An Adaptive Computation Framework of Distributed Deep Learning Models for Internet-of-Things Applications

Mu-Hsuan Cheng, QiHui Sun and Chia-Heng Tu
National Cheng Kung University, Tainan 70101, Taiwan
Email: chiaheng@mail.ncku.edu.tw

***Abstract*—We propose the computation framework that facilitates the inference of the distributed deep learning model to be performed collaboratively by the devices in a distributed computing hierarchy. For example, in Internet-of-Things (IoT) applications, the three-tier computing hierarchy consists of end devices, gateways, and server(s), and the model inference could be done adaptively by one or more computing tiers from the bottom to the top of the hierarchy. By allowing the trained models to run on the actual distributed systems, which has not done by the previous work, the proposed framework enables the co-design of the distributed deep learning models and systems. In particular, in addition to the model accuracy, which is the major concern for the model designers, we found that as various types of computing platforms are present in IoT applications fields, measuring the delivered performance of the developed models on the actual systems is also critical to making sure that the model inference does not cost too much time on the end devices. Furthermore, the measured performance of the model (and the system) would be a good input to the model/system design in the next design cycle, e.g., to determine a better mapping of the network layers onto the hierarchy tiers. On top of the framework, we have built the surveillance system for detecting objects as a case study. In our experiments, we evaluate the delivered performance of model designs on the two-tier computing hierarchy, show the advantages of the adaptive inference computation, analyze the system capacity under the given workloads, and discuss the impact of the model parameter setting on the system capacity. We believe that the enablement of the performance evaluation expedites the design process of the distributed deep learning models/systems.**

***Index Terms*—Distributed systems design, distributed deep learning models design, parallel computing, Internet-of-Things applications, embedded devices, OpenCL accelerations, MQTT**

I. Introduction

Deep learning algorithms have demonstrated their usefulness for real-world problems in the past few years. In particular, deep learning approaches, such as deep neural networks (DNNs), have produced the superior results compared with human experts in some cases, e.g., face recognition and image classification [1]. With the great successes, DNNs techniques are now widely adopted in various Internet-of-Things (IoT) applications to help make better and wiser decisions.

For IoT applications, sensors are often deployed at the perimeters in the fields to collect data for further usages. It is a common practice that a large quantity of data captured by these sensors is handled by the end devices nearby, rather than streaming these data back to the cloud. The handling of the collected data on the near-user devices provides lower service latency, as opposed to the cloud computing paradigm, where the data storage, management, and processing are performed at remote servers in the cloud. The idea of the geographical data handling is similar to the concept of the *fog computing* [2].

The shifting of the data processing from the cloud to the end devices has several advantages, such as lowering the latency responses, backbone network resources utilization, and the loading of servers in the cloud. Nevertheless, when DNNs are adopted in the fields, they are likely to consume significant resources (e.g., computation, memory, and battery life) of the end devices when handling the large DNN models, which are conventionally handled by the resource-rich machines in the cloud. Simply redesigning a compact and smaller DNN models do not seem to be a good choice. As the computing power varies from one device to another, it is almost impossible to have a DNN model that runs well on different types of embedded devices, in terms of the delivered accuracy and performance.

To tackle the above problem, the distributed deep neural networks (DDNN) [3] had been developed to enable the hierarchical inference of the developed DDNN models. Especially, the previous work developed the design of DNN models for the adaptive inference operation: the prediction result of a DDNN model is computed and returned by the end devices immediately, if the system is confident with the outcome; otherwise, the intermediate results of the model inference done at the end devices will be sent to the cloud for further processing and the corresponding outcome is sent back to the end devices.

In this paper, we develop the software framework to further facilitate the DDNN models running on physical systems with a distributed computing hierarchy. With our proposed framework, system designers for the IoT applications are allowed to check the delivered performance and model accuracy on the physical systems. Furthermore, the performance evaluation of the physical systems helps make the design decisions, which is hard to be done by the previous work. We have done a case study on the surveillance system to demonstrate the capabilities of the framework.The experimental results show that the framework opens a door for exploring the design spaces of DDNN-based systems.

In the rest of the paper, background of the DDNN design,

and the motivation and contributions of the proposed framework are introduced in Section II. The overview and the key components of the framework are described in Section III. Section IV presents the preliminary experimental results of the case study on the surveillance application. The previous works on the distributed computations are introduced and discussed in Section V. Section VI concludes this work.

II. Background and Motivation

The distributed deep neural network (DDNN) developed by Teerapittayanon et al. [3] allows the inference operations of a trained DNN to be computed by heterogeneous devices collaboratively. In particular, the distributed computing can be performed vertically and horizontally. The respect concepts are described as follows.

Vertical Distributed Inference. In the left of Figure 1, the end device takes the model input and performs a portion of the DNN inference computation on the device (also called *device inference*). There is an early exist point (Local Exit) after the device inference. If the current outcome at the early exit point is with sufficient confidence, then the end device uses the local inference result. Otherwise, for more difficult cases, where the local inference cannot provide confidence outcome, the intermediate DNN output (data right before the local exist point) is sent to the server, where further inference is performed by visiting the rest of the model layers and the outcome is computed (Server Exit). The partial inference done at the server side is also called *server inference* in this paper.

Horizontal Distributed Inference. In the right of the figure, which depicts the variant model from the left of the figure, two end devices are involved in an application and the outcomes at the local exit point are merged and checked the confidence. Similar to the above scenario, the result of the merged data is returned directly (Local Exit) or sent to the server for further inference (Server Exit), depending on the confidence of the result at the local exit point.

It is interesting to note that Figure 1 illustrates the simplified concept of the DDNN architectures. In theory, it could be further scaled vertically and horizontally. For example, the edge device, such as network gateway, could be added to the two-tier hierarchical distributed computing containing one end device and one server to form the three-tier hierarchy. Similarly, in the horizontal direction, more end devices could be added to provide more analyzed data to improve the result of the inference.

