On Designing the Adaptive Computation Framework of Distributed Deep Learning Models for Internet-of-Things Applications

XXX, XXX, and XXX, National Cheng Kung University, Taiwan

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.

CCS Concepts: • **Computer systems organization → Distributed architectures**; **Parallel architectures**; *Neural networks*; *Embedded systems*;

Additional Key Words and Phrases: Distributed systems design, distributed deep learning models design, parallel computing, Internet-of-Things applications, embedded devices, OpenCL accelerations, MQTT

ACM Reference Format:
XXX, XXX, and XXX. 2018. On Designing the Adaptive Computation Framework of Distributed Deep Learning Models for Internet-of-Things Applications. *ACM Trans. Des. Autom. Electron. Syst.* 1, 1, Article 1 (January 2018), 22 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Deep learning has demonstrated its 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 [8]. With the great successes, DNNs techniques are gradually being adopted in

Authors' address: XXX; XXX; XXX, National Cheng Kung University, Department of Computer Science and Information Engineering, Tainan, 70101, Taiwan, chiaheng@mail.ncku.edu.tw.

1084-4309/2018/1-ART1 $15.00
https://doi.org/0000001.0000001

Internet-of-Things (IoT) applications, e.g. help make better and wiser decisions [41], user behavior analysing [3], attack classification analysis [33], IOT security [11, 20, 32, 39], IOT big data analytics [27].

When adopted in IoT application fields, deep learning models tend to overwhelm computation and memory resources on the IoT devices. To alleviate the excessive amount of the resource consumptions, two classes of approaches have been developed to tackle the problem: device- and server-centric methods. The former class is related to resolve the problem by either altering the model designs with less computational complexity [13–15, 19, 22, 24, 26, 28] or developing specialized acceleration hardware to handle the required computations [5, 9, 16, 38, 41]. The latter class aims to shift the heavy computations to the back-end, resource-rich servers [29, 40], where the concept of *fog computing* [6] is a variant of the server-centric solutions to avoid the prolonged network latencies and the server loading by moving the computation to the local area network, such as a fog node or IOT gateway.

The major characteristic of the above approaches is that the DNN computations are performed at a certain site and the primary distinction between them is the distance from the data source. Nevertheless, the fixed-site computation relies on some presumptions. For example, for the device-centric approaches, it assumes that the end devices always deliver good results, whereas the server-centric ones heavy depend on the reliable network communications [30]. These systems may response poorly when the above conditions are failed.

With the end device become more powerful and server-centric desire large data movement cloud and mobile device collaborate is bring out [15, 21, 34]. Hauswald et al. [15] examines the trade-offs that executing onboard and remotely under various scenarios and later proposed Neurosurgeon [21], a scheduler that investigate computation partition strategies to automatically divide the DNN network in to mobile and could two parts. The distributed deep neural networks (DDNN) [34] had been developed to enable the DNNs computations across the sites in the distributed computation hierarchy, which may consist of end devices, edge devices, and central servers. Especially, the special forms of DNN models are developed for the adaptive inference: 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 server 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 within a distributed computing hierarchy. With our proposed framework, system designers for the IoT applications are allowed to evaluate the performance of certain model/system designs, so as to facilitate the co-design process of the deep learning models and systems. The performance evaluation of the physical systems is critical to the design of the DDNN systems, since the machines that are used to build the DDNN models are often different from the devices deployed in the field. The performance measurement, in terms of the model accuracy and the inference time, helps determine if the delivered performance meets the constraints of target applications, which is difficult to be done by the previous work [34]. We have built the prototype system for the real-world surveillance application to demonstrate the capabilities of the framework. The encouraging experimental results show that the proposed framework paves the road for examining novel designs on the 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 2. The overview and the key components of the framework are described in Section 3. Section 4 gives the system assumptions and remarks found during our design and development process. Section 5 presents the experimental

results of the built surveillance application. The previous works on the DNN computations are introduced and discussed in Section 6. Section 7 concludes this work.

2 BACKGROUND AND MOTIVATION

The distributed deep neural network (DDNN) developed by Teerapittayanon et al. [34] 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.

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.

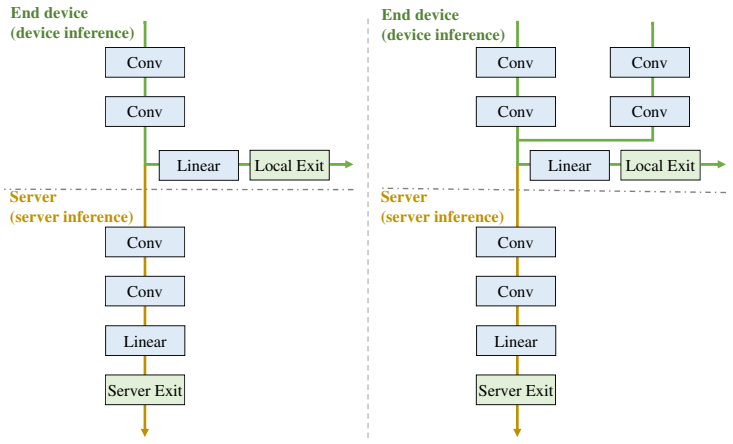

Fig. 1. Two forms of the basic DDNN models developed in [34].

