The Journal of Supercomputing (2021) 77:8593–8621
https://doi.org/10.1007/s11227-021-03623-9

GPU-based embedded edge server configuration and offloading for a neural network service

JooHwan Kim[1] · **Shan Ullah**[1] · **Deok-Hwan Kim**[1]

Accepted: 5 January 2021 / Published online: 25 January 2021

Abstract

Recently, emerging edge computing technology has been proposed as a new paradigm that compensates for the disadvantages of the current cloud computing. In particular, edge computing is used for service applications with low latency while using local data. For this emerging technology, a neural network approach is required to run large-scale machine learning on edge servers. In this paper, we propose a pod allocation method by adding various graphics processing unit (GPU) resources to increase the efficiency of a Kubernetes-based edge server configuration using a GPU-based embedded board and a TensorFlow-based neural network service application. As a result of experiments performed on the proposed edge server, the following are inferred: 1) The bandwidth, according to the time and data size, changes in local (20.4–42.4 Mbps) and Internet environments (6.31–25.5 Mbps) for service applications. 2) When two neural network applications are run on an edge server consisted with Xavier, TX2 and Nano, the network times of the object detection application are from 112.2 ms (Xavier) to 515.8 ms (Nano); the network times of the driver profiling application are from 321.8 ms (Xavier) to 495.7 ms (Nano). 3) The proposed pod allocation method demonstrates better performance than the default pod allocation method. We observe that the number of allocatable pods on three worker nodes increases from five to seven, and compared to other papers, the proposed offloading shows similar or better response times in environments where multiple deep learning applications are implemented.

Keywords Edge server · Kubernetes · Neural network service · GPU-based container · Pod allocation

✉ Deok-Hwan Kim
deokhwan@inha.ac.kr

JooHwan Kim
22191434@inha.edu

Shan Ullah
shan.ullah@iesl.inha.ac.kr

[1] Department of Electronic Engineering, Inha University, Incheon, South Korea

1 Introduction

Recently, interest in artificial intelligence (AI) has been increasing in various fields. Among these, the Internet of Things (IoT) is a domain where the need for AI is becoming more important as the variety and size of data available to users and the environment increases. Currently, AI services in IoT devices are provided via a central cloud over the Internet; therefore, these AI services are affected by factors that influence the performance and management of cloud services, including low latency, ease of development, database queries, and scaling for resource utilization [1]. Cloud companies are now considering new cloud models that work in a variety of ways to handle the massive amounts of data needed for AI models. Among these cloud models, edge computing is discussed as a new processing method that can compensate for the disadvantages of the central cloud structure [2]. The research firm Gartner in the USA reported autonomous edges as one of the top 10 strategic technologies announced in 2020, whereas IBM selected edge computing and leveraging Kubernetes as methodologies to use for network evolution in 2020 [3, 4]. Edge computing, considered the core of the next-generation cloud, refers to a paradigm that uses real-time data based on the needs of the surrounding environment and users rather than using a traditional central server to process all data. This technology is based on computing devices in edge-located areas called micro-data centers, cloudlets, and fog that are adjacent to the user plane. Its advantages compared to traditional clouds are low latency, traffic distribution, and protection of data privacy [5, 6].

Research related to edge computing has been conducted from various perspectives. Figure 1 shows the structure of an edge computing environment. A three-layer (user plane, edge computing plane, and cloud computing plane) structure that

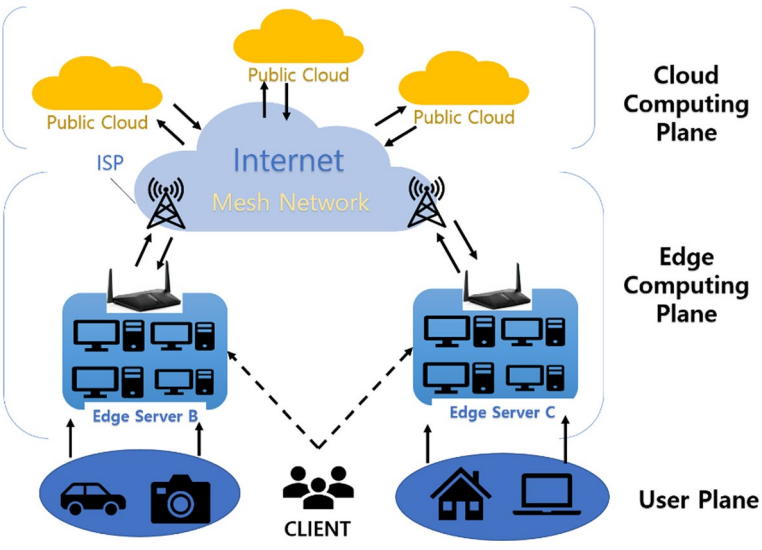

Fig. 1 Edge Server Environment

includes cloud, mobile edge computing (MEC), and IoT has been proposed [7]. In addition, studies have been conducted on the proposed use of an edge computing solution to configure services, contents, and functions in an area close to the end user [8] including a hierarchical MEC structure composed of components termed field, shallow, and deep cloudlets to increase service efficiency [9]. Regarding the edge computing configuration problem in using transparent computing, a new approach [10], Nebula, which is an edge computing platform configuration with computation and storage interaction that supports grid and peer-to-peer systems, has been proposed [11].

The planning management of edge computing, capacity planning that allows resources for CPUs and GPUs to meet quality of service (QoS) requirements [12], average server utilization, and latency of application deployment to MEC, has been the subject of research. In addition, the relationship [13] and the efficacy of dynamic computing web content provision for edge computing [14] have been studied.

Regarding resource management and the provisioning of edge computing, the dynamic service provisioning of edge clouds via the computation of new service load costs, limited node capacity, and the delivery process request costs on nodes with limited resources has been reported on in terms of resource management [15]. Migration using the Markov decision process (MDP) is based on responses such as user movement and network performance [16].

Furthermore, several studies have been conducted on various testbed types based on previous research. These studies include an evaluation of edge-hosted containers [17] and three edge-cloud implementations of the containerized Platform as a Service (PaaS) structure in a cluster utilizing Raspberry Pi, which is a single-board computer [18–20]. A comparative study was conducted of the performance of machine learning packages, including TensorFlow, Caffe2, MXNet, PyTorch, and TensorFlow Lite, on an edge device [21]. However, because single boards, such as the Raspberry Pi, have limitations in terms of hardware specifications for neural network implementation, edge computing must be implemented on a single-board computer (ARM core-based CPU) that includes a GPU (Nvidia GPU). Moreover, to use the GPU resources of the cluster for containers, the specifications required by the neural network service to monitor the GPU resources must be checked at each edge node., The container environment for edge devices was not considered by these techniques.

In this study, we propose an embedded edge server on a single board to process simultaneously multiple deep learning models composed of a Kubernetes-based container environment.

The contributions of this study are as follows: First, it enables independent environment configuration and resource sharing for each neural network service. Container-based deep learning applications support an independent operating environment at a high level; this solves the dependency problem of rapidly changing deep learning libraries. At the lower level, the application can be easily deployed to and managed for various users by sharing resources at the kernel level. Second, it supports scheduling to increase resource utilization for deep learning services. When a deep learning service executes, it does so multiple times on the nodes in the cluster, and the efficiency of resources is increased by optimizing their utilization using the

scheduling of the cluster system. Finally, an offloading method between the edge and the central cloud is proposed according to the difference in the amount of computation on the neural network. Depending on the state of the edge server, the computing location where additional deep learning services are executed is offloaded to the central cloud. Accordingly, it is possible to change the computing resources used for additional requests dynamically and to determine the computing location of the deep learning service by reconfirming the different delay times according to the varying amounts of computation.

The remainder of this paper is organized as follows. In Sect. 2, the network for the edge server and the container-based environment of the board, including the GPU task, are described. Section 3 includes the structure of the implemented edge server, the setting of GPU support in the container environment, the neural network model used in the edge server, the offloading, and the scheduling method with nodes in the cluster for deep learning. In Sect. 4, the detailed experimental results of the proposed scheduling and offloading method on a single-board edge server with a GPU are described. Finally, Sect. 5 presents the conclusion.