In the following subsections, the training and inference operations of the DDNN design are introduced. In addition, the motivation of the proposed framework for the design of the actual DDNN systems is presented, and the contributions of this work are listed.

A. DDNN Training

When the design of a DDNN model is available, the model training process could be performed at one powerful machine or several servers in the cloud. Considering the design of the multiple exit points in DDNN models, the training process

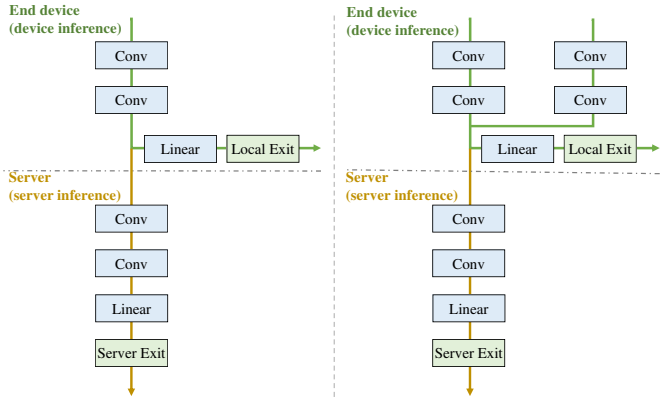

Figure 1. Two forms of the basic DDNN models developed in [3].

of DDNN models will be a little bit different from that of conventional DNN models, which is summarized as follows. During the feedforward phase of the model training, the training data set is passed through the model and the outcomes at all exit points are saved, where the error of the given model is derived as well. In the backward propagation phase, the calculated error from each exit point is passed back through the model and is combined for jointly training of the entire model. The formal definition of the model training is defined in [3].

B. DDNN Inference

The key to the adaptive, hierarchical inference of the trained DDNN model is the dynamic checking of the predicted result, which is done by comparing the prediction against the predetermined threshold, t, at a certain exit point. The thresholds, t_i, could be set for each exit point, i, at the end of the training process, where possible values of t are explored on the validation set of the test data and the value with the best accuracy is selected.

In the classification example used in the paper [3], the normalized entropy is computed and compared against t at each exit point, as a measure of the confidence checking for the prediction. The normalized entropy is a floating point value, ranging from 0 to 1. A lower value means greater confidence. During the inference, if the normalized entropy is less than the predefined threshold, t_i, it means the DDNN system is confident with the prediction at the i-th exit point, and the prediction will be returned by the system. Otherwise, the system falls back to the main model and performs the checking at the next exit point. This checking continues until the last exit is reached.

C. Motivation: Exploring DDNN System Designs

While the DDNN work [3] shows that the distributed inference of a trained DDNN model is able to run on the vertical and/or horizontal computing hierarchies as illustrated in Figure 1, extra efforts are required for the DDNN concept to be practically used in the field. In particular, when designing the DDNN models for the IoT applications, the computation

capabilities and the power consumptions of the IoT devices should be taken into account as well, instead of merely putting emphasis on tweaking the accuracies of the models. The major reason behind the observation is that for IoT applications, there are various types of end devices deployed on the perimeter in the field, and the computing power of these devices is not matched those machines used for training the models. It is hard to anticipate the execution time (and the consumed energy) of a trained DDNN model on an end device.

For better and efficient software/hardware co-designs, these trained models should be deployed onto the distributed systems to measure the execution times and the energy consumption. If the given time/energy constrains are violated, the models should undergo another model training process with the information, in terms of the measured time/energy data. We use the left model shown in Figure 1 as an example to illustrate the design considerations of a DDNN model for an IoT application. When the NN layers are fixed, which are determined by previous training iterations, the next decision to be made by the system/model designers are the number and the position of the local exit point, as well as the threshold value at the local exit. But, the decision is meaningless, unless the performance of the trained model is measured on the device and the designers ensure that the delivered performance meets the design goal. Without the performance measurement process to assist the model training, in the worst case, the model inference might take a very long time and its battery would be drained out quickly.

To facilitate the DDNN systems (and also models) design process, we build the software framework that accepts the trained DDNN model for the inference on a distributed computing hierarchy. The key contributions of this work are as follows.

1) Enable the inference of a built DDNN model to be performed adaptively on the end device, and the server, if necessary. The framework can be further extended (with little efforts) to support a deeper computing hierarchy, e.g., the three-tier hierarchy with device, edge, and server.
2) Allow the generation/execution of the C version for a trained DDNN model, where the C code is allowed to run on embedded devices. Furthermore, our framework generates the OpenCL/C version of a trained DDNN model, which will take the advantage from the accelerator(s) available on the hardware platform.
3) Facilitate performance measurements of the trained DDNN models on actual distributed systems, which helps the model optimization process before the systems are actually deployed in the filed.
4) Developed a case study of the real world surveillance application, which mimics the essential operations involved in various IoT applications, e.g., smart hospitals and warehousing. With the proposed framework, we are able to measure the performance of the developed model on the distributed computing hierarchy. The performance of the inference on the device is further accelerated by the OpenCL accelerator by a factor of 24. The preliminary performance results are also helpful for guiding the DDNN system design, i.e., estimating the server throughput.

III. The Proposed Framework

Figure 2 depicts the system architecture of the proposed framework. While one end device and one server are used as an example to illustrate the relationships among the key components, the proposed framework is able to support more complex forms.

The proposed framework requires that the applications running on the end devices use the trained *DDNN models* (for *DDNN Apps* on the end device in Figure 2). These models are mapped and run on the end devices and the server collaboratively with the support of the two main modules in the proposed framework.

Server Module is responsible for working with the *DNN framework*[1] for the model training, saving the trained models, and generating the C codes of the trained models. When the DDNN system is deployed, the Server Module is responsible for handling the requests made by the end devices by performing the inference operations of the selected models.

Device Module invokes the inference of the selected DDNN model required by the *DDNN App*. For example, if the *object recognition* is desired, the corresponding DDNN model is loaded to perform the inference operation (by executing the C code generated by the Server Module). The result of the inference operation is returned by the value computed either at the local exit point(s) or at the exit point(s) on the server.