2.1 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 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 joint training of the entire model. The formal definition of the model training is defined in [34].

2.2 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 [34], 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.

2.3 Motivation and Contribution

While the DDNN work [34] 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 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.

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 constraints are violated, the models should undergo another model training process with the information, in terms of the measured time/energy data. Without the performance evaluation process to guide the model training, in the worst case, the model inference might take a very long time and its battery would be drained out quickly. The key contributions of this work are as follows.

(1) Propose the software framework that enables the distributed inference of a built DDNN model to be performed adaptively on the end device, and the server, if necessary, to facilitate the DDNN systems (and also models) design and the system deployment. 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. Moreover, 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 to further reduce the inference time.

(3) Develop a general iterative evaluation method for guiding the DDNN system designs. In addition, we discuss the system design considerations for DDNN systems.

(4) Present important insights of DDNN system designs with the developed the case study on the real-world surveillance application. First, we show that the application-driven system design is critical since the model design is tightly-coupled with the delivered performance of the built system. Next, we present that the DDNN system is sufficient, in terms of the accuracy and the inference time, for the soft real-time applications, whose time constraints are within a second. Last, we discover that the model design parameter is related to the system capacity, which is important to the system design strategy.

To the best of our knowledge, we are the first work that facilitates the design of the DDNN systems and present the DDNN software framework from the perspective of the system design.

3 ADAPTIVE COMPUTATION 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 3.1 and 3.2, respectively. Section 3.3 introduces the execution model of the proposed framework.

3.1 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 [34], 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

[1]The compatibility of the proposed framework with the DNN frameworks is discussed in Section 4.1.

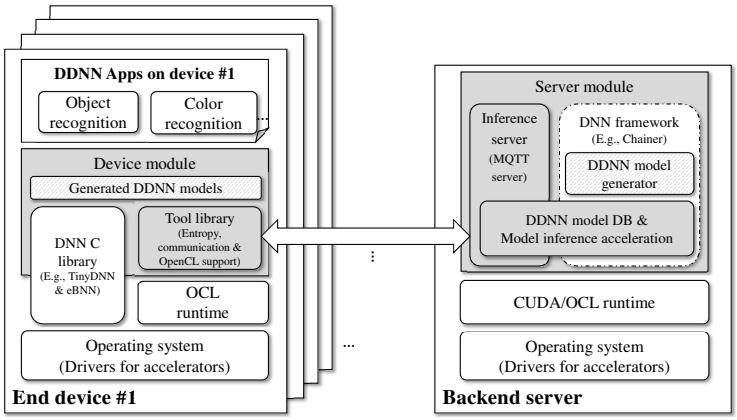

Fig. 2. System architecture of the proposed framework.

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 3.2.

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 5.

3.2 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 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 [26**?**] 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 [**?**]. 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 [26]) 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.

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

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

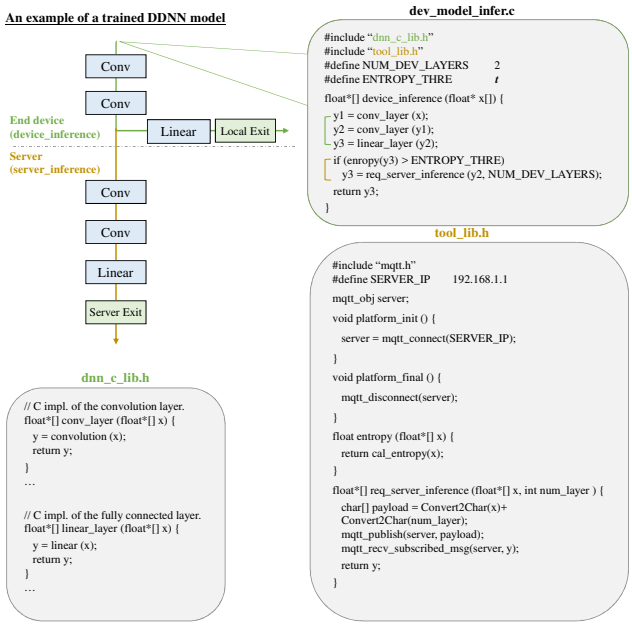

Fig. 3. Example codes for the inference performed at an end device (device inference).

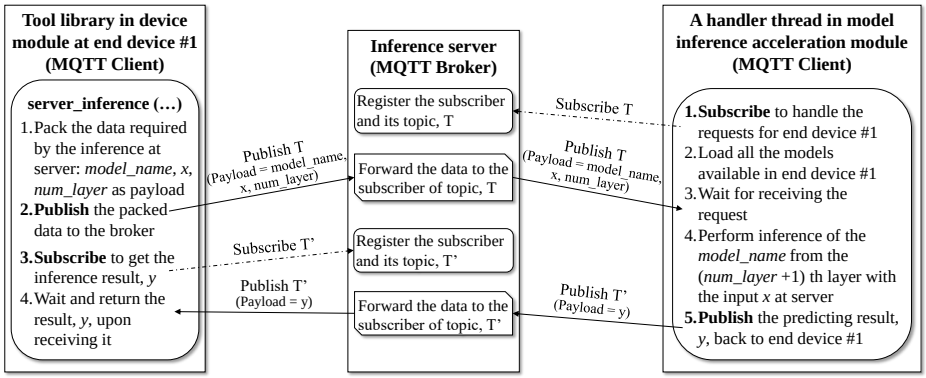

Fig. 4. Execution model of the server inference done with MQTT protocol, where the MQTT *broker* is responsible for recording the subscriptions and forwarding the published data to the subscribed MQTT *client*.

