# 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 actually 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, the 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 exit 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 exit 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.

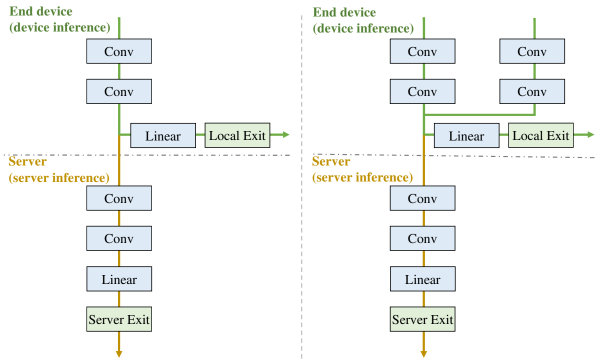


Figure 1. Two forms of the basic DDNN models developed in [3].

Compared the normalized entropy computed at the $i$-th exit point with the predetermined threshold $t_i$ which is set at the end of the training, can dynamic checking the predicted result. 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.

### A. 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 argument 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 has 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.

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.
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 field.

## 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* 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.

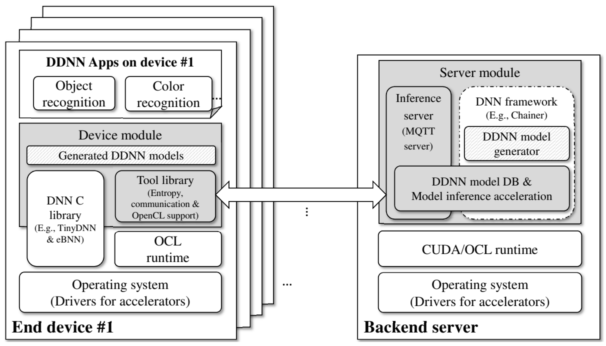

DDNN Apps on device #1
Object recognition
Color recognition
Device module
Generated DDNN models
DNN C library (E.g., TinyDNN & eBNN)
Tool library (Entropy, communication & OpenCL support)
OCL runtime
Operating system (Drivers for accelerators)
End device #1
Server module
Inference server (MQTT server)
DNN framework (E.g., Chainer)
DDNN model generator
DDNN model DB & Model inference acceleration
CUDA/OCL runtime
Operating system (Drivers for accelerators)
Backend server

Figure 2. System architecture of the proposed framework.

## 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 improve 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

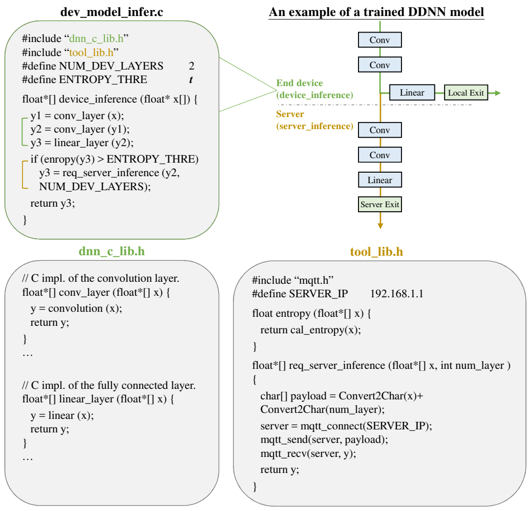
As shown in Figure 2, there are three major components in the device module: the *generated DDNN models*, the third-party implementation of *DNN C library*, and the *tool library*. Since our system aims to run on various types of embedded devices, the components within the device module are implemented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

## 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 the 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 an MQTT *client* and an 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



```
#include “dnn_c_lib.h”
#include “tool_lib.h”
#define NUM_DEV_LAYERS       2
#define ENTROPY_THRE         t

float*[] device_inference (float* x[]) {
  y1 = conv_layer (x);
  y2 = conv_layer (y1);
  y3 = linear_layer (y2);
  if (enropy(y3) > ENTROPY_THRE)
    y3 = req_server_inference (y2,
    NUM_DEV_LAYERS);
  return y3;
}
```

```
// C impl. of the convolution layer.
float*[] conv_layer (float*[] x) {
  y = convolution (x);
  return y;
}
…

// C impl. of the fully connected layer.
float*[] linear_layer (float*[] x) {
  y = linear (x);
  return y;
}
…
```

```
#include “mqtt.h”
#define SERVER_IP      192.168.1.1

float entropy (float*[] x) {
  return cal_entropy(x);
}

float*[] req_server_inference (float*[] x, int num_layer )
{
  char[] payload = Convert2Char(x)+
  Convert2Char(num_layer);
  server = mqtt_connect(SERVER_IP);
  mqtt_send(server, payload);
  mqtt_recv(server, y);
  return y;
}
```

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

published data and the subscription are matched by the *topic*, which is usually a non-empty string.