The two modules are detailed in sections III-A and III-B, respectively. Section III-C introduces the execution model of the proposed framework. Section III-D gives the system assumptions and important details during our design and development process.

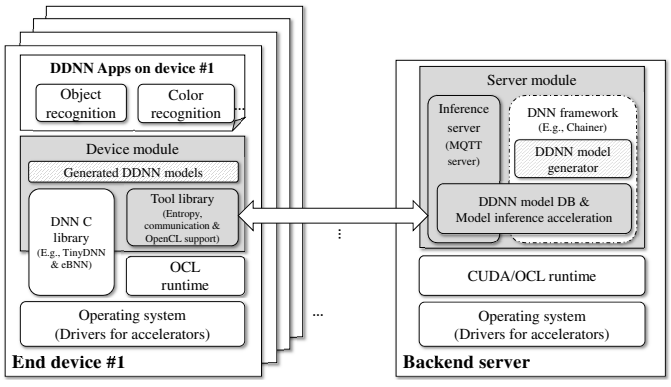

Figure 2. System architecture of the proposed framework.

[1]The compatibility of the proposed framework with the DNN frameworks is discussed in Section III-D1.

A. Server Module

This module is run in two different phases: before and after the DDNN system deployment. The module comprises several submodules that are described as follows. Before the DDNN system runs, the DDNN models are built with the *DNN framework*, which has the DDNN support [3], as illustrated in the white dashed block within the *Server Module* in Figure 2. The trained models are saved in *DDNN Model DB* for later usage.

Our enhanced *DDNN Model generator* is responsible for generating the C version code of the trained DDNN model, and the generated code will be run on the end devices later. Taking the DDNN model in the left of Figure 1 as an example, the generated C code is for the first two convolution layers and the linear layer, which are responsible for computing the prediction result at the local exit point. In addition, our generator is able to generate the OpenCL/C version code for the trained DDNN model. This capability can greatly improves the computing power at end devices when OpenCL accelerators are available. More about the C code generation is introduced in Section III-B.

The DDNN system starts with the running of *Inference Server*. The server is built upon the MQTT-based service [4] that establishes stable network connections between the end devices and the server. Upon receiving a request made by an end device, a thread is picked from the thread pool by the *Model Inference Acceleration* module, where the selected thread loads the designated DDNN model requested by the device from DDNN Model DB and feeds the intermediate data into the DDNN model for further inference. Using the above DDNN model example, the thread continues the inference from the third convolution layer (i.e., skipping the first two convolution layers shown in the upper-left of Figure 1), and returns the computed the result at the server exit point. When performing the inference at the server, the corresponding computation is able to be accelerated by CUDA/OpenCL devices, depending on the capabilities of the DNN framework on the system. Currently, the framework we used in the experiments supports CUDA framework. More detailed information is offered in Section IV.

B. Device Module

As shown in Figure 2, there are three major components in the device module: the *generated DDNN models*, the third-party implementation of *DNN C library*, and the *tool library*. Since our system aims to run on various types of embedded devices, the components within the device module are implemented in C language.

Each generated model contains the C code that defines the DNN network structure and trained parameters, where the actual C implementation of each network layer is defined in the third-party DNN library. Whenever the local exit points are reached, the functions defined in the tool library are invoked by the generated C code for check the confidence of the predictions and for requesting the server inference. We take the DDNN model in the left of Figure 1 as an example to concretely illustrate the inference done at an end device (also called device inference); Figure 3 gives C-like pesudocode for the generated DDNN model (`dev_model_infer.c`), the third-party implementation of the DNN layers (`dnn_c_lib.h`), and the tool library (`tool_lib.h`).

The design considerations are discussed as below.

- The generated model is application dependent, which means the trained model is able to be reused across different platforms for solving the same problem. However, when the same model runs on different platforms, one thing should be taken into account that the inference time required by one device may vary from another. If the inference time is a major concern, the model may be redesigned, e.g., by altering the position of the local exit point.
- While there are many DNN frameworks written with high-level languages to facilitate the model building process, the C/C++ based libraries [5], [6] become popular as the DNN computations are shifted from cloud servers to embedded devices for various application domains. Some of them support computation acceleration with OpenCL/CUDA, such as tiny-dnn [5]. To cope with the acceleration code, our model generator should be aware the acceleration framework (OpenCL or CUDA) supported by the DNN framework and generate the *host program* for the acceleration framework, so that the generated model allows to run with the faster version. In our current implementation, our framework generates the OpenCL host program, because of it is widely supported by the vendors of different types of accelerators, such as multicore processors, DSPs, and FPGAs. The generated OpenCL program contains the function calls to either the parallel implementations of the NN layers or the mixed of sequential and parallel implementations. Nevertheless, simply calling the CUDA/OpenCL version implementation of an NN layer may not deliver the best performance, as the computation offloading may involve extra overhead for data movements. Judiciously choosing between the sequential execution and the acceleration with CUDA/OpenCL for the DNN model is an important problem.
- Tool library is a platform independent software module and runs across different types of embedded systems after recompilation. This library provides everything needed for the inference at the server. In addition, when the selected library comes without OpenCL support, one can implement its own OpenCL version of the network layer and added into the tool library as an extension. In our current implementation, we choose to use the enhanced NN layers (i.e., eBNN [6]) that are designed specifically for low-end embedded systems with several tens of KBs of system memory. However, it does not support the OpenCL acceleration. We have ported the

C implementation of eBNN into the OpenCL version to facilitate the computation acceleration.