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). We discuss the potential two scaling forms enabled by the design, *scaling up* (i.e., vertical scaling) and *scaling out* (i.e., horizontal scaling), in the following paragraphs.

Scaling up 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 4.

Scaling out 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 be added into the system, each of the machines runs an acceleration module.

3.4 Iterative Evaluation Method for Exploring DDNN System Designs

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.

4 SYSTEM DESIGN REMARKS

4.1 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 [34], 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 that as the server inference is always performed starting 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 [34], 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

[2]To shorten the latency of each request, the subscription could be performed earlier during system initialization.
[3]From the related study [31], 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.

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.

4.2 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 [34] 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.

4.3 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.

4.4 Communication Cost of the Hierarchical Inference

When the hierarchical inference is taken place, the data to be transferred should be considered and analyzed to better understand the cost of the distributed inference. The costs are associated with the following items.

(1) The number of the distributed inference. It is directly linked to the distributed computing hierarchy. For example, the three-tier hierarchy may have up to two distributed inferences, whereas the two-tier hierarchy as illustrated in the left of the Figure 1 has at most one distributed inference.
(2) The number of filters for the DDNN model, f.
(3) The output size of a single filter for the last NN layer on the end device, d. For a 32 by 32 test sample, the output size of a convolution layer is 1,024.

(4) The size of a single element in output of the filter (in bytes), s. When the conventional NN layers are adopted, the data of the filter output is represented by a single precision floating-point number and in this case, s is four bytes. In our experiments, the eBNN layer is adopted and the output data of the filter is represented by the binary values. In such a case, the size is one-eighth byte.

The formula below gives the communication cost, c, when a distributed inference is occurred for a single test sample. The first item refers to the data size of six bytes for indicating the name of the DDNN model and the i-th DDNN layer for the remote inference. The second term, $(f \times d \times s)$, represents the size of the output data computed at the $(i-1)$-th DDNN layer. The third term is the result of the distributed inference returned by the machine at the higher level of the hierarchy.

$$c = 6 + (f \times d \times s) + 1$$

Note that the output sizes of the NN layers vary from one to another, depending on the layer type. For example, the output size of the convolution layer is different from that of the linear layer. In the above example, we formulate the output data size of the convolution layers, which are common NN layers and representative for communication cost analysis, since it has relatively large output data compared with other NN layers and is suitable for analyzing the communication overhead involved in the hierarchical inference. More about the actual data size and the delay required for transferring the intermediate results are presented in Section 5.

4.5 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.

5 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.

To build the surveillance system, we use the end device and the server shown in TABLE 1 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 [23] 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

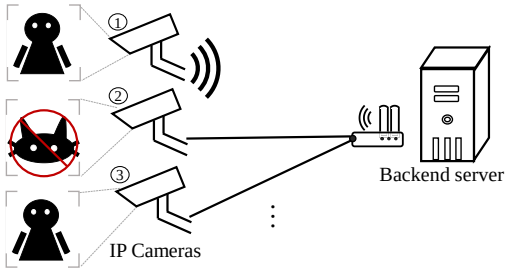

Fig. 5. System architecture of the target surveillance application for object detecting, where a camera is emulated by the Odroid-XU4 and the server listed in TABLE 1.

are served as the training data for the DDNN models, whereas the rest of them are for testing the model accuracy.

TABLE 1. 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)

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

- *Local Accuracy (Local)* is the ratio between 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 between 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 (Section 5.1). Next, we present the performance delivered by the DDNN systems (Section 5.2). Then, we show the delivered performance with different entropy thresholds and discuss the impact of the threshold value setting on the model and system capacity (Section 5.3). Finally, we reveal the potential of the OpenCL acceleration on the device inference (Section 5.4).

5.1 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. It is important to 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.

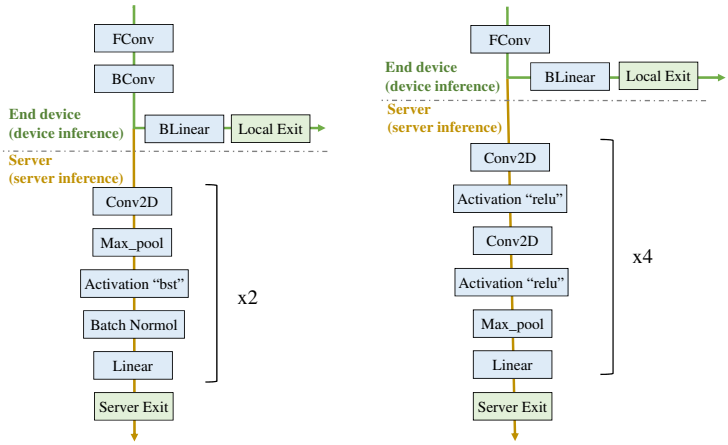

Fig. 6. Two DDNN models developed in the experiments, where the left one is referred to as the *shallow* model and the right one is the *deep* model.