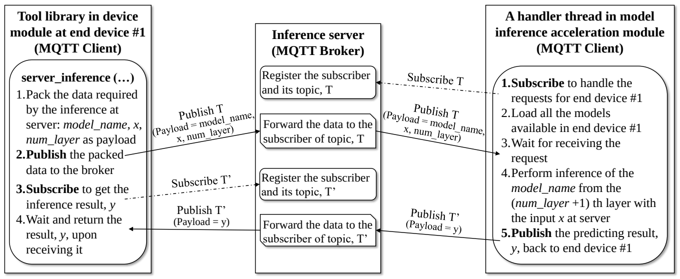

Tool library in device module at end device #1 (MQTT Client)
server_inference (…)
1.Pack the data required by the inference at server: *model_name*, *x*, *num_layer* as payload
**2.Publish** the packed data to the broker
**3.Subscribe** to get the inference result, *y*
4.Wait and return the result, *y*, upon receiving it
Inference server (MQTT Broker)
Register the subscriber and its topic, T
Forward the data to the subscriber of topic, T
Register the subscriber and its topic, T’
Forward the data to the subscriber of topic, T’
A handler thread in model inference acceleration module (MQTT Client)
**1.Subscribe** to handle the requests for end device #1
2.Load all the models available in end device #1
3.Wait for receiving the request
4.Perform inference of the *model_name* from the (*num_layer* +1) th layer with the input *x* at server
**5.Publish** the predicting result, *y*, back to end device #1
Publish T (Payload = model_name, x, num_layer)
Subscribe T’
Publish T’ (Payload = y)
Subscribe T
Publish T (Payload = model_name, x, num_layer)
Publish T’ (Payload = y)

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).

## 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 equipment and transporting goods could integrate the DDNN system with theirs to enhance their capabilities.

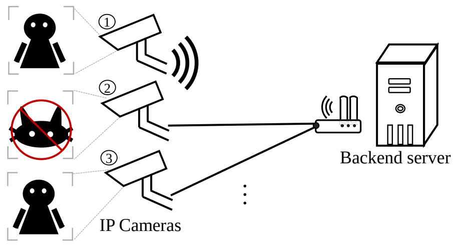


Figure 5. System architecture of the target surveillance application for object detecting, where a camera is emulated by the Odroid-XU4 and the server listed in TABLE I.

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 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 [5] 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.

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.

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** | 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 Core i7-8700 processor (w/ 12cores), 16 GB System RAM, and the NVIDIA GTX 1060 GPU (w/ 1280 cores and 6GB GDDR5X GPU RAM) |

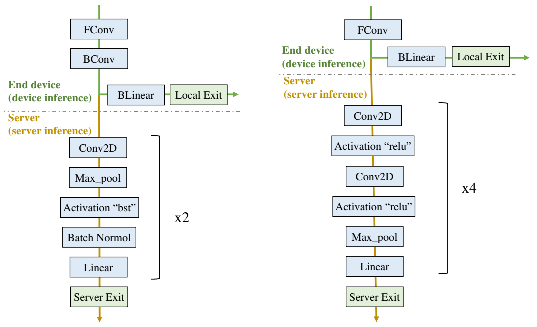


Figure 6. Two DDNN models developed in the experiments, where the left one is referred to as the *shallow* model and the right one is the *deep* model.

- *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 describe the models that we built in the process of prototyping the surveillance system and the lessons learnt during the process. In addition, we present the performance delivered by the DDNN systems.

## 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.

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 filter number has a positive impact on the accuracy.

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 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 opposed 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 of the end device. Nevertheless, the model with 64 filters has low accuracy, e.g., 59.1% 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 35 times faster than *shallow* and reports more accurate results.

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

| | | Model Accuracy (%) | | | | Time Decomposition (s) | | | |
|---|---|---|---|---|---|---|---|---|---|
| **DDNN Models** | **Configurations** | **Local** | **Server** | **Overall** | **Ideal** | **T_Device** | **T_Server** | **T_Comm.** | **T_Total** |
| ***Shallow*** | **Device&Server** | 48.5 | 46.7 | 47.1 | 59.1 | 26.290 | 0.017 | 0.127 | 26.543 |
| | **Server Only** | N/A | 47.1 | 47.1 | 47.1 | N/A | 0.015 | 0.123 | 0.143 |
| ***Deep*** | **Device&Server** | 44.2 | 65.1 | 65.6 | 72.7 | 0.615 | 0.018 | 0.126 | 0.744 |
| | **Server Only** | N/A | 65.7 | 65.7 | 65.7 | N/A | 0.022 | 0.116 | 0.125 |

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 real-time 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.

## 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 constraints [6]–[13]. Similarly, in the emerging era of IOT, the huge data collected by various types of sensors and devices deployed in the fields need cloud for further processing. However, sending the data to the cloud may suffer from long latencies. To avoid long latencies, Yi et al. [11] used an intermediate device (fog) between the end device and the cloud to handle the collected data as early as possible.

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 [14], 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 [15]–[20]. The notable examples are the open source software frameworks for deep learning, such as Caffe2, ChainerMN, and TensorFlow [15]–[17], [21], 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 [18], 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 [19], [20].

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 a case study that emulates the real-world surveillance 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-A.

## 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. 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. 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.

## Acknowledgment

This work was financially supported by the Ministry of Science and Technology of Taiwan under the grant number, 105-2218-E-006 -027 -MY2.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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

[10] 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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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/

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

[18] 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.

[19] 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.

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

[21] 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