDIA Jetson Orin and Xavier. NVIDIA Jetson Orin has a 1024-core NVIDIA Ampere GPU with 32 Tensor Cores, a 6-core Arm CPU, and 32 GB of shared memory. NVIDIA Jetson Xavier has a 512-core NVIDIA Volta GPU with 64 Tensor Cores, an 8-core Arm CPU, and 16 GB of shared memory.

Baselines: To demonstrate the performance of FluidEdge, we use four state-of-the-art serverless inference frameworks for

TABLE I
APPLICATION BENCHMARKS USED FOR EVALUATION

Application	Inference Pipeline	
Traffic Monitoring (APP-TM)	SSD	ShuffleNet
	Vehicle detection	Vehicle model recognition
Person Tracking (APP-PT)	TinyYOLOv3	VGG
	Human detection	Person identification
Video Understanding (APP-VU)	Inception	BERT
	Video content analysis	Content understanding

comparison, shown as follows: (1) *INFless* [3] is the state-of-the-art serverless system dedicated to ML inference. It proposes a dynamic batch mechanism under data parallelism during inference. It supports both CPU and GPU for inference. (2) *BATCH* [8] is a serverless ML serving system with an adaptive batch approach to reduce the inference cost. (3) *Gillis* [9] leverages model parallelism to achieve latency-optimal ML inference. (4) *Knative+* [16] is a state-of-practice open-source serverless framework. We enhance the vanilla BATCH and Knative by adding GPU support with input batching in our evaluation for fair comparison.

Benchmarks: In our evaluation, we use frequently used ML models for comparison, namely (1) AlexNet, (2) VGG-19, (3) VGG-108, and (4) ResNet. We also include recent LLMs for comparison, i.e., (5) BERT and (6) Llama2-7B. Furthermore, to evaluate FluidEdge in more real-world usage scenarios, we also excerpt three production ML applications from [3], [27] which leverage multiple ML models for complicated tasks. As shown in Table I, three application benchmarks are traffic monitoring (APP-TM), person tracking (APP-PT), and video understanding (APP-VU). All models selected as or included in benchmarks are directly downloaded from the ONNX model zoo and converted to TensorRT format.

Workloads: We use two production traces to evaluate how FluidEdge performs under real-world inference workloads. The first trace is the representative workload used in various serverless edge research studies, excerpted from Azure serverless functions [28]. Additionally, we utilize a workload trace collected from a commercial, multi-tenant edge service provider, Alibaba ENS, in China [21].

B. Latency Reduction and SLO Satisfaction

First, we evaluate how FluidEdge expedites ML inference serving in terms of latency and SLO satisfaction.

Comparison under different throughputs: We first evaluate how FluidEdge accelerates serverless inference under different throughputs. As we stated in Section IV-A, 90th percentile (P90) latency is a common metric for inference performance [3]. Hence, we compare the P90 latency under different workloads to provide a comprehensive view of FluidEdge's performance. The results are shown in Fig. 8.