2 Background and motivation

2.1 Network environment change

In emerging 5G networks, software networking is a programmable approach that makes extensive use of IT virtualization technologies, such as communication infrastructure, functions, and applications. Therefore, edge computing is a core technology and architecture concept that enables the evolution of 5G [22]. Changes in the network are directly related to the performance of edge computing in terms of low latency, speed, and structure. This emerging technology supports the stability of a service; however, the stability is affected by networking problems at each node in a cluster-type edge server. In a previous study, we implemented edge servers on various boards to check for stability problems and latency at the user plane [23].

2.2 Container

A container is a virtualization method for running a separate virtual Linux system (container) on Linux [24]. In the container, layers composed of independent file systems are connected as a single image; the layers use the union file system to store the environment. Hardware resources are used separately for each container. Using the saved image, we can run a containerized application. Figure 2 shows the steps for building an image and the process of running a containerized application. In this process, Linux container technology easily sets changes to the environment according to the container runtime and is a method for implementing micro-services composed of small units.

The advantages of using containers in edge servers are as follows: First, because of server hardware limitations, a hypervisor program has a considerable size

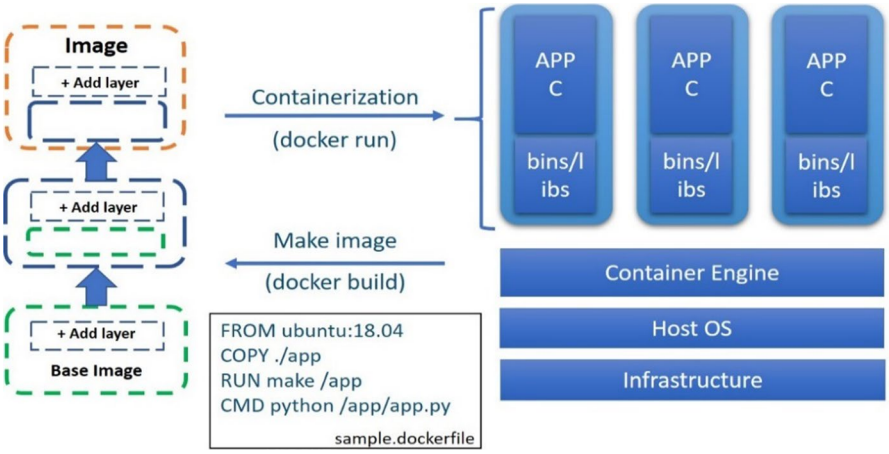

Fig. 2 Image Building and Containerization

constraint, whereas a container environment allows easy basic configuration of programs, such as light memory usage and quick starting. Second, flexible scaling is efficient because resource management is handled at the OS kernel. Figure 3 shows the container runtime interfaces (CRIs) based on the open container initiative (OCI), which standardizes the runtime that supports the container environment.

There are a number of CRIs including CRI-O, Docker, and gVisor. Among these, Docker is most widely used because it is the most mature. However, a CRI focuses on managing the container configuration in a host environment. In a large-scale cluster server configuration, efficient work management in terms of functionality is lacking in CRIs. Therefore, a platform that provides additional container management is required.

Footprint	Docker Engine (native)	gVisor CRI plugin	CRI-O Kata Containers
sponsors	Docker Inc	Google	Intel
started	Mar 2013	2015	2017
version	18.06	runc	1.3
runtime	runc	runsc	kata-runtime
kernel	shared	partially shared	isolated
syscall filtering	no	yes	no
kernel blobs	no	no	yes
footprint	-	-	30mb
start time	<10ms	<10ms	<100ms
io performance	host performance	slow	host performance
network performance	host performance	slow (see comment)	close to host performance

Fig. 3 CRI Comparison [25]

2.3 Container orchestration

Containers are gradually becoming a base environment for DevOps, a way to unify software development and operations [26]. Concurrently, container orchestration contributes to managing hardware resources between each terminal by stably updating and distributing work in a large-scale distributed computing context, such as the cloud. Therefore, the need for a container orchestration platform has increased.

Figure 4 shows the types of container orchestration. Container orchestration types include the Docker swarm, Kubernetes, and Apache Mesos, with Kubernetes currently being the most used and systematically mature [27–29]. This is because of the following reasons. First, Kubernetes can be constructed in a variety of environments, unlike other platforms. Second, it has advantages regarding scheduling, deployment, and managing containers based on user options. Third, it also has advantages regarding scheduling service tasks in various ways and utilizing the resources applied to these tasks [30]. Kubernetes is an application that runs on a node. It uses a set of pods for deployment, and a job- and daemon-set for the update operation. In addition, Kubernetes can use separate virtual networks internally to support container-specific environments and uses an API server that supports external access from the master node to operate separate service applications according to client requests. At this time, the internal network checks the pod connection according to each request

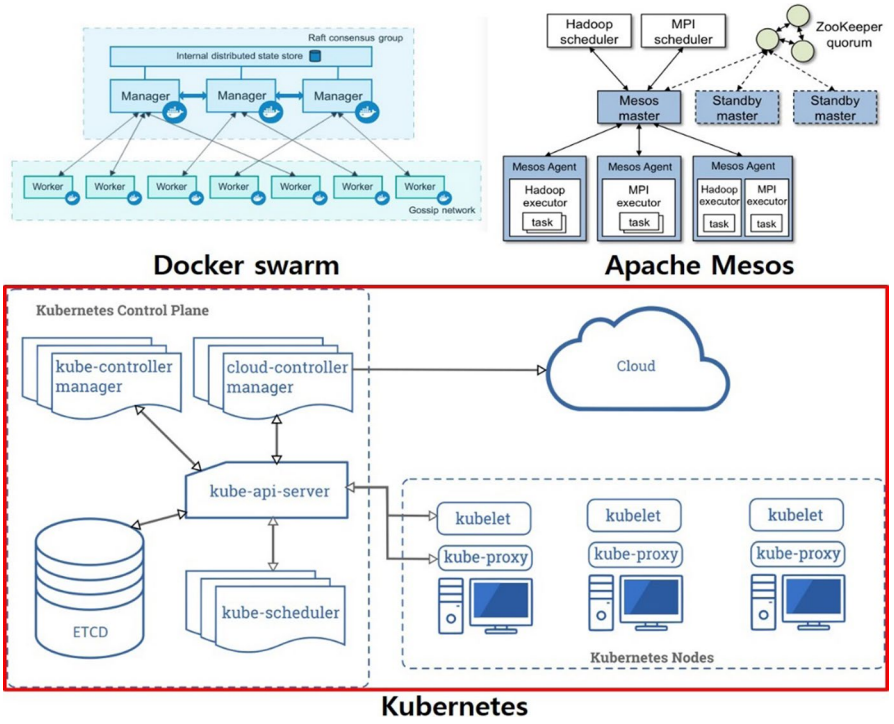

Fig. 4 Container Orchestration

through the operation of Kube-proxy and processes the connection of the pod using a distribution algorithm.

2.4 GPU resource availability in containers

Edge computing is mainly used for preprocessing tasks in nearby computing areas because of the difficulty of integrating each computing resource to the center cloud. Furthermore, with the emergence of boards with GPU resources available in the IoT area, such as Nvidia's Jetson Board, the utilization of GPU resources in edge computing is increasingly being considered. For example, Nvidia provides an inference framework, such as the DeepStream SDK, that simplifies the development of video analytics applications on Nvidia platforms by providing TensorRT [31].

Thus, because of GPU resources being utilized on edge servers, this framework has an advantage over central processing units (CPUs) in artificial neural network processing that mainly process via parallel computation and handle media data, such as videos and photos, rather than raw data from sensors. While reducing traffic to the central cloud, concurrently we must achieve rapid response and fast processing. NVIDIA, a leading GPU developer, is releasing a runtime called Nvidia-docker to GitHub to help manage Docker-based GPU resources and cope with the changes in the container environment [32]. Development tools and neural network models for AI on various platforms are being supported as container images through Nvidia-GPU Cloud (NGC) [33].

3 Proposed work

In this study, we addressed research questions dealing with the configuration of an edge cluster to enhance GPU resource utilization and allocate pods to proper nodes for deep learning services. We configured a native clustering edge server in a wireless environment using Kubernetes, which supports container applications. We also confirmed the implementation of the edge server for utilizing GPU resources, pod scheduling, and offloading methods. First, we propose an embedded edge server structure in a container environment for target applications.