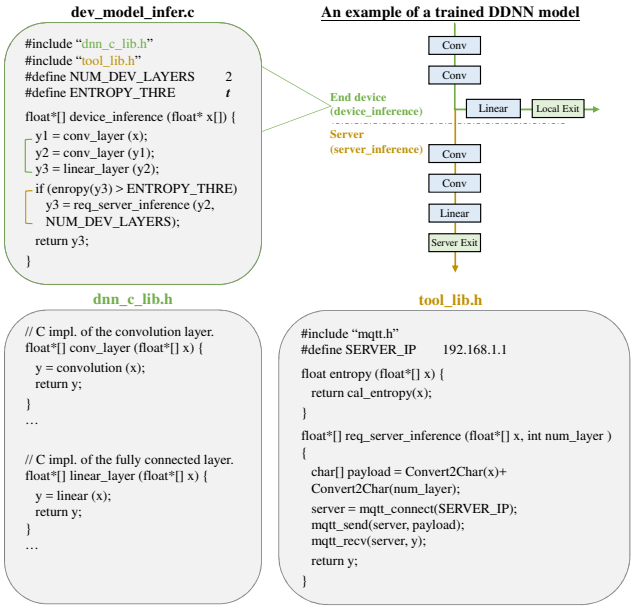

Figure 3. Example codes for the inference performed at an end device (device inference).

C. Execution Model

Continuing from the example shown in Figure 3, which illustrates how the hierarchical inference is taken place from the end device, this subsection further details the workflow and considerations at system level, where end devices and a server exchange model inference requests and the corresponding predictions with MQTT. MQTT uses the subscribe/publish model that involves at least a MQTT *client* and a MQTT *broker*. Initially, the client makes a subscription to the broker. Later, when the broker receives a piece of published data from some client, it forwards the data to the subscriber, where the published data and the subscription are matched by the *topic*, which is usually an non-empty string.

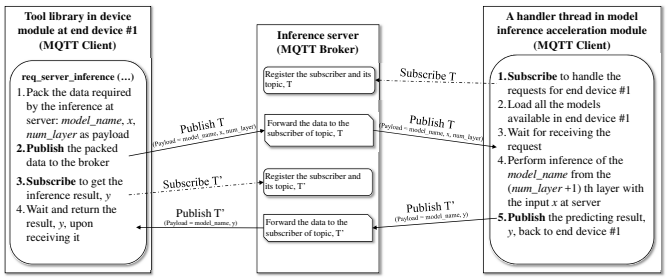

Figure 4. Execution model of the server inference done with MQTT protocol, where the MQTT *broker* is responsible for recording the subscriptions and forwarding the published data to the subscribed MQTT *client*.

Figure 4 shows the flow of a server inference request made by the end device and the return of the corresponding prediction from the server, where the tool library at the end device and the model inference acceleration module at the server act as the MQTT *client*, and the inference server at the server runs as the MQTT *broker*. It is interesting to note that the two clients act as both subscriber and publisher at the different time points. For example, the module is a subscriber for the topic, *T*, for handling the request of server inference. At the same time, the module becomes a publisher when the prediction of *T* is available, and at this moment, it runs as a publisher of *T'* sending the result to the end device through the broker. The tool library has similar behaviors.

The major advantage of the above design is the ability of scaling, which is achieved by the thread pool design of the model inference acceleration on top of the MQTT protocol. In this design, each thread registers (via subscription) to the inference server and handles the matched request (published data).

Scaling up (i.e., vertical scaling) would be done by increasing the computing power and the memory space on the machine. The increasing computation need is caused by handling the incoming publishing data[2]and the server inference, where the later would often be accelerated by the accelerators, such as GPUs. When handling the publishing data is not the major issue[3], adding the accelerators on to the server would increase the system throughput. On the other hand, the increased memory space is for loading all the trained models that could be invoked by the end device, as illustrated in the right of Figure 4. This design allows multiple requests made by an end device to be served quickly. Especially, when multiple end devices share common DDNN models, this design further increases the system throughput. More about the design strategies, such as the memory usage versus the request latency, are discussed in Section III-D.

Scaling out (i.e., horizontal scaling) would be achieved by running a model inference acceleration module on another machine other than that of the inference server. Thanks to the subscribe/publish model in MQTT, the porting effort is minor for matching the system configurations; for example, it might need as much as changing the IP address of the inference server to scale out. When the underlying machine of the inference server has sufficient computation power, more physical machines are allowed to added into the system, each of the machines runs a acceleration module.

D. System Design Remarks

1) Characteristics of the Target Systems: We present some implementation details which help identify the assumptions of this work. Our framework is developed on top of the Chainer-based framework [3], as the framework supports the DDNN functionality. The proposed framework is able to be ported onto other deep learning frameworks. It is important to note

[2]To shorten the latency of each request, the subscription could be performed earlier during system initialization.

[3]From the related study [7], where several MQTT implementations were benchmarks, the MQTT servers were able to sustain several thousands of publishing data (connections) on the machine with two CPU cores and 4GB RAM over the gigabit network. For some MQTT implementation, the CPU utilization kept steady at 50% when handling several thousands of connections.

that as the server inference is always performed staring from some layer of the given model, the target DNN framework should be able to start the inference at any specified layer. For our experiences, some of the software frameworks do not support this feature as the layers in a trained model is packed together to improve the inference performance, in which case, it will raise a problem for the porting. On the other hand, the frameworks which support layer-by-layer processing of a DNN have a greater chance to be extended for DDNN support.

Based on the requirement of the DDNN framework [3], the server machine should have the CUDA support for the model training and the server inference. On the other hand, the previous work leverages the C-based eBNN layers for building the network layers, which were able to run on the Arduino 101 based platform. Furthermore, to make the requests of server inference, MQTT requires the target platform has the TCP/IP support. As open source lightweight TCP/IP implementations, such as lwIP, are available for Arduino platforms, we believe that our implementation is able to be ported on to the microcontroller-grade platforms with little efforts.

2) Code Generation for End Devices: To facilitate the device inference, our framework generates the C program for the trained DDNN model. To avoid the huge efforts for building DNN C libraries for end devices, our design allows the use of third-party library under the assumption that the DNN framework at the server side should have the matched layers. For example, in our current design, the Chainer-based framework [3] has both of the Python code and the C implementation of the eBNN layers. In fact, their implementation helps generate the C program for device inference. In this work, we added the OpenCL/C implementation of the eBNN layers and generated the corresponding OpenCL/C code to accelerate inference performance. Together with the original functions supported by the Chainer framework, our framework would support three code versions for the end devices: Python, C, OpenCL.