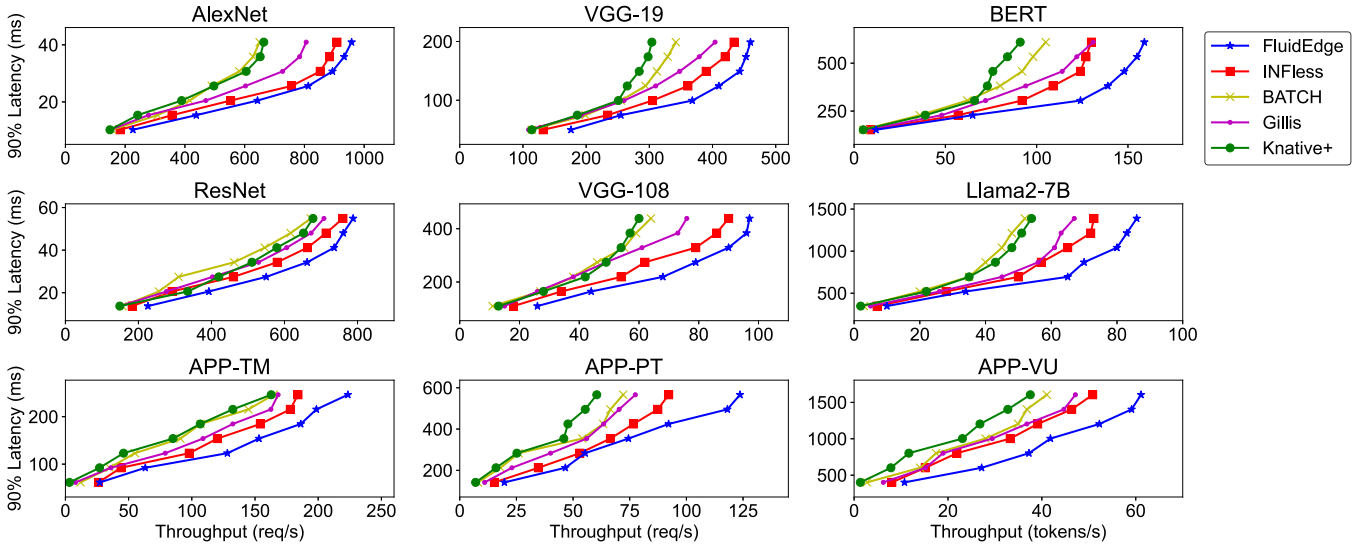


Fig. 8. Comparison of the inference latency under different ingress request throughputs among FluidEdge and baselines.

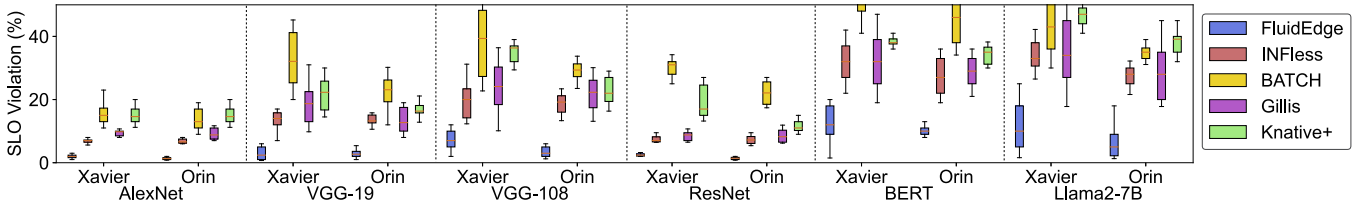


Fig. 9. SLO violations on different edge SoCs under *Azure workload*. SLOs are set to $1.2\times$ of the ideal inference latency of each benchmark.

We observe that FluidEdge achieves 37.4% average latency reduction across all the benchmarks compared with the best-performed baseline, INFless. We attribute this performance gain to bottleneck-aware partitioning (Section IV-A) and auto-scaling (Section V-B). On the one hand, INFless and BATCH mainly focus on the adaptive batching of input requests, which employ data parallelism to scale the entire model. In contrast to the two baselines, FluidEdge utilizes a more nuanced intra-function scaling. This fine-grained resource allocation distributes resources based on the architectural and runtime profiles of different components within the model, which leads to better performance. On the other hand, compared to the state-of-the-art model parallelism approach, Gillis, FluidEdge achieves 51.7% improvement on average because FluidEdge dynamically adjusts the topology of the model online to cope with different ingress patterns, while Gillis' offline partition cannot adapt to. For the application benchmarks, FluidEdge also exhibits 46.5% acceleration on average, indicating FluidEdge's adaptability to complex real-world applications.

Another main observation is that FluidEdge shows a larger lead in harsh environments, such as a larger ML model or higher ingress request throughputs. As the scaling law of ML models continues to affect, there is a growing potential for FluidEdge to play a more significant role in advancing edge intelligence in the future.

Comparison under real-world workloads: To evaluate how FluidEdge performs in different situations, we leverage two production workloads of edge computing and use two different edge devices, NVIDIA Orin and Xavier, in the experiment. We set the latency SLO to $1.2\times$ and $2.0\times$ of the ideal inference time and record the SLO violation rate for comparison. Fig. 9 shows the results of $1.2\times$ SLO, indicating a strict constraint, under the *Azure workload*. Figs. 10 and 11 illustrate the results of $2.0\times$ SLO, which means a relatively looser constraint, under *Azure* and *Alibaba workload*, respectively.

We observe in Fig. 9 that compared with the baselines, FluidEdge reduces the SLO violation rate by 78.0% -87.6% on Xavier and 67.3% -82.9% on Orin. We also see from the figure that the SLO violations on Orin are lower than on Xavier because Orin has better computation power. If we relax the latency SLO constraint to $2.0\times$ SLO, we see from Fig. 10 that FluidEdge still achieves the best SLO satisfaction. Compared with baselines, FluidEdge reduces SLO violation rate by 67.6% -86.3% on Xavier and 67.1% -85.7% on Orin, which illustrates FluidEdge's generality to both harsh and relaxed situations. BATCH has an unsatisfying performance mainly because it focuses on improving inference performance via a batching mechanism, which may violate the SLO for waiting for additional requests for optimal batch sizes. We further investigate the performance under different traces, i.e., Figs. 10 and 11. We observe that