3.1 Edge server structure

IoT data sources in each area of the edge server environment require improved hardware performance to preprocess relatively large media data (pictures, voices, etc.), and reliable transmission methods and data formats. In a previous study, we implemented a transport stream socket using stored photograph data. This paper proposes a structure that collects photograph data from a camera connected to an IoT device and implements a format using *GStreamer*. It puts the data in a basic transmission queue, connects to a stream socket, and transmits it.

The proposed edge server area includes high-performance processing using a neural network rather than simple data preprocessing. It is responsible for storing

the resulting data and transmitting it to the cloud computing plane. Furthermore, to determine the usefulness and usability of scaling, the operation is confirmed by implementing a workload in which multiple containers are operated according to additional client transmissions. Figure 5 shows the structure for realizing the proposed edge server. It uses Docker as the CRI of all nodes and adds Kubernetes to cluster and orchestrate each node for operating the GPU-enabled application requested from the edge server.

3.2 Proposed new setting for neural network

3.2.1 New extended resource for GPU board

We proposed a new edge server for the implementation of a neural network service in the edge server to utilize the GPU. For neural networks with higher computational requirements, computations are usually performed faster at the central cloud computing layer. However, if the neural network is lightweight with no significant accuracy difference in the user's perception, the small edge server provides a latency benefit by servicing the corresponding processes in a region closer to it. These edge servers require GPU resources to ensure fast inference times. However, to utilize the GPU in a typical container environment, it must be recognizable by mounting the graphics driver using the Nvidia-Docker runtime [32]. However, the existing Kubernetes scheduler distributes to pods by monitoring only the edge server's CPU and RAM resources. An application running a neural network model requires a method for identifying GPU resources and assigning a pod to an appropriate node. The desktop environment (×86 chipset) uses a k8s device plug-in; however, it does not work on the Jetson board (ARM64 chipset,) because support is lacking from the Nvidia

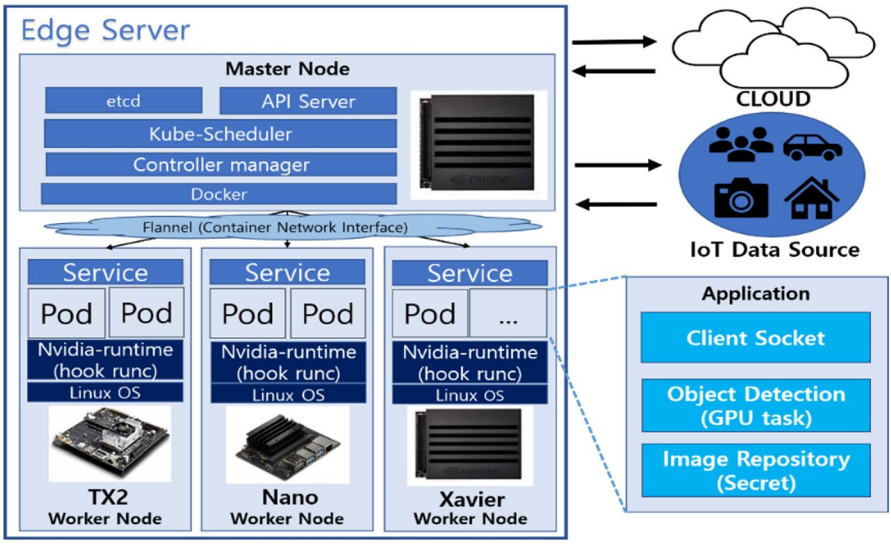

Fig. 5 Proposed Edge Network

Management Library (NVML). In addition, this plug-in is for virtualized GPU, and it is difficult to determine the optimal resource efficiency when the actual GPU is allocated to each node.

Therefore, we propose a new scheduling method to consider the maximum GPU clock and add it as an extended resource together with CPU and RAM capacity. This method will allocate pods to the appropriate nodes (Jetson board) and share GPU resources between containers for deep learning applications. The process was implemented using a device query, as shown in Fig. 6. In addition, on Jetson series boards with different hardware specifications configured inside an edge server, the neural network applications' scalability can be increased while increasing work efficiency.

3.2.2 Neural network models for proposed edge server

To evaluate and verify the proposed edge server configuration for deep learning services, we deployed a diverse range of neural networks for entirely different applications as follows:

1. Object detection using SSD-MobileNetv2 [34, 35]
   - Based on 2D images
2. Driver behavior profiling using DeepConvRNN [36]
   - Based on 1D scalar data (driver data, i.e., acceleration, braking, etc.)

The reason behind selecting an algorithm for the proposed work was to generate a real-time environment scenario, where an image from the client's camera would require deep learning services (object detection, in this case) from worker nodes under the proposed research configuration. Similarly, in the case of 1D scalar data, we assume a connected car environment, where the sensor data from an in-vehicle controller area network (CAN) bus would require services from the proposed edge server configuration. In this regard, through deep learning services, the edge server would detect the identity of the driver via driver behavior profiling.

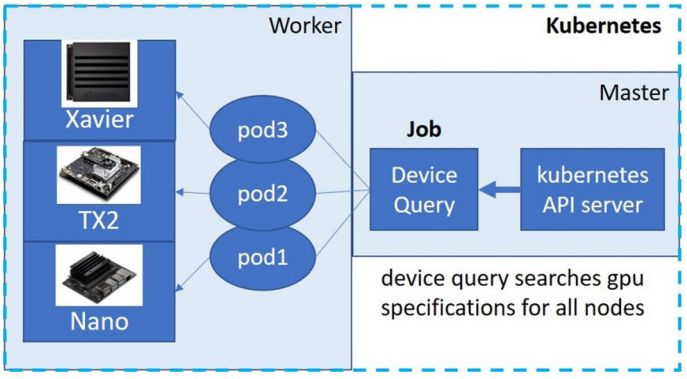

Fig. 6 Device Query

Before deploying the selected deep learning algorithms in our proposed configuration, all worker nodes (embedded hardware) were benchmarked to evaluate resource utilization [44]. In addition, the computational complexity of several driver profiling models in a container environment (without Kubernetes) was studied in [45], wherein a lightweight model was proposed using a sparse learning technique for a container environment. However, this study focuses on offloading deep learning services using the proposed configuration (pod allocation) in the Kubernetes environment (edge computing). In this regard, the selected deep learning models are explained briefly to describe the levels of complexity of the algorithms used in the proposed work.

For object detection, a well-known deep learning algorithm named SSD-Mobilenetv2, which consists of two models, a single-shot multi-box detector (SSD) [34] and MobilNetv2 [35], is deployed. MobileNetv2 is used as a feature extractor and achieves competitive accuracy with significantly fewer parameters (4.3 million). It requires lower computational complexity when combined with an optimized version of SSD [35], together named SSD-MobileNetv2. Recently, SSD-MobileNetv2 has become available for the Jetson series (Xavier, TX1, TX2, and Nano) in the Jetson inference library, where it is highly optimized using TensorRT. However, in our case, we deployed the frozen model available on the official TensorFlow GitHub page. This model is pretrained on the COCO [37] dataset and can be modified easily using TensorFlow for development, as compared to that available in the Jetson inference library. Further details of execution under the proposed configuration are explained in Sect. 4.3.1.