3) Thread Handling Models in Model Inference Acceleration: In our work, we leverage the message forwarding mechanism of MQTT for the inference request forwarding and handling. The inference server is responsible for dispatching incoming inference requests to the corresponding handler threads within module inference acceleration module. In particular, this is achieved by the subscribe/publish model of MQTT, which allows multiple subscribers to accept a single published data as long as their (subscribers/publisher) *topics* are matched. However, in practice, this is not an efficient way for the DDNN system, since handling a single request with multiple threads wastes the server resources. For a thread to deal with just one inference request, we judiciously design the *topics* that should be assigned in the DDNN system.

The possible topics could be the names of the supported DDNN models at the server, the identifiers of the end devices, or even the combination of both. The decision made on the *topics selection* impacts the latency of the server inference. Assuming that the name of models is chosen as the topics, and it happens that multiple end devices make server inference requests of the same model with different parameters concurrently, the average latency for the server inference handling will be extended, since in the current topic setting, the server inference of the model will be done by the handler thread and the handling of the concurrent requests will be serialized. In sum, the topic selection problem is application- and system-dependent and should be considered carefully for optimizing the system performance.

4) Miscellaneous Feature: For some IoT applications, data protection is an important issue to protect the contents of the data transferred between the end devices and the server(s) from the malicious third party. In this case, the MQTT implementations with the SSL/TLS support become a good choice. However, it should note that while turning on the data security feature helps data security, the performance of the server inference would be affected, especially at the end devices, due to lower computing power on these devices.

IV. Experimental Results

In this section, we developed the proposed framework and used it to build the prototype DDNN system, whose system architecture is illustrated in Figure 5. The prototype system emulates the essential operations in the DDNN-based surveillance systems that could be further adopted in the IoT applications. Especially, the prototype system acts like the fixed view surveillance cameras in a restricted area within a certain site where unwanted things are forbidden to enter. For instance, humans are allowed to enter the space, but others (e.g., cats and dogs) are not. When the unwanted things are entered, the camera which detects the unwanted object will raise an alarm for security guards to handle. Further, the DDNN systems are able to be plugged into the existing systems. For example, self-driving robotic vehicles in the smart healthcare and smart warehousing for moving medical equipments and transporting goods could integrate the DDNN system with theirs to enhance their capabilities.

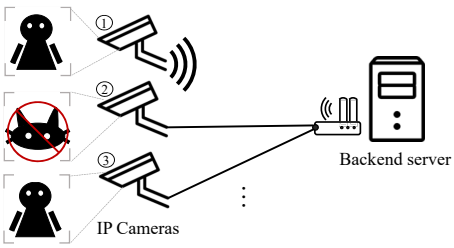

Figure 5. System architecture of the target surveillance application for object detecting, where a camera is emulated by the Odroid-XU4 and the server is run by the PowerEdge R730 listed in TABLE I.

To build the surveillance system, we use the end device and the server shown in TABLE I to establish the prototype system, where the device and the server are linked by the Gigabit Ethernet. Note that while the two-layer of the distributed computing hierarchy is used as illustrated in Figure 1, our system would be extended to support the three-layer computing hierarchy, including the end device, edge device, and

server. The CIFAR-10 dataset [8] is chosen to build our deep learning models and for exploring design alternatives. The dataset consists of sixty thousands of colour images in ten classes, each of the images is 32 by 32 pixels. Fifty thousands of the images are served as the training data for the DDNN models, whereas the rest of them are for testing the model accuracy.

TABLE I
THE CONFIGURATIONS OF THE END DEVICE AND THE SERVER USED IN OUR EXPERIMENTS.

	Software	Hardware
End device (Odroid-XU4)	Linux kernel 4.14, OpenCL 1.2, Chainer-1.17.0 on Python 2.7.12, Paho/C 1.2.0 for MQTT client	Samsung Exynos5422 Octa-Core processor (w/ four Cortex-A15 and four Cortex-A7 cores), 2GB System RAM, and the ARM Mali-T628 MP6 GPU (w/ six cores)
Server (PowerEdge R730)	Ubuntu16.04, CUDA Toolkit 8.0 (w/ cudnn library v5.1), and Chainer-1.17.0 on Python 2.7.12, Mosquitto 1.4.15 for MQTT borker, Paho/Python 1.3.1 for MQTT client	Intel Xeon E5-2650V4 processor (w/ 48cores), 94 GB System RAM, and the NVIDIA Titan Xp GPU (w/ 3840 cores and 12GB GDDR5X GPU RAM)

We used the following accuracy measures for computing the accuracy at exit points in DDNN models.

- *Local Accuracy (Local)* is the ratio of the number of the correct answers observed at the Local Exit point and the total number of test samples.
- *Server Accuracy (Server)* focuses on those samples that do not exit at the Local Exit point (i.e., the test samples reach the Server Exit point). The accuracy is the percentage of the correct answers on the tested samples at the server.
- *Overall Accuracy (Overall)* is the accuracy measured by the percentage of the samples exited at each exit point, i, with the given entropy threshold t_i.
- *Ideal Accuracy (Ideal)* is the ratio of the number of the correct answers observed at the Local or Server Exit points and the total number of test samples.

In our experiments, we aim to show that the proposed framework is able to facilitate the design of DDNN systems, i.e., co-designing of the DDNN model and the underlying systems. To this end, in the remaining of this section, we first describe the models that we built in the process of prototyping the surveillance system and the lessons learnt during the process. Then, we present the performance delivered by the DDNN systems. Next, we profile the performance of the server inference and use the data for further analyzing the server capacity. Finally, we display the delivered performance with different entropy thresholds and discuss the impact of the threshold value setting.