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 filter number has a 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 shorten the inference time, which is further discussed in Section 5.4.

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 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.

5.2 Distributed Inference

Two inference configurations are adopted in our experiments to show the advantages of the distributed inference, in terms of the model accuracy and the inference time. One of the configurations is *Device&Server* for the distributed inference computation with the proposed framework and the other is *Server Only* for the server to handle the entire model inference operation. The latter represents the setup for the conventional surveillance system, where all the video contents are sent to the server for further process.

In the remainder of this subsection, we first present the performance of the *shallow* and *deep* models for the two configurations (TABLE 2), and discuss their respective results. Next, we further analyze the performance of the *deep* model for the *Device&Server* configuration. Last, we summarize the advantages of the proposed framework.

5.2.1 Device&Server Configuration. The *shallow* model has the BConv block for extracting the features from the sample images. We expected that the convolution layer could enhance the accuracy at the Local Exit point. Nevertheless, the model with 64 filters has low accuracy, i.e., *Overall Accuracy* is 47%, and it costs tens of seconds to perform the device inference (T_Device), which further affects the combined inference time (T_Total)[4]. It is interesting to note that the *Local Accuracy* drops only four percentage after removing the BConv block to form 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 from 46% to 65%, where the server inference time is slightly increased to reflect the fact of the heavier loads. As a result, *deep* is 36 times faster than *shallow* and reports more accurate results.

TABLE 2. 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**	98.5	46.7	47.1	59.1	25.108	0.018	0.003	25.126
	Server Only	N/A	47.1	47.1	47.1	N/A	0.020	0.018	0.040
Deep	**Device&Server**	87.8	65.1	65.6	72.8	0.551	0.023	0.001	0.582
	Server Only	N/A	65.7	65.7	65.7	N/A	0.021	0.017	0.040

5.2.2 Server Only Configuration. This configuration relies solely on the results at the Server Exit and has slightly higher accuracy, compared with the *Device&Server* 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 small packets. In addition, 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.

[4] $T\_Total \approx T\_Device + T\_Server + T\_Comm.$, where each item is the averaged time of the 10,000 samples.

5.2.3 Analysis of Deep Model for Device&Server Configuration. While the averaged accuracy of the *deep* model shown in TABLE 2 is fair, it may be difficult to apply the model for real-world uses. As the tested samples in CIFAR-10 are formed by ten image classes, we further look into the performance of each class, which is shown in TABLE 3.

The first observation is that the model accuracy varies across ten image classes, where the *Local Accuracy* could be as low as 15.2% and as high as 90.4%. The results suggest that the built model is not sufficient to cover the classification problem with wide spectrum of images, such as airplane, automobile, different animals, ship, and truck. On the other hand, the model performs relatively good for the images with larger and simpler outlines; that is, images in *Class 8* and *9* are ships and trucks, respectively, and their *Overall Accuracies* are larger than 85%.

The above results imply that the distributed inference (with the simplified NN layers at the end devices) is good for the clarified and simple problem, e.g.,, determining if the image contains a truck. This claim is supported by the result, where the device inference has more than 90% accuracy for the samples in *Class 9*. This is an encouraging result that shows the end device is capable of solving the problem by itself. Another evidence supporting the claim is that the Local Exit Rate of *Class 9* is 15.7%, which means about 157 out of 1,000 samples are handled by the end device and the server loading is alleviated remarkably. Note that the server loading is allowed to be controlled by the setting of the entropy threshold, which is discussed in Section 5.3.

We further analyze the performance of *Class 9*. The lower number of its *Server Accuracy* results in the lower *Overall Accuracy*. In addition, the higher *Ideal Accuracy* (95.1%) than its *Overall Accuracy* (85.4%) means that there are some samples, which are predicted correctly at the end device but are finally existed at the Server Exit point with wrong results due to higher entropy values at the Local Exit point. The above phenomena suggest re-designing of the NN layers at the server side would produce a better model. Nevertheless, the re-designing should be done carefully to avoid the overfitting problem.

TABLE 3. Performance of the ten image classes in CIFAR-10 with the *deep* model, *Device&Server* configuration, and entropy threshold of 0.4. The Local Exit Rate refers to the ratio between the number of samples observed at the Local Exit point and the number of total test samples.

	Model Accuracy (%)				Time Decomposition (s)				
Class	**Local**	**Server**	**Overall**	**Ideal**	**T_Device**	**T_Server**	**T_Comm.**	**T_Total**	**Local Exit Rate (%)**
0	0.0	69.6	69.0	73.6	0.552	0.023	0.001	0.584	0.9
1	30.0	76.6	75.7	81.0	0.548	0.023	0.001	0.579	2.0
2	100.0	37.9	38.0	51.9	0.555	0.023	0.001	0.587	0.1
3	60.0	70.1	70.1	75.3	0.550	0.023	0.001	0.585	0.5
4	0.0	53.2	53.2	57.1	0.547	0.023	0.001	0.576	0.0
5	92.8	55.7	56.3	73.4	0.553	0.023	0.001	0.589	1.4
6	0.0	53.4	53.4	56.1	0.554	0.023	0.001	0.587	0.1
7	93.3	68.6	69.0	75.2	0.547	0.023	0.001	0.574	1.5
8	93.7	86.4	86.6	89.3	0.550	0.023	0.001	0.580	1.6
9	100.0	82.6	85.4	95.1	0.550	0.023	0.001	0.579	15.7
Avg.	56.9	65.4	65.6	72.8	0.550	0.023	0.001	0.582	2.3