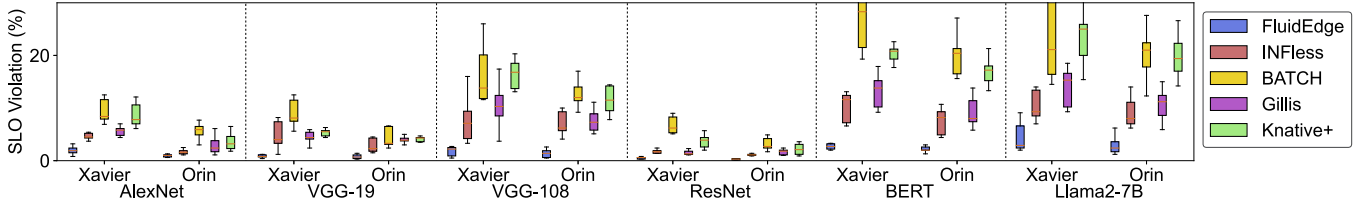


Fig. 10. SLO violations on different edge SoCs under *Azure workload*. SLOs are set to $2.0\times$ of the ideal inference latency of each benchmark.

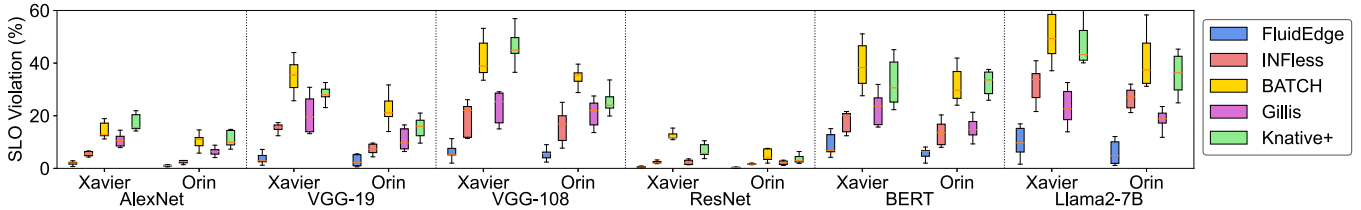


Fig. 11. SLO violations on different edge SoCs under *Alibaba workload*. SLOs are set to $2.0\times$ of the ideal inference latency of each benchmark.

FluidEdge reduces the SLO violation rate by 49.1% -77.8% on Xavier and 57.0% -85.4% on Orin under the Alibaba workload. The requests in the Alibaba workload are more fluctuating than Azure's. FluidEdge shows superior performance under the Alibaba workload, illustrating its ability to handle highly dynamic scenarios, which is advantageous in edge computing.

Summary: FluidEdge achieves better performance on both pressure tests and realistic workloads. Compared with existing approaches, the performance gain of FluidEdge comes from the following design choices: (1) The key idea of FluidEdge is bottleneck-aware partitioning and dynamic scaling. Existing frameworks (e.g., INFless, Gillis) utilize inflexible partitioning (data/model parallelism) that cannot adapt to dynamic bottlenecks. The bottleneck detection and scaling of FluidOPs eliminate under-provisioned operators, directly reducing queuing delays caused by ML operator dependencies, which leads to better performance. (2) Based on the bottleneck detection, FluidEdge combines intra-function (fine-grained operator scaling) and inter-function (coarse-grained instance scaling), allowing it to balance resource allocation across multiple dimensions. This design is suitable for serverless edge computing scenarios, which typically have limited resources. (3) The context-based FluidModel composition during the offline stage and the tensor sharing during the online stage further reduce the overhead for bottleneck handling, which is also necessary for better inference performance.

C. Bottleneck Detection and Mitigation

We now give a more detailed evaluation of our key contribution – bottleneck-aware pre-partitioning and auto-scaling.

Offline pre-partitioning effectiveness: To evaluate the effectiveness of our pre-partition algorithm, we implement an exhaustive partition approach that separates all the ML operators, denoted as *AllCut*, and compare the online scaling results under the same workload. If the scaling results are similar between the two partition strategies, it suggests that FluidEdge accurately

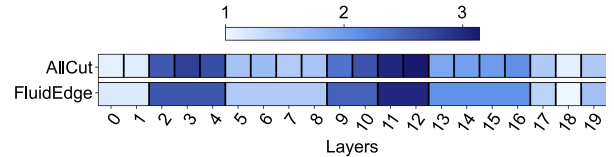


Fig. 12. Effectiveness of the bottleneck-aware pre-partitioning.