A. Model Design

While the model design is not the major focus of our framework, we share our model tweaking process when building the prototype surveillance system as a concrete example of the co-design process of the DDNN system. Note that the purpose of describing the process is not presenting a way to design the best DDNN model with the highest accuracy, but to emphasize the importance of the performance evaluation of the built model on the actual platforms. The parameters used in the model building are the type and length of network layers, the number of filters, and the entropy threshold. We examined the model accuracy and the inference time to evaluate if a model design is good enough.

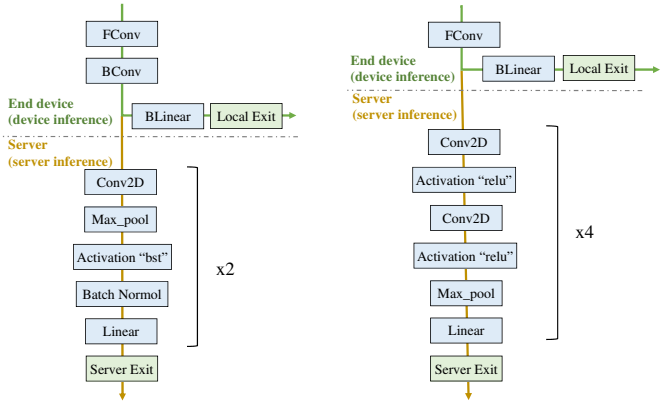

Figure 6. Two DDNN models developed in the experiments, where the left one is referred to as the *shallow* model and the right one is the *deep* model.

We started the design process with the default model developed in the DDNN source, whose structure is depicted in the left of Figure 6 referred to as the *shallow* model. In particular, the eBNN blocks are used for the end device and the enhanced Chainer layers are chosen for the server. As the types of eBNN blocks are limited, where only three kinds of blocks are available: FConv, BConv, and BLinear, we first changed the length of network layers and number of filters. We found that in our case, a deeper network contributes little to the model accuracy; i.e., we tried a deep network with more BConv layers, but the accuracy was not improved. On the other hand, the filer number has positive impact on the accuracy.

For example, the model accuracy was improved to 77% when the network is with 180 filters. Nevertheless, the number of adopted filters is proportional to the computation time. The inference time for the model with 180 filters is about 269 seconds. Obviously, it takes too much for practical use, and we found that the computation was dominated by the two eBNN blocks, FConv and BConv. We developed the OpenCL version of the blocks to enhance the inference time. As a result, the time is reduced to 11 seconds, which represents 24x speedups[4].

Aside from the OpenCL acceleration approach, we attempt to ease the burden of the end device via the model design approach. We removed the BConv block from the network and shift the loading to the server and change the types of the adopted network layers. We built a deeper network at the

[4]The inference time was measured in the early stage of our developed of OpenCL blocks on the machine with Intel i7-6700 and AMD Radeon RX 480 GPU. Further experiments are required to be performed in the experimental environment listed in TABLE I

server than the previous model, where the Conv2D layers are performed four times, referred to as the *deep* model shown in the right of Figure 6, as opposite to performed two times in the *shallow* model. As a result, the *Ideal Accuracy* of the model with 64 filters is up to 72% while the inference time (including the device and the server inference) is less than one second. Note that the entropy setting would affect both the model accuracy and the delivered performance. In this paper, the entropy threshold is set to 0.4 unless otherwise specified. The effects of the entropy setting are discussed later.

B. Adaptive Computation

Two inference configurations are adopted in our experiments to demonstrate the performance of the adaptive computation. One is the *Device&Server* for the distributed inference computation of the proposed framework and the other is the *Server Only* for the server to handle the entire model inference operation. The latter represents the setup for the conventional surveillance system that all the video contents are sent to the server for further process.

TABLE II lists the accuracy and the time for the *shallow* and *deep* models. The *shallow* model has the BConv block for filtering the features from the image contents. We expected that the convolution layer could enhance the accuracy at the end device. Nevertheless, the model with 64 filters has low accuracy, e.g., 54% for the *Ideal Accuracy*, and it costs tens of seconds to perform the device inference (T_Device), which further affects the combined inference time (T_Total). It is interesting to note that the *Local Accuracy* drops only one percentage after removing the BConv block in the *deep* model, where the time for device inference (T_Device) is decreased significantly to less than one second. Furthermore, more intense convolution layers performed in the server helps improve the accuracy, where the server inference time is increased to reflect the fact of the load shifting. As a result, *deep* is 37 times faster than *shallow* and reports more accurate results.

The *Server Only* configuration relies solely on the results at the Server Exit and has lower accuracy, compared with the distributed computation configuration. The time of T_Server will grow as the increasing of the filter number. In our experiments, the size of the payload is about 8KB, representing the intermediate data for 64 filters. As the server and the end device is linked by the campus network, (T_Comm.) is affected by the realtime network traffic, especially for the smaller packet. On the other hand, in our preliminary implementation, the raw data is not compressed and converted into the text format for MQTT transmission, and the size of the encoded data is more than 36KB, which results in the larger transmission time. In addition to the accuracy, the adaptive computation helps mitigate the server loading, which is introduced in the following subsection.

C. Server Capacity Analysis

We measured the GPU utilization and the peak memory footprint as the indicators to show the advantage of this work and to roughly estimate the server capacity, in terms of the maximum number of concurrent requests that could be handled by the server and the maximum number of end devices supported by a server. The former is discussed in the following paragraphs, whereas the latter is presented in the next subsection. We estimated the maximum number of concurrent requests with the *Server Only* configuration.

The performance status of GPU is considered since the server has many CPU cores and sufficient system memory for current setting, and the GPU has relative few device memory, which are the major bottleneck in our experiments. We run three different workloads, W1, W2, and W3, on the two system configurations. W1 represents the mixed execution of both the *deep* and *shallow* models, each runs with the first 100 test samples in the test images. W2 means the execution of three *deep* and *shallow* models, whereas W3 is for five *deep* and *shallow* models. In other words, the three workloads has the concurrent execution of two, six, and ten distinct models, respectively.