Moreover, we implemented the algorithm proposed in [36] and modified it based on the proposed configuration to a lightweight deep learning model for the container environment under the umbrella of edge computing. It is based on a famous multimodel network that comprises a convolutional layer followed by a recurrent neural network. According to the container environment, we require a compact network with fewer parameters and a memory image. In this regard, we further optimized the network [36] by tuning parameters such as kernel size, kernel depth, the window stride, batch size, and the number of hidden layers for LSTM; we also dropped the last attention unit. We successfully reduced the size of the network by compromising a degree of accuracy. The final configuration contains the kernel size (1, 20) of the first depthwise convolution layer with a depth multiplier of 20, followed by a maxpool layer with a kernel size of (1, 53) with a stride value of (1, 2). The second depthwise convolution layer has a kernel size of (10, 1) with a depth multiplier of 10. The architecture [36], explained in Fig. 7 exploits the Ocslab [38] driving dataset for driver profiling and identification. In the Ocslab driving dataset, 51 driving features were acquired using the in-vehicle CAN data bus. However, for driver identification, 15 shortlisted features that significantly corresponded to the personal skills of a driver were used. The selected features were related to the engine (engine coolant temperature, engine torque, friction torque, etc.), fuel (intake air pressure, fuel consumption, etc.), and transmission (transmission oil temperature, wheel velocity, torque convertor speed, etc.). These 15 properties were processed further with statistical (mean, median, and standard deviation) features, creating 45-dimensional features (15×3). We implemented the algorithm using TensorFlow 1.15, in a container

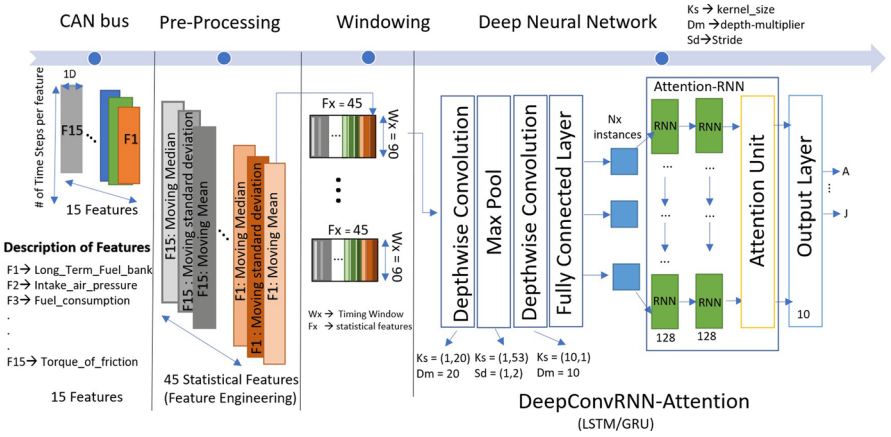

Fig. 7 DeepConvRNN Used in [36]

environment using a Docker image. Further details on the execution are provided in Sect. 4.3.1. The reason for explaining Fig. 7 is that, unlike SSD-MobileNetv2, driver profiling is not part of the Jetson inference library. For our scenario, we evaluated different algorithms, selected this tool for driver profiling, and further modified it according to the proposed configuration of the edge server deep learning services.

Figure 7 shows the operation of the application according to the data flow process. In addition, the assumed neural network model that performs object detection in the application uses a pretrained model that can obtain the SSD-MobileNetv2 model trained with the COCO dataset in the TensorFlow object detection API [39].

3.3 Scheduling and offloading for GPU resources

The structure is based on a container environment using Kubernetes, and the entire process for the edge server to run the neural network as follows is illustrated in Fig. 8.

Step 1 First, the device query application is executed with the preset settings of all edge servers and the update of the extended resource for the GPU specification is sent to the API server as an HTTP patch request and registered to all nodes.

Step 2 The client near the edge server requests the desired service from the API server in a small local edge. The program that communicates each request of the client uses the Kubernetes API that supports libraries for authentication to access the cluster and internal cloud operation.

Step 3 After confirming the request, the edge server operates a data-receiving server connected to the IoT data source. The transmitted data include a large media file (e.g., picture, video formatting file, etc.) and scalar data (e.g., sensor data, single value data, etc.) and are transmitted to an edge server using a stream socket from a data source.

Step 4. The Kube-scheduler included in Kubernetes is then used for scheduling the node assignment of the neural network-driven pod. Then, the newly added GPU

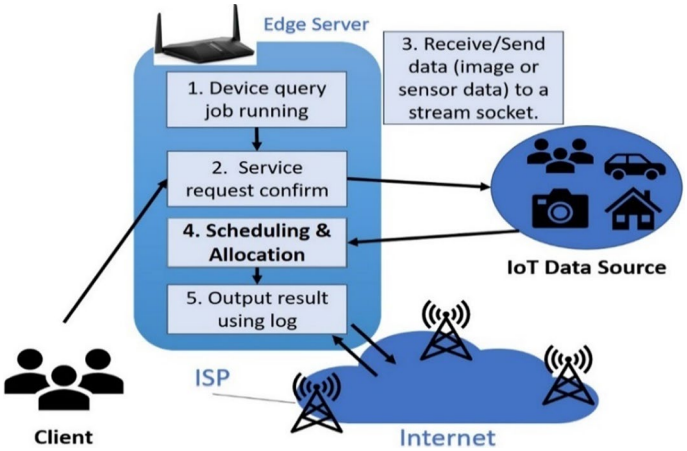

Fig. 8 Application progress

resource is set as a new filtering condition in the configuration file of the neural network-driven pod and the pod is allocated.

Step 5. The assigned pod processes the external data through the neural network. As the same communication is used, given that data retransmission affects the performance of neural network pods that are processed, the edge server uses a separate retransmission pod or monitoring pod inside the edge server via authentication from the outside.

In Kubernetes, basic scheduling is divided into two stages: predicate scheduling and priority scheduling. Predicate scheduling divides the executable and pending pods by comparing the computing resources required for the pod with the computing resources of all nodes. The priority scheduling process changes the priority according to the pod task conditions and the resource limit, affinity, and user factor. The proposed method is implemented by adding a user factor and affinity to intervene in the priority scheduling process and setting a factor for the GPU's performance:

$$\mathrm{CPU}_{\mathrm{nodetotal}} = 80\% * \sum_{i=1}^{N} \mathrm{CPU}_{\mathrm{no}_i} > \mathrm{CPU}_{\mathrm{req}} = \sum_{g=1}^{M} CPU_{\mathrm{pod}_g}$$

$$\mathrm{MEM}_{\mathrm{nodetotal}} = 80\% * \sum_{i=1}^{N} \mathrm{MEM}_{\mathrm{no}_i} > \mathrm{MEM}_{\mathrm{req}} = \sum_{g=1}^{M} \mathrm{MEM}_{\mathrm{pod}_g}$$

$$S_1\left(\mathrm{pod}_g, \mathrm{Cluster}\right) = \left\{ \mathrm{no}_i | \left(\mathrm{CPU}_{\mathrm{no}_i} * 80\% - \mathrm{CPU}_{\mathrm{pod}_g} \right) \geq 0 \right\}$$

$$S_2\left(\mathrm{pod}_g, \mathrm{Cluster}\right) = \left\{ \mathrm{no}_i | \left(\mathrm{MEM}_{\mathrm{no}_i} * 80\% - \mathrm{MEM}_{\mathrm{pod}_g} \right) \geq 0 \right\}$$

We assume that the total amount of CPU and memory is $\mathrm{CPU}_{\mathrm{nodetotal}}$ and $\mathrm{MEM}_{\mathrm{nodetotal}}$, the node's CPU usage and memory usage are $\mathrm{CPU}_{\mathrm{no}_i}$ and $\mathrm{MEM}_{\mathrm{no}_i}$, and the CPU and memory requests to the pod for execution are $\mathrm{CPU}_{\mathrm{pod}_g}$ and $\mathrm{MEM}_{\mathrm{pod}_g}$ respectively. In addition, the total number of nodes is N, and the total number of pods is M. Equations (1) and (2) are conditional expressions that compare the total resources required by all pod_g running against the entire cluster's CPU/RAM. The total available resources should be limited to the host's required resources, which is limited to 80%. Equations (3) and (4) define the predicate functions S_1 and S_2 to place pod_g on the no_i node that provides sufficient CPU and memory resources. Subsets S_1 and S_2 are the sets of nodes that can be obtained for each node resource as a result; we select the collection of nodes S_1 and S_2 with sufficient resources for pending pods requesting different resources from the CPU and MEM. Before the utilization of various resources of the system reaches full load, the system throughput increases concurrently with resource usage.