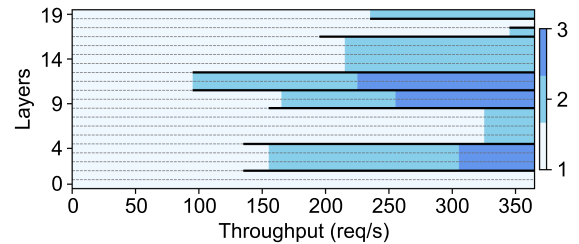


Fig. 13. Bottleneck detection and auto-scaling results of VGG-19 under different throughputs. Bold lines indicate the partition points, and the color shade indicates the number of scaled instances.

identifies the bottleneck that needs to be scaled. Due to the page limit, we use VGG-19 as an example to show the average scaling number under real-world workload in Fig. 12. The partition points of FluidEdge is after operator #1, #4, #8, #10, #12, #16, #17 and #18. We observe that both partition plans are scaled similarly. The operators #2-#4, and #9-#12 are convolutional layers with high computing complexity, which is scaled accordingly. This phenomenon means FluidEdge achieves similar scaling performance with fewer cutting points, which leads to lower runtime partitioning overhead.

Online bottleneck scaling and mitigation: We then move on to how FluidEdge performs for online bottleneck scaling with Fig. 13. As the input throughput grows, operators #9-#12 are partitioned and scaled first, and then operators #3-#5. The operators #0-#1 are not partitioned until the last. This is because the #3-#5 and #9-#12 contain multiple convolution operators,

TABLE II
CONCURRENT INFERENCE LATENCY (MS) UNDER DIFFERENT THROUGHPUT

RPS	FluidEdge	INFless	BATCH	Gillis	Knative+
100	50.0, 121.4	69.6, 131.6	89.5, 122.6	70.8, 154.4	74.6, 128.5
200	90.1 , 211.0	129.3, 195.2	220.0, 293.1	140.9, 307.7	213.8, 427.6
300	208.0 , 461.0	223.8, 482.4	<u>278.4</u> , <u>516.4</u>	<u>244.3</u> , <u>527.6</u>	<u>269.1</u> , 549.6

* Bold/Underlined value denotes the lowest latency/SLO violation.

TABLE III
THROUGHPUT COMPARISON OF FLUIDEDGE W/ AND W/O RL

(RPS)	AlexNet	ResNet	VGG-19	VGG-108	BERT	Llama2-7B
w/ RL	643.68	550.36	368.44	68.47	124.15	65.53
w/o RL	493.34	503.80	295.69	54.85	76.29	41.82

The throughputs are measured when SLOs are set to 2.0× of the ideal inference latency.

which are computationally intensive and should be scaled for better performance. We also find that when the throughput is low, i.e., the request pressure is low, FluidEdge still composites the pre-partitioned FluidModels to avoid inter-FluidOP transmitting overhead (Section IV-B). This shows the superiority of FluidEdge compared with the existing model parallelism.

D. Concurrent Inference Serving

We then evaluate the multi-tenant inference performance of FluidEdge by simultaneously serving multiple ML models, which is a common scenario for serverless computing. We exemplify this through the colocation of VGG-19 and VGG-108. We set the latency SLO to a relaxed constraint, specifically 4× ideal inference time for each model, which is 240ms and 500ms. We show the average inference latency of the VGG-19 and VGG-108 models in Table II.

The first observation is that FluidEdge satisfies the latency SLO under all workloads, while BATCH, Gillis, and Knative+ baselines fail, especially under high pressure. Furthermore, FluidEdge achieves better latency performance than all counterparts except 200 RPS. Under 200 RPS, we observe that the summed latency is minimized, although FluidEdge does not achieve the lowest latency for both models. FluidEdge sacrifices the performance of VGG-108 without SLO violation for more gains of the VGG-19, aiming to achieve a balanced inference performance in concurrent inference serving scenarios. When we calculate the proportion between observed latency and SLO of the concurrent inferences, we find values of (0.38, 0.42) for FluidEdge and (0.54, 0.39) for INFless across the two models. We see that FluidEdge achieves more fair scheduling for both models, which is important for multi-tenancy.

E. Effectiveness of RL-Based Auto-Scaling

In this part, we assess the impact of our RL-based intra-function autoscaler on performance improvement. In this ablation study, we replace the RL-based algorithm with a direct scaling algorithm to emphasize the significance of our design. The evaluation results are shown in Table III. We observe from the table that, on average, the RL-based fine-tuned resource allocation within functions enhances throughput by 34.7%. This improvement is attributed to our RL-based scaling algorithm's

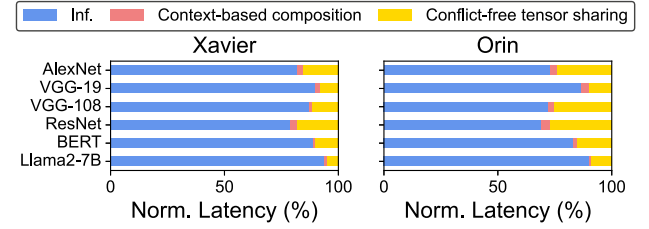


Fig. 14. Breakdown of the inference latency of each benchmark.