During the execution of the workloads, we logged the performance data on the GPU, which are summarized in TABLE IV. In general, with the adaptive computation support, the peak GPU loading is decreased across all three workloads, where the GPU utilization keeps steady at a relatively low point. Unlike the GPU loading, the device memory usages have similar trend across the two configurations. In average, the peak memory footprint per model instance is about 285 MB, which will rise as the model number increases. Based on the observation, the maximum concurrent requests that will be handled by the GPU are about 70. Further experiments on other GPUs could be done to see if there is another GPU with better cost performance ratio. On the other hand, reducing the model size would be a good research direction to improve the number of concurrent requests. It is interesting to note that in the distributed computation scheme, the data communication incurred by a model inference is negligent, which means it will not be the bottleneck in the current setup, since the Gigabit Ethernet would sustain several thousands of the connections.

D. Entropy Threshold

The entropy threshold is used to determine if the system should take the prediction made at the exit point. When a threshold value t_i is set to one, it means the system will accept every prediction at the i-th exit point. A value of zero means that the system accepts no samples from the exit point. In practice, the setting of the threshold value affects not only the delivered system performance, but also the system capacity (i.e., the maximum number of end devices served by the server).

We ran the experiments with the *deep* model, the *Device&Server* configuration, and two entropy thresholds (0.4 and 0.8), using all the test samples in CIFAR-10. The corresponding results are shown in TABLE III, which is similar to those in TABLE II, but with the data field, Local (Server) Exit Rate, which refers to the percentage of the prediction results at the Local (Server) Exit point accepted by the system.

TABLE II
THE DELIVERED SYSTEM PERFORMANCE DELIVERED FOR THE *shallow* AND THE *deep* MODELS UNDER DIFFERENT CONFIGURATIONS.

DDNN Models	Configurations	Model Accuracy (%)				Time Decomposition (s)			
		Local	Server	Overall	Ideal	T_Device	T_Server	T_Comm.	T_Total
Shallow	**Device&Server**	45	47	48	54	25.456	0.028	0.374	25.759
Deep	**Device&Server**	44	65	65	72	0.586	0.055	0.002	0.686
	Server Only	N/A	65	65	65	N/A	0.016	0.083	0.118

TABLE III
EVALUATING THE IMPACT OF THE ENTROPY SETTINGS ON THE SYSTEM PERFORMANCE.

DDNN Models	$t_{LocalExit}$	Model Accuracy (%)				Time Decomposition (s)				Exit Rate (%)
		Local	Server	Overall	Ideal	T_Device	T_Server	T_Comm.	T_Total	Local (Server)
Deep (Device&Server)	**0.4**	44	65	65	72	0.586	0.055	0.002	0.686	2 (98)
Deep (Device&Server)	**0.8**	44	56	58	65	0.587	0.055	0.027	0.616	43 (57)

TABLE IV
SERVER LOADING MEASUREMENT WITH AND WITHOUT THE SUPPORT FROM THE PROPOSED WORK.

Perf. Data/Config.	Server Only			Device&Server		
	W1	W2	W3	W1	W2	W3
GPU Util. (%)	60	100	100	14	20∼43	30∼45
(Duration; sec)	(0.5s)	(1s)	(2s)	(3s)	(1s)	(4s)
Peak Mem. Usage (MB)	582	1,726	2,872	578	1,714	2,852

In TABLE III, the accuracies of the *deep* model with the two threshold values are almost the same, except for the *Overall Accuracy* which is affected by the exit rates. Further, the averaged time for device computation (T_Device), server computation (T_Server), data transferring (T_Comm.), and total inference time observed from the end device (T_Total) is slightly changed due to the difference in the Local Exit Rate.

In addition to the above, the threshold setting facilitates the control of the server capacity as a threshold value affects the percentage of the requests for server inference (i.e., Server Exit Rate). In our experiments, the threshold of 0.4 has around 98% of Server Exit Rate, which means most of the test samples are handled by the server to get the prediction results. On the other hand, when the value is set to 0.8, more than half of the test samples are requested for the server inference. Theoretically, the server loading is reduced by 41% via changing the threshold from 0.4 to 0.8.

After the system deployment, the administrator could leverage the MQTT to broadcast the new threshold settings for the end devices to control the number of served end devices. For instance, in the DDNN system with multiple end devices running the same DDNN model, at the some time point, the value of 0.4 could be chosen due to the higher accuracy for server inference. Later, when more end devices are added into the server, the threshold would be set to 0.8, so that the server is able to handle the requests made by the new added end devices.

V. RELATED WORK

There have been many studies done in literature, especially for cloud computing and mobile cloud computing communities, to accelerate the computations on the mobile devices with battery and/or computing power constrains [9]–[16]. Similarly, in the emerging era of Internet of Things, various types of sensors and devices are deployed in the fields to collect data and send the data to the cloud for further processing. However, sending the data to the cloud may suffer from long latencies. The common practice is introducing an intermediate device (i.e., edge or fog) between the end device and the cloud to handle the collected data as early as possible to avoid long latencies. For example, in [14], two applications (virtual machine migration and face recognition) are developed to demonstrate the advantage of fog computing, where the response time is reduced when the computation is accelerated by the fog instead of by the cloud.

The previous works on the remote computation accelerations are often bounded by the programming languages, the runtime libraries, the virtual machines, and other platforms. Taking the work done in [15] as an example, the distributed computation among various devices is performed upon the javascript language. Nevertheless, this sometimes requires the efforts for structuring the source codes of the applications in order to achieve the distributed computation, which is time-consuming and tedious work.

Fortunately, Teerapittayanon et al. [3] proposed the DDNN system which allows the distributed computation of DNN applications done at the DDNN model level. While developing the DDNN model, the model designers are allowed to map some partial DNN layers onto a distributed computing hierarchy. The trained DDNN model is able to be run across the distributed computing hierarchy by passing the intermediate results of the processed DDNN layers. Furthermore, considering the limited computing power on the end devices, their work leverages the embedded binarized neural networks (eBNNs) design [6], which is a type of neural networks with the weights of -1 or 1 in the linear and convolutional layers. In addition to the binary weight values, eBNN fuses and reorder the operations involved in the inference, so that the temporaries floating-point data are reduced during the inference.