Next, the priority-scheduling process determines the priority between the adjusted nodes by applying the user factor and resource limit that are additional priority factors. Through adjustment during this process, the scheduling of the cluster server can be determined to suit the purpose. Equation (5) is a formula that determines the final priority value of a pod executed for a node. The priority function (P_i) is a function determined for affinity, the resource limit, and user factor; it is calculated by multiplying the additional weights for each node and priority function. In the equation, u denotes the number of priority functions to operate and no_i denotes a node for which the priority is set:

$$P\left(\mathrm{pod}_g,\ \mathrm{no}_k\right) = \sum_{i=1}^{u} \left\{\mathrm{weight}_{(i,k)} * P_i\left(\mathrm{no}_k\right)\right\}$$

The method proposed for factoring uses the value of the GPU clocks and graphic memory obtained from the device query; it does not depend on the presence/absence of GPUs or the number of GPUs. This factor allows differences in performance to be shown. Because the proposed factor can be used to represent the relative GPU performance difference for all individual board nodes, basic scheduling based on the distribution of pods can be operated appropriately for neural networks. Therefore, the proposed factor is set to have a priority value of 0, indicating the most preferred state for a node that is faster in inference. This factor affects the priority function of the *BalancedResourceAllocation* method according to differences in the GPU graphics memory allowances and operates in the priority function of the *NodeAffinityPriority* method according to differences in GPU performance.

Algorithm 1. Pseudocode for the Scheduling Process

```
for each pod in Cluster do
    if node.CPU_nodetotal < pod.CPU_req then
        is_valid = false
    end if
    if is_valid AND (node.MEM_nodetotal < pod.MEM_req ) then
        is_valid = false
    end if
    if not is_valid then
        break
    end if
    for each node in Cluster do
        if node.CPU < pod.CPU then
            is_valid = false
        end if
        if node.MEM < pod.MEM then
            is_valid = false
        end if
        if not is_valid then
            break
        end if
        Add node to the valid node list

        for each priority_function do
            node.priority+= P_i.weight * node.P_i
        end for
        highest_node := highest priority node in node.priority
    end for
execute pod in highest_node
end for
```

One of the contributions of the proposed scheme is to introduce integral offloading to the central data center if the edge is not adaptable to a given neural network service. The point to be considered for offloading is the total latency, including the network latency and neural network processing time for each of the central cloud and edge servers. Initially, in the existing "end–edge" structure, offloading is determined between the computing device located in the IoT data source area containing the data and the surrounding edge server's devices. However, in the request for a neural network service, a comparison of offloading times in the "edge–cloud" area is required when there are insufficient local computing resources. The sum of the time (T_{centerup}, T_{edgeup}) required for total data transmission in each area and the execution time (T_{centerup}, T_{edgeup}) represented by computing device performance can be viewed as the total time required for processing in each area. It is expressed by Eq. (6) and (7), respectively:

$$T_{\text{offloading}} = T_{\text{centerup}} + T_{\text{centerexe}}$$

$$T_{\text{edge}} = T_{\text{edgeup}} + T_{\text{edgeexe}}$$

In this case, assuming that the neural network inference time at the edge including the GPU does not occupy much of the total delay time compared to the network delay time caused by the data transmission condition, the condition that requires offloading is as shown in Eq. 8:

$$T_{\text{centerup}} \gg T_{\text{centerexxe}} \wedge T_{\text{edgeup}} \gg T_{\text{edgeexe}}$$

As the difference due to the physical distance of the network is the main cause of the delay time in the edge computing system, it can be confirmed that offloading is necessary only when $T_{\text{centerup}} > T_{\text{edgeup}}$ is always satisfied and the edge server accepts the neural network service at its maximum. Conversely, to satisfy the offloading condition under the assumption that the neural network inference time is greater than the network delay time, the time for the neural network operation must first be checked and reflected in offloading.

In the first case, if the edge server does not have enough computing resources to perform the requested service, the requested service is kept on hold, resulting in working time loss. Therefore, with a structure using integral offloading, the computing work is entrusted directly to the central cloud. This has the advantage of service availability regardless of the neural network operating time. For the second case, the trade-off must be found between the neural network execution time and the network delay time that can be verified by the experimental results of integral offloading.

4 Experimental results

4.1 Experimental environment

The purpose of this experiment was to check the efficiency of the proposed provisioning of edge server bandwidth and resources, such as CPU, memory, and GPU for deep learning services. The experiments were conducted in the following order.

In the network performance experiment, we measured the bandwidth and server delay of the client-edge server according to the network environment and data size. In the GPU job allocation and scaling experiments, we measured each neural network model's inference time in the GPU container environment and confirmed the number of operated pods and response times using the proposed pod allocations.

4.1.1 Hardware specification

Experiments were conducted to implement and check the scalability of GPU operations based on the previously proposed small-scale edge server. The specifications of the hardware equipment as well as the clear formats for data used in the edge servers are described. Each cluster node of the edge server at the center of implementation

Table 1 Hardware specifications

Jetson series Hardware specifications			
Device	Jetson Nano	Jetson TX2	Jetson Xavier
CPU	4-core ARM cortex-57	4-core ARM A57 2 MB L2 + HMP Dual Denver 2/2 MB L2	8-core ARMv8.2 8 MB L2 + 4 MB L3
RAM	4 GB LPDDR4	8 GB LPDDR4	16 GB LPDDR4
GPU	128-core Maxell	256-core Pascal	512-core Volta with Tensor Core
Network module	Intel AC9560	Intel AC9560	Intel AC9560, AGW 200

Table 2 Software and network environment

Software specifications	
OS/Kernel	Linux Ubuntu 18.04, Tegra 4.9
Kubernetes	Kubernetes ver 1.16 (kubeadm, kubectl kubelet)
Docker	18.09.7
Router	802.11ax Dual Band Wi-Fi 5 GHz AX: 2×2 (Tx/Rx) 1024 QAM 160/80/40/20 MHz, Maximum 2400 Mbps
TensorFlow	1.15.0
OpenCV	3.4.6

used Jetson boards from Nvidia with different hardware specifications to determine the possibility of utilization according to the GPU's specifications. Table 1 lists the specifications. Minimization of the network overhead through wireless communication was effected via the AC9560 driver used as a back-ported communication adapter for the board. The local network of the edge server was a local wireless network; it was connected through an edge router that was used for environment setting and managing the local network.

4.1.2 Software specification

Table 2 lists the versions of the software used for configuring the edge server. During the experiment, the data used in the GPU work were transmitted from the camera of the device mimicking an IoT data source, and the transmission data format was adjusted for operational reliability in object detection. To implement the neural network within the edge server, graphics hardware and a model implementation IDE were required to utilize the GPU resources, but because the chipset specifications of the Jetson Series are different from the general PC specifications (Jetson series: ARM, Desktop: AMD64), some of the official container images

supported by Nvidia-GPU Cloud (NGC) [32] may not work properly. For compatibility, therefore, an individual container environment was implemented as an image file and used as a base image for the application running in the experiment.

4.2 Evaluation of network performance

As a preliminary experiment to check the configuration of the edge server, the network speed was measured between the edge computing plane and the user plane. To show the differences in the network environment, we tested two network types as a control group. Network type 1 had a connection environment through an Internet network, assuming a cloud computing connection. Network type 2 had a connection environment through the user plane (local wireless network) on the same router as an edge server. To check the speed and bandwidth, iPerf3 [40] was used as the network measurement program. For all nodes in the proposed edge server, Kubernetes service configuration, deployment and NodePort types of service were added. This was done for iPerf3 to run as a server mode pod while the pod was configured to be accessible to the internal cluster from the outside. The test period for each device was 30 s, and throughout the test period, the maximum bandwidth was measured at 1 s intervals for comparative evaluation. In addition, in the same environment, we set the transmitted data to 5, 10, and 15 MB to check the traffic usage according to the size difference.