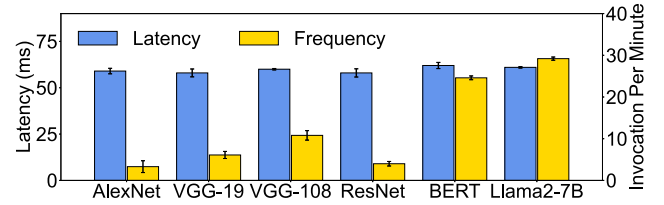


Fig. 15. Scheduling overhead of RL-based hierarchical auto-scaler.

ability to dynamically adjust resource allocation for each FluidModel, resulting in more efficient utilization of limited serverless function resources. Furthermore, it can adapt to different model structures and sizes, eliminating the need for manual scaling rule adjustments for varying models.

F. Overhead Analysis of FluidEdge

Next, we examine the performance impact of FluidEdge in both the offline and online phases.

Overhead during online inference: To have a more detailed investigation of how the context-based FluidModel composition (Section IV-B) and conflict-free tensor sharing (Section V-D) reduce the overhead, we illustrate the breakdown of the latency under the real-world invocation trace in Fig. 14. The context-based composition takes up 1.8% of the overall latency on Xavier and 2.6% on Orin. Specifically, it includes the time spent on creating new contexts and transferring FluidOPs between contexts when FluidEdge decided to scale out. We attribute this negligible latency to aggregating the unpartitioned FluidOPs to a single inference thread with multiple contexts, which enables dynamic online partitioning. The time spent on the data transmission in the ring buffer takes up 11.5% of the latency on Xavier and 16.8% on Orin. This ring buffer simultaneously facilitates the asynchronous inference between CPUs and GPUs, improving overall performance.

Another kind of online overhead of FluidEdge is mainly due to the scheduling tax of the RL-based hierarchical auto-scaler. Hence, in Fig. 15, we illustrate the latency to obtain a scheduling decision and the frequency to invoke the RL algorithm. We observe that, regardless of the model, the scheduling latency is 59.7 ms on average, while the scheduler is invoked 3.3-29.2 times per minute. We also notice that for BERT and Llama2-7B models, the scheduler is invoked more frequently than other models. This is because a larger model generates more FluidOPs after partition, and scheduling each FluidOP has a smaller

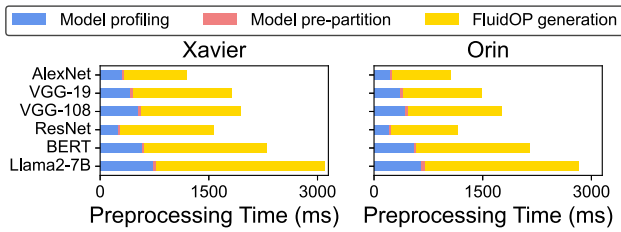


Fig. 16. Offline FluidOP generation overhead of FluidEdge.

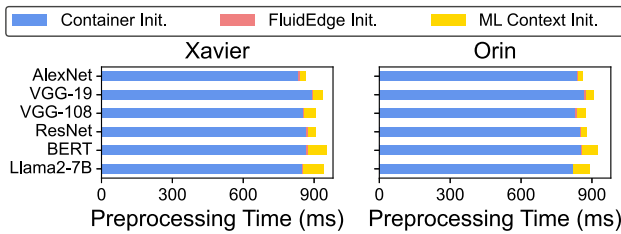


Fig. 17. Function cold-start overhead of FluidEdge.

change in resource usage. Therefore, the scheduler needs to schedule more times to match the expected performance.

Offline generation overhead: During the offline generation (Section IV), FluidEdge first profiles the ML model, pre-partitions the model while fully aware of the inference bottleneck, and finally generates the FluidOPs. Fig. 16 demonstrates the time spent on this generation process. We observe that the pre-partitioning time only takes 2.0% of the overall offline generation time on average, which illustrates the efficiency of FluidEdge’s bottleneck-aware pre-partitioning algorithm. Most of the time is spent on the FluidOP generation, which recompiles the ML model and exports the executable. Hence, the larger the model (e.g., Llama2-7B), the more time it requires.