5.2.4 Summary. The above results exhibit the benefits of the collaborative inference computation. First, the accuracy of the device inference is sufficient for practical uses when the problem is simple and clearly defined, as *Class 9* for identifying the trucks in images. In addition, leveraging the end device for the inference helps alleviate the network and server loadings while the inference time is controlled within seconds. In particular, for the soft real-time applications with the latency constraints within one second, the inference done at the end device can meet the timing requirement. To further shorten the time for device inference, we have developed the OpenCL layers that can enjoy the accelerator on the platform, and the related results are given in Section 5.4.

5.3 Impact of Entropy Threshold

The entropy threshold is used to determine if the system should take the prediction made at the i-th exit point. For instance, when a threshold value t_i is set to one, it means the system will accept every prediction at the i-th exit point. On the contrary, a value of zero means that the system accepts no samples from the exit point. We use the exit rate at i-th exit point to specify the ratio between the number of samples accepted by the system at the i-th exit point and the number of total samples. For the following experiments, when referring to the models in Figure 6, we use the Local Exit Rates to represent the exit rates of the first exit points, i.e., Local Exits for *shallow* and *deep* models.

The entropy threshold controls not only the delivered accuracy but also the system capacity (i.e., the maximum number of end devices supported by the system). That is, a smaller threshold value means more rigorous considerations of the predictions produced at the early exit point, and less samples accepted at the early exit point, which means more samples are sent to the server and increases the server loading. In the remainder of this subsection, we present the impact of the entropy threshold on the model accuracy, and inference time. In addition, we discuss the relationship between Local Exit Rate and system capacity.

5.3.1 Model Accuracy and Inference Time. To analyze the impact of the threshold setting, we changed its value from 0.4 to 0.8, and present the results in TABLE 4. By comparing the averaged results at the bottom of TABLE 3 and 4, we find that the system delivers the lower accuracy, and lower inference time with the higher threshold value. In particular, the *Overall Accuracy* decreases from 65.6% to 58.2% because the *Local Accuracy* is lower than the *Server Accuracy* and the higher threshold value makes that the results of device inferences (*Local Accuracy*) dominate the overall results (*Overall Accuracy*).

The exception is the data of *Class 9*, it performs a little bit better with *Overall Accuracy* of 88.8%, which is 85.4% when the threshold is 0.4, in spite of the drops in *Server Accuracy*. This is because that the device inference is highly accurate, where 90.4% of time the Local Exit point produces correct results, and 83.9% of the samples are returned by the end device, which dominates the *Overall Accuracy*.

At the same time, the higher threshold value means less samples are sent to the server as the system is more confident with the results produced by the end device. Therefore, T_Total drops slightly from 0.744 to 0.686 second to reflect the reduction of the communication overhead (T_Comm.) and the server inference time (T_Server). Another observation is that the inference time has little to do with the input samples, and is related to the complexity of the trained model. Therefore, T_Server, T_Device, and T_Comm. for the 10 classes inputs are similar in both of TABLE 3 and 4.

5.3.2 Local Exit Rate and System Capacity. The results shown in TABLE 3 and 4 also indicate that the threshold is related to Local Exit Rate, which in turn affects the number of end devices requesting for the server inference. Taking the *Class 9* data as an example, its Local Exit Rate grows from 15.7% to 83.9% as the change of the thresholds. The delta of the exit rates indicates the server loading decreases by 68.2%, which suggests the system is able to accept about three times more requests that are similar to *Class 9* theocratically.

To further analyze the impact on the system capacity, we use the peak GPU memory footprint as the indicator to estimate the server capacity, which is then used to project the system capacity. Note that the *server capacity* refers to the maximum number of concurrent requests that can be handled by the server, whereas the *system capacity* is the maximum number of end devices supported by the system formed by the server. That is, the former is fixed with the given workloads for the system, and the latter is affected by the setting of the threshold value.

TABLE 4. Performance of the ten image classes in CIFAR-10 with the *deep* model, *Device&Server* configuration, and entropy threshold of 0.8. The Local Exit Rate refers to the ratio between the number of samples observed at the Local Exit point and the number of total test samples.