As shown in Fig. 9 and Table 3, the response of the client connecting the original complex network was obviously slower. These experimental results also show

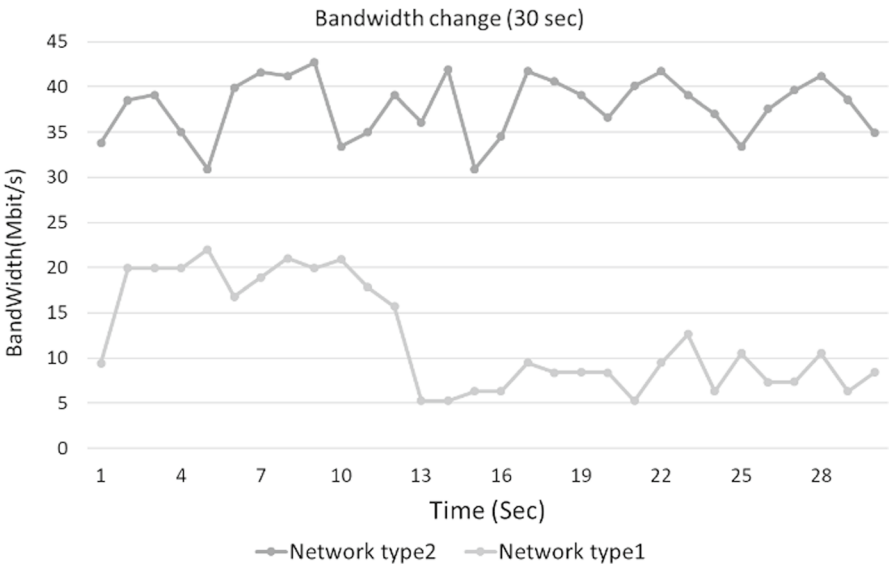

Fig. 9 Bandwidth According to Two Different Network Types

Table 3 Bandwidth changes according to various data sizes

Network	Network type2 (local)			Network type1 (Internet)		
Transmitted data size	5 MB	10 MB	15 MB	5 MB	10 MB	15 MB
Average bandwidth (Mbps)	30.25	34.29	37.43	11.94	12.21	13.46
Max	35.9	41.2	42.4	25.5	23.5	18.1
Min	23.2	20.4	29.6	5.74	6.31	10.1

that an object detection model at an edge server can be more efficient in terms of bandwidth than a central cloud model serviced in the Internet environment.

4.3 Efficiency evaluation for GPU task-based scheduling

The purpose of the second experiment was to demonstrate the efficiency of the proposed scheduling of the GPU task application, using the extended resources of the edge server and the offloading method, compared with the basic scheduling method.

The experiment was divided into two cases: 1) when the basic scheduling method was used, and 2) when the scheduling method used the extended resource and additional affinity. The GPU job application was executed using two neural networks (object detection and driver profiling) with different utilization data. The first was the SSD-MobileNetv2 [35], a pretrained object detection neural network found on the TensorFlow GitHub, where images are used. The second is a lightweight neural network that uses the binary classifier DeepConvRNN, using small sensor values as inputs.

As the setting for the external connection of the edge server, requests were distributed to each of the pods using the load balancer, a service component of Kubernetes. The load balancer is changed from the round-robin algorithm to the never-queue algorithm; this increases the number of connected clients as much as possible and was set to increase the number of connection pods. Furthermore, all boards used in the experiment were released and verified using the jetson_clocks script implemented inside the Jetson Board, for performance comparison.

4.3.1 Experimental result of basic scheduling method

First, in a basic pod allocation method, we check the availability of a neural network service application that utilizes image data. As listed in Table 4, object detection neural network-based pods can operate in standby mode on Nano, which is a board with low specifications. However, the memory requirements for input data and output data for driving a neural network are large; thus, the connection may be disconnected during the session operation for service.

To solve this discrepancy, the session option to limit the memory in a neural network application was added to the TensorFlow code, and the results are summarized in Table 5. However, Table 5 shows that the connection was lost because of the long wait caused by the Nano board using network time. Tables 4

Table 4 Execution results of the object detection neural network

Number of executing neural network(pod)	Number of request clients	Xavier		TX2		Nano	
		Neural network time (s)	Total running time (s)	Neural network time (s)	Total running time (s)	Neural network time (s)	Total running time (s)
1	**1**	0.1141	0.3130	Not connecting		Not connecting	
2	**1**	Not connecting		0.3048	0.4907	Not connecting	
	2	0.1142	0.2856	0.2995	0.4432	Not connecting	
3	**1**	Not connecting		0.3036	0.4978	Not connecting	
	2	0.2600	0.4016	0.3007	0.5158	Not connecting	
	3	0.1122	0.2584	0.2997	0.3768	Out of memory / Pod is on TX2	

Table 5 Results of object detection neural network with memory limit

Number of executing neural network (pod)	Number of request clients	Xavier		TX2		Nano	
		Neural network time (s)	Total running time (s)	Neural network time (s)	Total running time (s)	Neural network time (s)	Total running time (s)
1	**1**	0.1129	0.3498	Not connecting		Not connecting	
2	**1**	0.1147	0.2644	Not connecting		Not connecting	
	2	0.1120	0.3328	0.3002	0.4555	Not connecting	
3	**1**	Not connecting		Not connecting		TLS handshaking failed	
	2	Not connecting		0.3050	0.6662	TLS handshaking Failed	
	3	0.1146	0.2715	0.3023	0.4492	Freezing	

and 5 show the configuration problems of the neural network service applications that deal with media data (image and video). First, according to node performance, a difference occurs in the processing time of the neural network. Second, improper node allocation with nodes scheduled without considering the GPU resources does not satisfy the pod's operational performance.

The second experiment for the driver profiling model was to select the number of samples used for inference randomly and run the model, followed by a continuous inference test for at least five minutes. Table 6 summarizes the results of a driver profiling model in the general allocation method. Figure 10 shows the results of allocation according to pod scaling.

The results in Table 6 were obtained using the driver profiling model and show that the edge server can be operated stably by running a lightweight neural network for small-sized data. At this time, in the general method of allocation, the generated and scheduled pods do not show a significant difference in priority; therefore, the balanced pod distribution results can be checked, as shown in Fig. 10. However, when the number of pods is more than five, the pods assigned to the Nano board are automatically pending within Kubernetes because of insufficient memory for running the neural networks. To solve this problem, it is necessary to determine the level of weighting and the actual degree of GPU workload and allocate it to the node.

4.3.2 Experimental results of new scheduling

The proposed allocation method adds a GPU clock as an extended resource that can be recognized by the Kubernetes system and executes the device query job performed in each node before the pod allocation process. The device query job uses the internal jetson_stats library that confirms the specifications. In the experiment, values of 1377 (Xavier), 1300 (TX2), and 612 (Nano) were obtained as the GPU specifications for each node. A stable driver profiling neural network was used because more than one pod should be executed in one node. In addition, given the availability of each board identified in the previous experimental results, the available GPU clock required in one pod setting was limited to 400. As shown in Fig. 11, the results ensure pod scaling. Table 7 summarizes the performance of the neural network on each node, and a standard deviation of 2.0 to 2.4% is confirmed. This means that the performance of each neural network was not affected even if the number of pods in one node increased.

The weight of each node was set to a value of 70 in Xavier and 40 in TX, which changed the priority. Subsequently, the new pod allocation order according to the increase in the number of pods produced the result shown in Fig. 11. As shown in Fig. 11, the pod preemption between TX2 and Xavier can be checked under similar conditions and confirms that up to seven pods can be serviced during the same processing period.

Table 6 Results of driver profiling neural network with basic scheduling

Number of pods	Xavier		TX2		Nano	
	Testing accuracy (%)	Running time (s)	Testing accuracy (%)	Running time (s)	Testing accuracy (%)	Running time (s)
1	Not connecting		85.96	0.3246	Not connecting	
2	86.43	0.3661	85.88	0.4114	Not connecting	
3	84.33	0.3412	84.47	0.4296	86.93	0.4957