Function cold-start overhead: When FluidEdge scales up a function from zero, the cold-start process involves the cold start of containers, the initialization of the FluidEdge core components within those containers, and the construction of the FluidModel inference context. Fig. 17 demonstrates the time spent on this cold-start process. We observe that the initialization of FluidEdge components takes only 0.6% -0.7% of the overall overhead across different devices, illustrating the lightweight nature of FluidEdge. Most of the cold start time is due to initializing containers and inference contexts. The container initialization process, which is akin to the existing serverless cold-start period, mainly creates an executable container instance and mounts necessary files from the disk, including the pre-partitioned model. During the ML context initialization, the ML model runtime instantiates each FluidModel to an inference context. Although these overheads are not negligible, various existing studies aim to accelerate these two processes. For example, to reduce the cold start latency of containers, researchers propose mitigation strategies such as function pre-warming [3], [6] and container sharing [29]. Due to FluidEdge’s non-intrusive design concerning existing serverless frameworks, these optimizations for container-level and context-level cold starts are easily adopted.

VII. DISCUSSION

Platform generality: The key insight of FluidEdge lies in the observation of the inference bottleneck. It is not specific to platforms like CUDA or NVIDIA Jetson but rather a widespread occurrence. Specifically, the architectural bottleneck arises from the topological sort of operators. This technique is commonly used in ML inference engines, such as CUDA by NVIDIA and QNN by Qualcomm, leading to inference bottlenecks. Conversely, the resource-bounded bottleneck results from imbalanced resource utilization among model partitions, which is also platform-agnostic.

Auto-scaling over multi-node SoC clusters: Though our experiments are conducted on top of a single-node edge device, FluidEdge is also applicable to multi-node SoC clusters. The current scaling policy in FluidEdge includes intra-function and inter-function scaling. When applying to a multi-node system, the inter-function scaling can be easily extended to scale functions to other nodes. Specifically, modifying the scaling strategies specified in Section V-C to consider the load balancing and heterogeneity of different nodes.

Energy and cost metrics: FluidEdge shows substantial improvements in the latency and throughput of ML inference by achieving a fine-grained yet optimized resource allocation via bottleneck-aware auto-scaling. These enhancements also lead to better energy efficiency in practical edge computing systems by reducing energy waste on non-bottleneck parts of the models. In terms of costs, the existing commercial serverless system charges based on function invocation times and per-invocation resource usage, calculated by the multiplication of memory usage and duration. Our fine-grained resource allocation facilitates cost reduction by decreasing the duration of each request and lowering memory consumption more effectively.

VIII. RELATED WORK

Serverless edge computing: With the proliferation of edge clouds and the ever-growing demands on low-latency and privacy-preserving computation, serverless edge computing (or deviceless computing) [30] has emerged as an important way to deploy applications at the edge. Several works focus on building serverless edge frameworks such as Sledge [31] and Mu [32]. Other works mainly focus on function deployment optimization, which is a vital problem in serverless edge, different from traditional serverless on the cloud. Zygote container [33] provides an efficient container reuse mechanism for the serverless edge. StepConf [34] optimizes the resource allocation of the serverless function workflows. FluidEdge borrows its idea of two-level optimization inside and outside the function and proposes the hierarchical auto-scaling technique. However, upon examining the current landscape, it lacks a dedicated serverless framework for ever-growing ML tasks on the edge, which motivates us to propose FluidEdge.

Serverless ML inference: Various approaches exploit different levels of parallelism to accelerate ML inference without modifying the model, such as data parallelism, model parallelism, and hybrid parallelism. For example, Opapa [35] proposes an

operator parallelism, which is a fine-grained model parallelism, to expedite ML inference.

In the realm of serverless ML inference, some existing works leverage data parallelism [4], [8], [36] to optimize the resource efficiency. The most related work to FluidEdge is INFless [3]. Instead of “patching” existing serverless systems such as AWS Lambda [24] or Azure Functions [25], INFless presents a domain-specific serverless inference system. INFless leverages adaptive batching and cold-start mitigation mechanisms to better support serverless inference tasks. Compared with these data parallelism works, FluidEdge adopts an orthogonal view on inference serving, which optimizes the inference bottleneck of ML models with auto-scaling. Other works focus on partitioning the model into several functions [9], [37]. AsyFunc [10] handles the burstiness via asymmetric functions. FluidEdge builds on top of these prestigious works and proposes a fluid operator parallelism that dynamically adjusts the partition against the fixed partition of the above works. Recently, there has been a growing trend in utilizing the serverless paradigm to handle LLM workloads [38]. However, existing works mainly focus on optimizing serverless ML inference on cloud servers, while FluidEdge focuses on serving ML workloads on resource-constrained edge devices.

IX. CONCLUDING REMARKS

This paper presents FluidEdge, a serverless system designed to address the challenges of machine learning (ML) inference on edge SoC devices. By identifying bottlenecks within ML models and implementing a bottleneck-aware scaling technique, FluidEdge facilitates fine-grained resource allocation on the edge, resulting in significant latency and SLO violation reduction. The experimental results show FluidEdge achieves 37.4% average inference latency reduction compared with the best-performed baselines, which showcases its potential to enhance edge-based ML inference capabilities while meeting stringent performance requirements.