	Model Accuracy (%)				Time Decomposition (s)				
Class	**Local**	**Server**	**Overall**	**Ideal**	**T_Device**	**T_Server**	**T_Comm.**	**T_Total**	**Local Exit Rate (%)**
0	46.9	66.9	58.0	73.6	0.558	0.021	0.001	0.569	44.9
1	45.5	64.4	50.2	81.0	0.556	0.024	0.001	0.560	75.5
2	45.2	36.3	38.3	51.9	0.553	0.022	0.001	0.568	22.1
3	49.3	69.1	63.2	75.3	0.553	0.023	0.001	0.569	30.2
4	25.9	53.0	49.6	57.1	0.560	0.022	0.001	0.577	12.7
5	68.2	46.2	56.2	73.4	0.555	0.022	0.001	0.569	45.3
6	14.5	54.0	47.5	56.1	0.558	0.022	0.001	0.577	16.5
7	60.4	53.1	56.7	75.2	0.560	0.022	0.001	0.570	49.0
8	70.2	79.8	74.1	89.3	0.559	0.024	0.001	0.570	59.8
9	94.5	59.0	88.8	95.1	0.550	0.021	0.001	0.555	83.9
Avg.	52.0	58.1	58.2	72.8	0.556	0.022	0.001	0.568	43.9

The memory consumption of the GPU is used as the indicator of the server capacity since the server has many CPU cores and sufficient system memory for current setting, and the GPU has relative few device memory, which is the major bottleneck in our experiments. In particular, the server inference is mainly done by the GPU and the GPU cannot do the computation without sufficient size of memory. Hence, we profile the memory footprint of the workloads and project the maximum requests that can be handled by the server.

We logged the performance data on the GPU, and summarized in TABLE 5 for the concurrent execution of the three respective workloads under the two system configurations. The device memory usages have a 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 would be handled by the GPU are about 21 (*server capacity*). In theory, when we further consider the growth of the Local Exit Rate mentioned above, the system capacity would be increased to about 63 end devices, each of which runs the *Class 9* workloads. This example shows that the change of the entropy threshold allows the system administrator to control the system loading dynamically.

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 Gigabit Ethernet would sustain several thousands of the connections.

TABLE 5. Memory footprints on the GPU memory of the two system configurations with three different workloads. W1 represents the mixed execution of both the *deep* and *shallow* models, each runs with the first 100 test samples of the test images. W2 means the execution of three *deep* and *shallow* models, whereas W3 is for five *deep* and *shallow* models. The three workloads have concurrent executions of two, six, and ten distinct models, respectively.

	Server Only			Device&Server		
	W1	**W2**	**W3**	**W1**	**W2**	**W3**
Peak Mem. Usage (MB)	582	1,726	2,872	578	1,714	2,852

5.4 Accelerating Device Inference with OpenCL

As mentioned in Section 3.2, the proposed framework helps generate the OpenCL version of the built DDNN model in OpenCL/C with the trained model parameters. The following paragraphs show the performance improvement achieved by the automatically generated code, and describe our experiences in developing the parallel code.

TABLE 6 lists the delivered performance with and without the OpenCL acceleration[5]. The numbers of the model accuracy are the same across the two versions show that the acceleration does not compromise the accuracy. On the other hand, T_Device is significantly reduced from 0.615 to 0.318 second achieving nearly two times speedups, thanks to the GPU acceleration of the FCov block, where each of the filters is computed by a OpenCL worker in parallel. In our experiments, the BConv block used in the *shallow* model consumes too much time to be accelerated by the OpenCL. In fact, while the speedup of the BConv block is more significant than that of FConv, it still takes more than one second for the device inference, which makes the *shallow* model not ideal for practical uses.

In addition, when generating the OpenCL host program, we move its initialization and finalization codes to the platform startup and termination procedures to avoid the latencies caused by redundant creations and deletions of OpenCL objects, which costs about 0.462 second. Consequently, removing the initialization/finalization code is significant to the OpenCL acceleration in this experiment. Without removal of the redundant operations, the OpenCL version will be slower than the sequential version.

It is important to note that while the system designers are able to develop a DDNN model that achieve both high accuracy and low computation time, the OpenCL acceleration further reduce the inference time, which provide bigger opportunities to find a good model design. In our experiments, the device inference time takes only one-third second and the total inference time is cut down to less than half a second, which we believe is good for practical uses.

TABLE 6. The performance of the *deep* model with and without the OpenCL acceleration.

DDNN Models	Code Versions	Model Accuracy (%)				Time Decomposition (s)			
		Local	Server	Overall	Ideal	T_Device	T_Server	T_Comm.	T_Total
Deep	**Sequential**	87.8	65.1	65.6	72.8	0.551	0.023	0.001	0.582
	OpenCL	87.8	65.1	65.6	72.8	0.324	0.023	0.001	0.359

6 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 [10, 12, 17, 18, 36**?** , 37]. 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. To aggregated the computational power of these resource constrained devices, Hadidi et al. proposed Musical Chair [13], a localized and distributed approach can parallel computing data and DNN model to process real-time data.

To reduce the memory and disk usage, Kim et al. present a effective compression method [22] combing with rank selection and tucker decomposition to compress the entire CNN. Lei et al. [24] realize large vocabulary on-device dictation by performing on-the-fly rescoring with a compressed n-gram LM. Researchers also brought new execution model for DNN on resource-constrained devices. e.g., MCDNN [14] specialize DNNs to contexts and share resources across multiple simultaneously executing DNNs. SqueezeNet [19] a small DNN architecture need less communication, bandwidth and feasibility to deploy on FPGAs with 50X fewer parameters compare with AlexNet on ImageNet. Parvez and Bilal [28] analyzes many machine learning algorithms and implements a prototype that hardware independent as reference for deploying machine learning method on end device.

[5]The *Device&Server* configuration is adopted in the experiments.