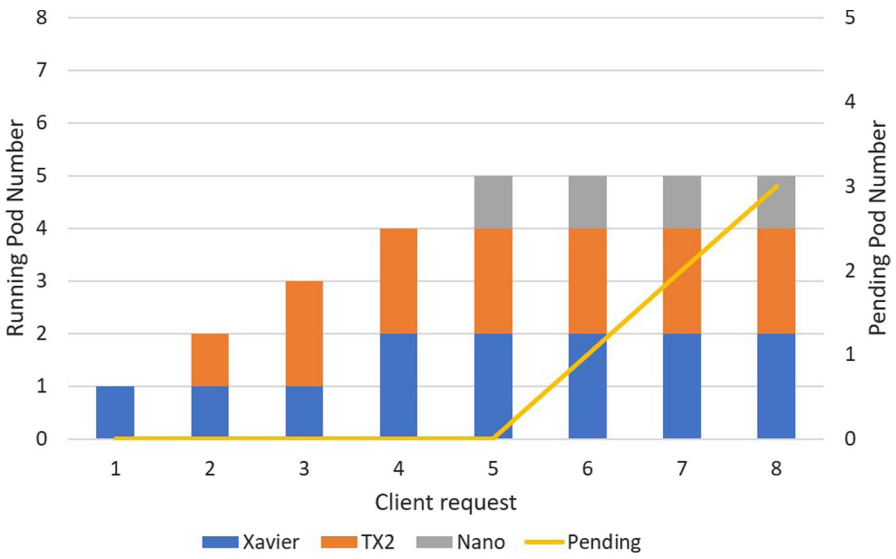

Fig. 10 Result of Pod Scaling Using Driver Profiling Neural Network

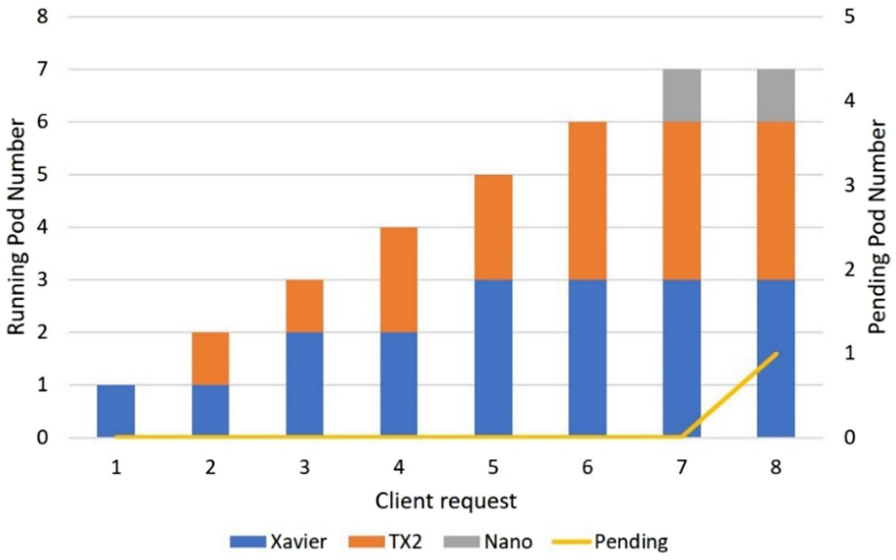

Fig. 11 Result of Scheduling using Driver Profiling with Extended Resource

4.3.3 Experimental result of Integral offloading

The purpose of the experiment was to measure the delay between the edge server and the central cloud when using integral offloading and compare it with previous

Table 7 Neural network performance in Pods running on each board

Number of pods for each node	Number of all pods	Xavier		TX2		Nano		Average (when node hybrid)
		Testing accuracy (%)	Running time (s)	Testing accuracy (%)	Running time (s)	Testing accuracy (%)	Running time (s)	Running time (s)
1	3	88.69	0.3744	81.76	0.4333	86.38	0.5345	0.4474
2	4	86.59	0.4016	83.85	0.3709	Out of memory		0.3416
		85.46	0.2845	82.85	0.3092			
3	6	88.03	0.2871	86.33	0.3709	Out of memory		0.3466
		81.30	0.3178	87.26	0.3454			
		87.12	0.3413	86.19	0.4173			
Average (one type nodes)		86.20	0.3345	84.71	0.3745	86.38	0.5345	

studies [42–43] dealing with other deep learning services that applied edge computing structures. In the experiment, before the implemented WebSocket receiving server responded to the requested deep learning service, nodes added or excluded from the entire edge server's cluster configuration were checked, and the total capacity was calculated by monitoring the hardware performance of the nodes. Subsequently, the offloading condition was determined according to the neural network service and the edge server's availability. Then, deep learning was performed by selecting a computational domain. As a deep learning model, we used an image recognition model [35] to confirm the application of other deep learning models in the edge server. The results of this experiment are shown in Table 8, along with the image detection neural network [39] results used in the previous experiment. We measured the end-to-end time and the inference time and averaged them when multiple container applications were executed in the edge server.

In Table 8, the end-to-end time is the time until a response is received from the client and represents the performance of the proposed structure. In addition, inference time is added to check the difference in network delay separately from the performance of each model because the calculation area executed through offloading is different. First, in the paper on Deep Decision [41], the resulting final time in each computational domain is shown through the YOLO model using image data. This study uses the same offloading as the proposed server, but because the tested center cloud covers a larger network distance than the local server, constructing an extended structure of the proposed server can be seen as the result of the experiment. The paper referenced for the comparison of neural network inference time was EdgeEye [42]. This paper presented API framework in the edge area and implemented a deployable deep-learning application. The model to be compared was DetectNet and a direct comparison with the image recognition model is difficult because the optimization through tensor RT is in progress. However, compared to the driver profiling model, which requires less computation, it has a similar inference time. Finally, in paper [43], the method was implemented on a cloudlet-based VM rather than a container-based environment, and the deep learning model was run on mobile devices and cloudlets using snapshot data of the Virtual Machine (VM) for partial offloading. This method is numerically similar to that of the proposed server. However, the method in this paper is implemented with partial offloading, which has disadvantages in terms of the configuration of multiple deep learning applications compared to the proposed edge server.

5 Conclusion

In this paper, we propose the configuration of a Kubernetes-based edge server: 1. the configuration of a Kubernetes-based edge server, 2. the identification of the network environment, 3. the implementation of a neural network model that can be configured for neural network applications, 4. pod allocation using affinity, and 5. a new extended resource method for improving the efficiency of GPU nodes. The proposed embedded edge server is the basis for providing services with low

Table 8 Offloading results and comparison with other papers

Structure	Offloading	Neural network type	Computing layer	End-to-end time (sec)	Inference time (sec)
Proposed container-based edge server	None (only edge server)	Image detection (ssd-mobilenet-v2 [39])	Edge	(Xavier) 0.305 (TX2) 0.524	(Xavier) 0.113 (TX2) 0.3025
	Integral offloading	Driver profiling	Edge	0.289	0.0207
			Center Cloud	0.435	0.0229
		Image recognition (mobilenet-v2 [35])	Edge	0.323	0.113
			Center Cloud	0.453	0.0554
Deep decision [41]	Integral offloading	YOLO	Front-end device	2.0	N/A
			Server	0.25 ~ 0.3	N/A
EdgeEye [42]	Integral offloading	DetectNet (optimized)	Server	N/A	0.018
Snapshot-based offloading [43]	Partial offloading	GoogleNet	Mobile + cloudlet	0.6	N/A
		AgeNet		0.34	
		GenderNet		0.34	

latency to nearby clients and is composed of Kubernetes cluster servers using containers to compose complex applications.

As a result of our first experiment, the service advantage of the edge server was confirmed because of the three times difference in transmission bandwidth between the edge local area and the Internet network.

The second set of experiments was conducted on two service applications using a newly constructed neural network model on an edge network. By comparing the two neural networks, the edge server operation for object detection confirmed a memory limitation problem, which is a limitation of the embedded edge server model, and showed that the lightweight neural network model—the driver profiling model—is suitable for operation on the edge server.

Third, when scaling a pod, we experimented by increasing the number of pods that can be operated, and with the efficient allocation of nodes using the proposed method to solve the problem of resource allocation imbalances caused by using embedded board GPUs. The proposed method sets the number of internal pods available on each board, which is confirmed through additional extended resource and affinity settings for the GPU inside the node.

Finally, it was possible to reduce the overhead of moving the node because of the additional impossibility of running the pod. Moreover, by adding offloading, we checked the end-to-end time and the inference time of the neural network model being executed, compared the structure with those in other papers, and comparatively analyzed the merits.

Acknowledgments This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2018R1D1A1B07042602) and in part by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korean government (MSIT) (No. 2019-0-00064, Intelligent Mobile Edge Cloud Solution for Connected Car, No. 2019-0-00240, Deep Partition-and-Merge: Merging and Splitting Deep Neural Networks on Smart Embedded Devices for Real Time Inference).