REFERENCES

- [1] B. Guo et al., “Concurrent order dispatch for instant delivery with time-constrained actor-critic reinforcement learning,” in *Proc. IEEE Real-Time Syst. Symp.*, 2021, pp. 176–187.
- [2] B. Guo et al., “WePos: Weak-supervised indoor positioning with unlabeled WiFi for on-demand delivery,” in *Proc. ACM Interactive, Mobile, Wearable Ubiquitous Technol.*, vol. 6, no. 2, pp. 1–25, 2022.
- [3] Y. Yang et al., “INFless: A native serverless system for low-latency, high-throughput inference,” in *Proc. 27th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2022, pp. 768–781.
- [4] C. Zhang et al., “MArk: Exploiting cloud services for cost-effective, SLO-Aware machine learning inference serving,” in *Proc. 2019 USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 1049–1062.
- [5] S. Hu, Z. Qu, B. Tang, B. Ye, G. Li, and W. Shi, “Joint service request scheduling and container retention in serverless edge computing for vehicle-infrastructure collaboration,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 6, pp. 6508–6521, Jun. 2024.
- [6] K. Zhao, Z. Zhou, L. Jiao, S. Cai, F. Xu, and X. Chen, “Taming serverless cold start of cloud model inference with edge computing,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 8, pp. 8111–8128, Aug. 2024.
- [7] K. Xiao, S. Yang, F. Li, L. Zhu, X. Chen, and X. Fu, “Making serverless not so cold in edge clouds: A cost-effective online approach,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 9, pp. 8789–8802, Sep. 2024.
- [8] A. Ali, R. Pincirol, F. Yan, and E. Smirni, “BATCH: Machine learning inference serving on serverless platforms with adaptive batching,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2020, pp. 1–15.
- [9] M. Yu, Z. Jiang, H. C. Ng, W. Wang, R. Chen, and B. Li, “Gillis: Serving large neural networks in serverless functions with automatic model partitioning,” in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst.*, 2021, pp. 138–148.
- [10] Q. Pei et al., “AsyFunc: A high-performance and resource-efficient serverless inference system via asymmetric functions,” in *Proc. ACM Symp. Cloud Comput.*, New York, NY, USA, 2023, pp. 324–340.
- [11] J. Jarachanthan et al., “AMPS-inf: Automatic model partitioning for serverless inference with cost efficiency,” in *Proc. 50th Int. Conf. Parallel Process.*, 2021, pp. 1–12.
- [12] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, “GPU scheduling on the NVIDIA TX2: Hidden details revealed,” in *Proc. IEEE Real-Time Syst. Symp.*, 2017, pp. 104–115.
- [13] C. Zhang, F. Zhang, K. Chen, M. Chen, B. He, and X. Du, “EdgeNN: Efficient neural network inference for CPU-GPU integrated edge devices,” in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 1193–1207.
- [14] M. Xu et al., “Efficient, scalable, and sustainable DNN training on SoC-clustered edge servers,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 14344–14360, Dec. 2024.
- [15] L. Zhang et al., “More is different: Prototyping and analyzing a new form of edge server with massive mobile {SoCs},” in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2024, pp. 285–302.
- [16] “Knative project,” 2025. [Online]. Available: <https://knative.dev/>
- [17] C. Min et al., “SensiX: Bringing MLOps and multi-tenant model serving to sensory edge devices,” *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 6, pp. 98:1–98:27, Nov. 2023.
- [18] J. Yi, U. G. Acer, F. Kawsar, and C. Min, “Argus: Enabling cross-camera collaboration for video analytics on distributed smart cameras,” *IEEE Trans. Mobile Comput.*, vol. 24, no. 1, pp. 117–134, Jan. 2025.
- [19] D. Cao et al., “Microedge: A multi-tenant edge cluster system architecture for scalable camera processing,” in *Proc. 23rd ACM/IFIP Int. Middleware Conf.*, 2022, pp. 322–334.
- [20] L. Zhang et al., “High-density mobile cloud gaming on edge {SoC} clusters,” in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2024, pp. 339–347.
- [21] M. Xu et al., “From cloud to edge: A first look at public edge platforms,” in *Proc. 21st ACM Internet Meas. Conf.*, 2021, pp. 37–53.
- [22] “CUDA programming - technical specifications per compute capability,” 2025. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities>
- [23] “NVIDIA Nsight developer tool,” 2025. [Online]. Available: <https://docs.nvidia.com/nsight-graphics/index.html>
- [24] “AWS Lambda,” 2025. [Online]. Available: <https://aws.amazon.com/lambda>
- [25] “Azure function,” 2025. [Online]. Available: <https://azure.microsoft.com/en-us/products/functions/>
- [26] T. Chen et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in *Proc. 13th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 578–594.
- [27] S. S. Shubha and H. Shen, “AdaInf: Data drift adaptive scheduling for accurate and SLO-guaranteed multiple-model inference serving at edge servers,” in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2023, pp. 473–485.
- [28] M. Shahrad et al., “Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider,” in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, pp. 205–218.
- [29] Z. Li et al., “Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing,” in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 69–84.
- [30] A. Glikson et al., “Deviceless edge computing: Extending serverless computing to the edge of the network,” in *Proc. 10th ACM Int. Syst. Storage Conf.*, New York, NY, USA, 2017, Art. no. 1.
- [31] P. K. Gadepalli et al., “Sledge: A serverless-first, light-weight wasm runtime for the edge,” in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 265–279.
- [32] V. Mittal et al., “Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds,” in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 168–181.
- [33] Y. Li, D. Zeng, L. Gu, M. Ou, and Q. Chen, “On efficient Zygote container planning toward fast function startup in serverless edge cloud,” in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–9.
- [34] Z. Wen et al., “StepConf: SLO-aware dynamic resource configuration for serverless function workflows,” in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1868–1877.