In the contexts of distributing deep networks, aside from the DDNN work, one of the major research trends is for improving the time required by training the deep neural network [17]–[22]. The notable examples are the open source software

frameworks for deep learning, such as Caffe2, ChainerMN, and TensorFlow [17]–[19], [23], which offer a software platform for developing DNN models and allow the training process of the DNN model to be run across multiple machine nodes. In addition, there have been researches done to achieve the similar goal. In [20], Dean et al. developed a framework which is able to train a large deep network using thousands of CPUs on the computing clusters. Similarly, the techniques for training DNN models across GPU clusters have been developed [21], [22].

The proposed work is the most similar to the DDNN design [3] . In particular, this work realizes the DDNN design with the proposed software framework, upon which we built the actual distributed system as an case study that emulates the real-world surveilance applications using cameras to detect the environmental events. In other words, our work complements the DDNN design for real-world uses. The concrete contributions of this work over the previous work are given in Section II-C.

VI. Conclusion

In this work, we have designed and developed a software framework for the adaptive computation of distributed DNN applications. To demonstrate the capabilities of the framework, we have built a prototype system for the real-world surveillance application, i.e., object recognition. The trained DDNN model is fed to the end device and the server. When the end device receives an input image frame for analysis (e.g., object recognition), the device inference is performed and returns the prediction result if possible; otherwise, the server inference is taken place automatically and returns the result to the device. Note that the framework is with high applicability as the distributed inference is suitable for the trained DDNN models.

We compared our work with the commonly seen system configuration, where all of the computation is done at the server. Our preliminary results show that in some cases, the device inference completely avoids the necessity of data transmission of raw image data, which implicitly saves the resource usages, such as network bandwidth, CPU/GPU utilization and memory on the server. In addition, our results demonstrate that the model design is tightly-coupled with the system design. The model accuracy is not the only optimization goal, but system designers should consider the computing power of the end devices, the intermediate inference data to be transferred for the server inference, the computing power at the server, and the entropy threshold setting. We consider that the proposed framework is a software infrastructure that would be adopted in various IoT application fields, such as autonomous medical carts for smart hospitals and robotic carts in smart warehousing, as a plug-in subsystem to the existing systems.

References

[1] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2012, pp. 3642–3649.

[2] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big data and internet of things: A roadmap for smart environments*, 2014, pp. 169–186.

[3] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*, 2017, pp. 328–339.

[4] B. Aziz, "A formal model and analysis of the MQ Telemetry Transport protocol," in *Proceedings of the 9th International Conference on Availability, Reliability and Security*, 2014, pp. 59–68.

[5] T. Nomi, "Tiny-dnn documentation," 2018. [Online]. Available: https://media.readthedocs.org/pdf/tiny-dnn/latest/tiny-dnn.pdf

[6] B. McDanel, S. Teerapittayanon, and H. Kung, "Embedded binarized neural networks," in *Proceedings of the International Conference on Embedded Wireless Systems and Networks*, 2017, pp. 168–173.

[7] Scalagent Distributed Technologies, "Benchmark of MQTT servers," 2015. [Online]. Available: http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf

[8] A. Krizhevsky, V. Nair, and G. Hinton, "The CIFAR-10 dataset," 2014. [Online]. Available: http://www.cs.toronto.edu/kriz/cifar.html

[9] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W.-c. Feng, "VOCL: An optimized environment for transparent virtualization of Graphics Processing Units," in *Proceedings of the IEEE Conference on Innovative Parallel Computing*, 2012, pp. 1–12.

[10] C. Tu, H. Hsu, J. Chen, C. Chen, and S. Hung, "Performance and power profiling for emulated Android Systems," *ACM Transactions on Design Automation of Electronic Systems, ACM*, vol. 19, no. 2, p. 10, 2014.

[11] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Computer Systems, Elsevier*, vol. 29, no. 1, pp. 84–106, 2013.

[12] S. Hung, T. Tzeng, J. Wu, M. Tsai, Y. Lu, J. Shieh, C. Tu, and W. Ho, "MobileFBP: Designing portable reconfigurable applications for heterogeneous systems," *Journal of Systems Architecture - Embedded Systems Design, Elsevier North-Holland*, vol. 60, no. 1, pp. 40–51, 2014. [Online]. Available: https://doi.org/10.1016/j.sysarc.2013.11.009

[13] S. Hung, T. Tzeng, G. Wu, and J. Shieh, "A code offloading scheme for big-data processing in Android applications," *Software: Practice and Experience, Wiley Online Library*, vol. 45, no. 8, pp. 1087–1101, 2015. [Online]. Available: https://doi.org/10.1002/spe.2265

[14] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *Proceedings of the IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015, pp. 73–78.

[15] T. Tseng, S. Hung, and C. Tu, "Migratom.js: A Javascript migration framework for distributed web computing and mobile devices," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 798–801.

[16] S. Federico, "rCUDA: Virtualizing GPUs to reduce cost and improve performance," in *Proceedings of the STAC Summit*, 2014, pp. 1–43.

[17] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[18] A. Markham and Y. Jia, "Caffe2: Portable high-performance deep learning framework from Facebook," 2017. [Online]. Available: https://devblogs.nvidia.com/caffe2-deep-learning-framework-facebook/

[19] S. Tokui, "Introduction to Chainer: A flexible framework for deep learning," 2017. [Online]. Available: https://chainer.org/

[20] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Proceedings of the Advances in Neural Information Processing systems*, 2012, pp. 1223–1231.

[21] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer, "FireCaffe: Near-linear acceleration of deep neural network training on compute clusters," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2592–2600.

[22] J. Dean, "Large scale deep learning," in *Proceedings of the Keynote GPU Technical Conference*, 2015.

[23] T. Akiba, K. Fukuda, and S. Suzuki, "Chainermn: Scalable distributed deep learning framework," *arXiv preprint arXiv:1710.11351*, 2017. [Online]. Available: http://arxiv.org/abs/1710.11351