References

1. Duan Q (2017) Cloud service performance evaluation: status, challenges, and opportunities—a survey from the system modeling perspective. Digital Commun Networks 3(2):101–111
2. Shirazi SN, Gouglidis A, Farshad A, Hutchison D (2017) The extended cloud: Review and analysis of mobile edge computing and fog from a security and resilience perspective. IEEE J Sel Areas Commun 35(11):2586–2595
3. Burke B, Cearley D, Jones N, Smith D, Chandrasekaran A, Lu CK, Panetta K (2019) Gartner top 10 strategic technology trends for 2020-Smarter with Gartner
4. Contributor IBM (2019, December 13) IBM BrandVoice: IBM Tech Trends To Watch In 2020 ... And Beyond. Retrieved from https://www.forbes.com/sites/ibm/2019/12/09/ibm-tech-trends-to-watch-in-2020--and-beyond/#280a11974c1c
5. Shi W, Pallis G, Xu Z (2019) Edge computing [Scanning the Issue]. Proc IEEE 107(8):1474–1481
6. Yousefpour A, Fung C, Nguyen T, Kadiyala K, Jalali F, Niakanlahiji A, Jue JP (2019) All one needs to know about fog computing and related edge computing paradigms: a complete survey. J Syst Architec 98:289–330
7. Lyu X, Tian H, Jiang L, Vinel A, Maharjan S, Gjessing S, Zhang Y (2018) Selective offloading in mobile edge computing for the green internet of things. IEEE Network 32(1):54–60

8. Markakis EK, Karras K, Sideris A, Alexiou G, & Pallis E (2017) Computing, caching, and communication at the edge: The cornerstone for building a versatile 5G ecosystem. In: IEEE Communications Magazine, 55(11), 152–157.]
9. Kiani A, Ansari N (2017) Toward hierarchical mobile edge computing: an auction-based profit maximization approach. IEEE Internet Things J 4(6):2082–2091
10. Ren J, Guo H, Xu C, Zhang Y (2017) Serving at the edge: a scalable IoT architecture based on transparent computing. IEEE Network 31(5):96–105
11. Ryden M, Oh K, Chandra A & Weissman J (2014, March) Nebula: Distributed edge cloud for data intensive computing. In: 2014 IEEE International Conference on Cloud Engineering (pp. 57–66). IEEE
12. Noreikis, M., Xiao, Y., & Ylä-Jaäiski, A. (2017, May). QoS-oriented capacity planning for edge computing. In: 2017 IEEE International Conference on Communications (ICC) (pp. 1–6). IEEE
13. Malandrino F, Kirkpatrick S & Chiasserini CF (2016, December) How close to the edge? delay/utilization trends in mec. In: Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking (pp. 37–42)
14. Kamiyama N, Nakano Y, Shiomoto K, Hasegawa G, Murata M & Miyahara H (2016, December) Analyzing effect of edge computing on reduction of web response time. In: 2016 IEEE Global Communications Conference (GLOBECOM) (pp. 1–6). IEEE
15. Hou IH, Zhao T, Wang S & Chan K (2016, July) Asymptotically optimal algorithm for online reconfiguration of edge-clouds. In: Proceedings of the 17th ACM International Symposium on Mobile Ad Hoc Networking and Computing (pp. 291–300)
16. Zhang W, Hu Y, Zhang Y & Raychaudhuri D (2016, December) Segue: Quality of service aware edge cloud service migration. In: 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (pp. 344–351). IEEE
17. Ismail BI, Goortani EM, Ab Karim MB, Tat WM, Setapa S, Luke JY & Hoe OH 2015, August) Evaluation of docker as edge computing platform. In: 2015 IEEE Conference on Open Systems (ICOS) (pp. 130–135). IEEE
18. Pahl C, Helmer S, Miori L, Sanin J & Lee B (2016, August) A container-based edge cloud paas architecture based on raspberry pi clusters. In: 2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW) (pp. 117–124). IEEE
19. Helmer S, Pahl C, Sanin J, Miori L, Brocanelli S, Cardano F & Sharear AM (2016, November) Bringing the cloud to rural and remote areas via cloudlets. In: Proceedings of the 7th Annual Symposium on Computing for Development (pp. 1–10)
20. Elkhatib Y, Porter B, Ribeiro HB, Zhani MF, Qadir J, Rivière E (2017) On using micro-clouds to deliver the fog. IEEE Internet Comput 21(2):8–15
21. Zhang X, Wang Y & Shi W (2018) pcamp: Performance comparison of machine learning packages on the edges. In: {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)
22. Pahl C & Lee B (2015, August) Containers and clusters for edge cloud architectures—a technology review. In: 2015 3rd international conference on future internet of things and cloud (pp. 379–386). IEEE
23. Kim JH, Tulkinbekov K, Kim DH (2019) Benchmarking Kubernetes based Edge Server in Embedded Environment (pp. 49–52). In: The 5th International Conference on Next Generation Computing 2019 Proceeding
24. Kubernetes Runtimes (2018, November 18) Retrieved April 20, 2020, from https://docs.google.com/spreadsheets/d/17ak_fVtWNUwMMJNpo7dNkoR8KK1ezTZVvh6v_tBcP7Y/edit#gid=0
25. Bernstein D (2014) Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comp 1(3):81–84
26. Kang H, Le M & Tao S (2016, April) Container and microservice driven design for cloud infrastructure devops. In: 2016 IEEE International Conference on Cloud Engineering (IC2E) (pp. 202–211). IEEE
27. Docker (2020, March 31) docker/classicswarm. Retrieved from https://github.com/docker/swarm/
28. Production-Grade Container Orchestration (n.d.). Retrieved from https://kubernetes.io/
29. Hindman B, Konwinski A., Zaharia M, Ghodsi A., Joseph AD, Katz RH & Stoica I (2011, March) Mesos: A platform for fine-grained resource sharing in the data center. In: NSDI (Vol. 11, No. 2011, pp. 22–22)
30. Hoque S, de Brito MS, Willner A, Keil O & Magedanz T (2017, July) Towards container orchestration in fog computing infrastructures. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC) (Vol. 2, pp. 294–299). IEEE

31. NVIDIA DeepStream SDK (2020, April 25) Retrieved April 27, 2020, from https://developer.nvidia.com/deepstream-sdkNvidia. (2020, February 26). NVIDIA/nvidia-docker. Retrieved from https://github.com/NVIDIA/nvidia-docker
32. Nvidia (2020, February 26) NVIDIA/nvidia-docker. Retrieved from https://github.com/NVIDIA/nvidia-docker
33. GPU-Accelerated Innovation with NGC (n.d.). Retrieved from https://www.nvidia.com/en-us/gpu-cloud/
34. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016, October). Ssd: Single shot multibox detector. In: European conference on computer vision (pp. 21–37). Springer, Cham
35. Sandler M, Howard A., Zhu M, Zhmoginov A & Chen LC (2018) Mobilenetv2: Inverted residuals and linear bottlenecks. In: Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 4510–4520)
36. Zhang J, Wu Z, Li F, Xie C, Ren T, Chen J, Liu L (2019) A deep learning framework for driving behavior identification on in-vehicle CAN-BUS sensor data. Sensors 19(6):1356
37. Lin TY, Maire M, Belongie S, Hays J, Perona P, Ramanan D & Zitnick CL (2014, September) Microsoft coco: Common objects in context. In: European conference on computer vision (pp. 740–755). Springer, Cham
38. Kwak BI, Woo J & Kim HK (2016, December) Know your master: Driver profiling-based anti-theft method. In: 2016 14th Annual Conference on Privacy, Security and Trust (PST) (pp. 211–218). IEEE
39. Tensorflow (2020, April 13). tensorflow/models. Retrieved from https://github.com/tensorflow/models/tree/master/research/object_detection
40. Mortimer M. (2018) iperf3 documentation
41. Ran X, Chen H, Zhu X, Liu Z & Chen J (2018, April) Deepdecision: A mobile deep learning framework for edge video analytics. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications (pp. 1421–1429). IEEE
42. Liu P, Qi B & Banerjee S (2018, June) Edgeeye: An edge service framework for real-time intelligent video analytics. In: Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking (pp. 1–6)
43. Jeong HJ, Jeong I, Lee HJ & Moon SM (2018, July) Computation offloading for machine learning web apps in the edge server environment. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS) (pp. 1492–1499). IEEE
44. Ullah S and Kim DH 2020 Benchmarking Jetson Platform for 3D Point-Cloud and Hyper-Spectral Image Classification. In: 2020 IEEE International Conference on Big Data and Smart Computing (BigComp), pp. 477–482. IEEE
45. Ullah S, Kim D-H (2020) Lightweight driver behavior identification model with sparse learning on In-Vehicle CAN-BUS sensor data. Sensors 20(18):5030