- [35] A. Chen et al., “Opara: Exploiting operator parallelism for expediting DNN inference on GPUs,” *IEEE Trans. Comput.*, vol. 74, no. 1, pp. 325–333, Jan. 2025.
- [36] A. Ali et al., “Optimizing inference serving on serverless platforms,” in *Proc. VLDB Endowment*, vol. 15, no. 10, pp. 2071–2084, 2022.
- [37] H. Li, X. Li, Q. Fan, Q. He, X. Wang, and V. C. M. Leung, “Distributed DNN inference with fine-grained model partitioning in mobile edge computing networks,” *IEEE Trans. Mobile Comput.*, vol. 23, no. 10, pp. 9060–9074, Oct. 2024.
- [38] Y. Fu et al., “ServerlessLLM: Locality-enhanced serverless inference for large language models,” in *Proc. 18th USENIX Conf. Operating Syst. Des. Implementation*, 2024, pp. 135–153.



Borui Li (Member, IEEE) received the PhD degree in computer science from Zhejiang University, China, in 2022. He is now an assistant professor with the School of Computer Science and Engineering, Southeast University. His research interests include edge computing and deviceless computing. He has published research papers in prestigious conferences and journals, including NSDI, MobiCom, ATC, TC, and TOSN. He is a member of the ACM.



Shuai Wang received the PhD degree from the Department of Computer Science, George Mason University (GMU). He is currently an assistant professor in the School of Computer Science and Engineering with Southeast University, in China. He is broadly interested in wireless sensing, mobile computing, wireless communication, IoT, and embedded systems for improving wireless connectivity. He won the Best Paper Award with ICDCS, 2018.



Chenhong Cao received the BS and MS degrees in computer science from Northeastern University, China, in 2011 and 2013, respectively, and the PhD degree from Zhejiang University, in 2018. She is currently an associate researcher with the University of Science and Technology of China. Her research interests include backscatter communications, network measurement, mobile computing, and the Internet of Things.



Tiange Xia received the BS degree from the School of Electronic Science and Engineering, Southeast University. He is currently working toward the MASc degree with the School of Computer Science and Engineering, Southeast University. His research interests include cloud native, serverless systems, and edge computing.



Zheng Dong (Member, IEEE) received the PhD degree from the Department of Computer Science, University of Texas at Dallas, in 2019. He is an assistant professor in the Department of Computer Science with Wayne State University. His research interests include real-time cyber physical systems and mobile edge computing. He received the Outstanding Paper Award with the 38th IEEE RTSS.



Weilong Wang received the MS degree from the School of Computer and Control Engineering, Yantai University. He is currently working toward the PhD degree with the School of Computer Science and Engineering, Southeast University. His research interests include serverless computing, WebAssembly, and edge computing.



Jingyuan Zhang received the PhD degree from the DS2 research lab, George Mason University, where he worked with Prof. Yue Cheng. His research interests are serverless computing and its optimization for stateful applications, e.g., data analytics tasks. He has published research papers on serverless computing in prestigious conferences including VLDB, FAST, ASPLOS, and SoCC.



Shuai Wang (Member, IEEE) received the BS and MS degrees from the Huazhong University of Science and Technology, China and the PhD degree from the Department of Computer Science and Engineering, University of Minnesota, in 2017. He is currently a professor with the School of Computer Science and Engineering, Southeast University. His research interests include the Wireless Networks and Sensors, Cyber-physical Systems, Data Science.