Although the solution above have highly improved the total performance, the major bottleneck still is the low power efficiency. Deguchi et al. [9] proposed 3D-TLC (triple-level cell) NAND flash-based solidstate drive (SSD) for DNN which can extend the lifetime and reliable data storage. Hong and Parck [16] proposed an efficient accelerator for small neural networks on FPGA. Wang et al. [38] proposed on-chip memory architecture for CNN inference acceleration. Bang et al. [5] introducing stored all weights of the DNN on-chip for mobile device and a programmable deep learning accelerator(DLA).

These methods improve the power efficiency, however, the expensive specialized hardware cost a lot, offloading the data to cloud still cause lot of cautions. Sending the data to the cloud may suffer from long latencies and data unreliable. ==reliable== To reduce the communication cost. RAO et al. [30] proposed novel encryption techniques to compute data intensively and minimize the total energy cost for the application. 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 [**?**], 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 [36] 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. [34] 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 [26], 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 [1, 25, 35**? ? ?**]. The notable examples are the open source software frameworks for deep learning, such as Caffe2, ChainerMN, and TensorFlow [1, 2, 25, 35], 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 [**?**], 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 [**? ?**].

Compared with our RTCSA 2018 work [7].

7 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 programs of the trained DDNN model are fed to the end device and the server, respectively. When the end

device receives an input image frame for recognizing the object within, 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 results show that in some cases, the device inference completely avoids the necessity of the server inference since the inference done at the end device is with sufficient accuracy and done within reasonable time. The distributed inference takes advantages of the computing resources available from the end device and implicitly saves the system resource usages, such as network bandwidth, CPU/GPU utilization, and memory on the server. The performance results also reveal that the entropy threshold indirectly controls the delivered accuracy and the system capacity. Incorporating with the OpenCL acceleration, our framework is suitable for the soft real-time applications with timing constraints less than half a second. We believe that as the model design is tightly-coupled with the design of DDNN systems, especially for IoT applications, our framework helps facilitate the empirical, iterative system evaluations to search for good system design parameters, such as 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 for the system capacity and the delivered accuracy.

ACKNOWLEDGMENTS

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] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 265–283.

[2] Takuya Akiba, Keisuke Fukuda, and Shuji Suzuki. 2017. ChainerMN: Scalable distributed deep learning framework. *arXiv preprint arXiv:1710.11351* (2017). http://arxiv.org/abs/1710.11351

[3] H. Amroun, M. H. Temkit, and M. Ammi. 2017. DNN-based approach for identification of the level of attention of the TV-viewers using IoT network. In *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. 1–4. https://doi.org/10.1109/UIC-ATC.2017.8397660

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

[5] S. Bang, J. Wang, Z. Li, C. Gao, Y. Kim, Q. Dong, Y. P. Chen, L. Fick, X. Sun, R. Dreslinski, T. Mudge, H. S. Kim, D. Blaauw, and D. Sylvester. 2017. 14.7 A 288 #x00B5;W programmable deep-learning processor with 270KB on-chip weight storage using non-uniform memory hierarchy for mobile intelligence. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 250–251. https://doi.org/10.1109/ISSCC.2017.7870355

[6] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. 2014. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*. 169–186.

[7] Mu-Hsuan Cheng, QiHui Sun, and Chia-Heng Tu. 2018. An adaptive computation framework of distributed deep learning models for Internet-of-Things applications. In *Proceedings of the 24th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (to be appeared)*.

[8] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. (2012), 3642–3649.

[9] Y. Deguchi and K. Takeuchi. 2018. 3D-NAND Flash Solid-State Drive (SSD) for Deep Neural Network Weight Storage of IoT Edge Devices with 700x Data-Retention Lifetime Extention. In *2018 IEEE International Memory Workshop (IMW)*. 1–4. https://doi.org/10.1109/IMW.2018.8388776

[10] Silla Federico. 2014. rCUDA: Virtualizing GPUs to reduce cost and improve performance. In *Proceedings of the STAC Summit*. 1–43.

[11] B. Fekade, T. Maksymyuk, M. Kyryk, and M. Jo. 2017. Probabilistic Recovery of Incomplete Sensed Data in IoT. *IEEE Internet of Things Journal* (2017), 1–1. https://doi.org/10.1109/JIOT.2017.2730360

[12] Niroshinie Fernando, Seng W Loke, and Wenny Rahayu. 2013. Mobile cloud computing: A survey. *Future Generation Computer Systems, Elsevier* 29, 1 (2013), 84–106.

[13] Ramyad Hadidi, Jiashen Cao, Matthew Woodward, Michael Ryoo, and Hyesoon Kim. 2018. Musical Chair: Efficient Real-Time Recognition Using Collaborative IoT Devices. *arXiv preprint arXiv:1802.02138* (2018).

[14] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An execution framework for deep neural networks on resource-constrained devices. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*.

[15] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge. 2014. A hybrid approach to offloading mobile image classification. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 8375–8379. https://doi.org/10.1109/ICASSP.2014.6855235

[16] S. Hong and Y. Park. 2017. A FPGA-based neural accelerator for small IoT devices. In *2017 International SoC Design Conference (ISOCC)*. 294–295. https://doi.org/10.1109/ISOCC.2017.8368903

[17] Shih-Hao Hung, Tien-Tzong Tzeng, Gyun-De Wu, and Jeng-Peng Shieh. 2015. A code offloading scheme for big-data processing in Android applications. *Software: Practice and Experience, Wiley Online Library* 45, 8 (2015), 1087–1101. https://doi.org/10.1002/spe.2265

[18] Shih-Hao Hung, Tien-Tzong Tzeng, Jyun-De Wu, Min-Yu Tsai, Yi-Chih Lu, Jeng-Peng Shieh, Chia-Heng Tu, and Wen-Jen Ho. 2014. MobileFBP: Designing portable reconfigurable applications for heterogeneous systems. *Journal of Systems Architecture - Embedded Systems Design, Elsevier North-Holland* 60, 1 (2014), 40–51. https://doi.org/10.1016/j.sysarc.2013.11.009

[19] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360* (2016).

[20] P. G. Kalhara, V. D. Jayasinghearachchd, A. H. A. T. Dias, V. C. Ratnayake, C. Jayawardena, and N. Kuruwitaarachchi. 2017. TreeSpirit: Illegal logging detection and alerting system using audio identification over an IoT network. In *2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*. 1–7. https://doi.org/10.1109/SKIMA.2017.8294127

[21] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 615–629. https://doi.org/10.1145/3093337.3037698

[22] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530* (2015).

[23] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. http://www.cs.toronto.edu/kriz/cifar.html

[24] Xin Lei, Andrew W Senior, Alexander Gruenstein, and Jeffrey Sorensen. 2013. Accurate and compact large vocabulary speech recognition on mobile devices.. In *Interspeech*, Vol. 1. Citeseer.

[25] Aaron Markham and Yangqing Jia. 2017. Caffe2: Portable High-Performance Deep Learning Framework from Facebook. Retrieved April 19, 2017 from https://devblogs.nvidia.com/caffe2-deep-learning-framework-facebook/

[26] Bradley McDanel, Surat Teerapittayanon, and Hsiang-Tsung Kung. 2017. Embedded binarized neural networks. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks*. 168–173.

[27] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. 2018. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Communications Surveys Tutorials* (2018), 1–1. https://doi.org/10.1109/COMST.2018.2844341

[28] Bilal Parvez. 2017. Embedded Vision Machine Learning on Embedded Devices for Image classification in Industrial Internet of things.

[29] Li Quan, Zhiliang Wang, and Fuji Ren. 2018. A Novel Two-Layered Reinforcement Learning for Task Offloading with Tradeoff between Physical Machine Utilization Rate and Delay. *Future Internet* 10, 7 (2018), 60.

[30] SANGEETA RAO and SAURABH YADAV. [n. d.]. A NOVEL AND EFFICIENT TECHNIQUE OF COMMUNICATION IN IOTS USING AUTOENCODERS. ([n. d.]).

[31] Scalagent Distributed Technologies. 2015. Benchmark of MQTT servers. Retrieved January 2015 from http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf

[32] Cong Shi, Jian Liu, Hongbo Liu, and Yingying Chen. 2017. Smart User Authentication Through Actuation of Daily Activities Leveraging WiFi-enabled IoT. In *Proceedings of the 18th ACM International Symposium on Mobile Ad Hoc Networking and Computing (Mobihoc '17)*. ACM, New York, NY, USA, Article 5, 10 pages. https://doi.org/10.1145/3084041.3084061

[33] Bayu Adhi Tama and Kyung-Hyune Rhee. [n. d.]. Attack Classification Analysis of IoT Network via Deep Learning Approach. ([n. d.]).

[34] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2017. Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. 328–339.

[35] Seiya Tokui. 2017. Introduction to Chainer: A flexible framework for deep learning. Retrieved July 18, 2017 from https://chainer.org/

[36] Tai-Lun Tseng, Shih-Hao Hung, and Chia-Heng Tu. 2015. Migratom.js: A Javascript migration framework for distributed web computing and mobile devices. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 798–801.

[37] Chia-Heng Tu, Hui-Hsin Hsu, Jen-Hao Chen, Chun-Han Chen, and Shih-Hao Hung. 2014. Performance and power profiling for emulated Android Systems. *ACM Transactions on Design Automation of Electronic Systems, ACM* 19, 2 (2014), 10.

[38] Y. Wang, Huawei Li, and Xiaowei Li. 2016. Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–6. https://doi.org/10.1145/2966986.2967068

[39] Liang Xiao, Xiaoyue Wan, Xiaozhen Lu, Yanyong Zhang, and Di Wu. 2018. IoT Security Techniques Based on Machine Learning. *arXiv preprint arXiv:1801.06275* (2018).

[40] K. Yang, S. Ou, and H. H. Chen. 2008. On effective offloading services for resource-constrained mobile devices running heavier mobile Internet applications. *IEEE Communications Magazine* 46, 1 (January 2008), 56–63. https://doi.org/10.1109/MCOM.2008.4427231

[41] Xiaofan Zhang, Anand Ramachandran, Chuanhao Zhuge, Di He, Wei Zuo, Zuofu Cheng, Kyle Rupnow, and Deming Chen. 2017. Machine learning on FPGAs to face the IoT revolution. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design*. 819–826. https://doi.org/10.1109/ICCAD.2017.8203862

Received XXX July 2018; revised October 2018; accepted December 2018