Real-Time Image Recognition Using Collaborative IoT Devices

Ramyad Hadidi
Georgia Institute of Technology
rhadidi@gatech.edu

Jiashen Cao
Georgia Institute of Technology
jcao62@gatech.edu

Matthew Woodward
Georgia Institute of Technology
mwoodward@gatech.edu

Michael S. Ryoo
mryoo@egovid.com

Hyesoon Kim
Georgia Institute of Technology
hyesoon@cc.gatech.edu

Abstract

Internet of things (IoT) devices capture and create various forms of sensor data such as images and videos. However, such resource-constrained devices lack the capability to efficiently process data in a timely and real-time manner. Therefore, IoT systems strongly rely on a powerful server (either local or on the cloud) to extract useful information from data. In addition, during communication with servers, unprocessed, sensitive, and private data is transmitted throughout the Internet, a serious vulnerability. What if we were able to harvest the aggregated computational power of already existing IoT devices in our system to locally process this data? In this artifact, we utilize Musical Chair [3], which enables efficient, localized, and dynamic real-time recognition by harvesting the aggregated computational power of these resource-constrained IoT devices. We apply Musical chair to two well-known image recognition models, AlexNet and VGG16, and implement them on a network of Raspberry PIs (up to 11). We compare inference per second and energy per inference of our systems with Tegra TX2, an embedded low-power platform with a six-core CPU and a GPU. We demonstrate that the collaboration of IoT devices, enabled by Musical Chair, achieves similar real-time performance without the extra costs of maintaining a server.

ACM Reference Format:
Ramyad Hadidi, Jiashen Cao, Matthew Woodward, Michael S. Ryoo, and Hyesoon Kim. 2018. Real-Time Image Recognition Using Collaborative IoT Devices. In *Proceedings of 1st ACM Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning (ReQuEST at ASPLOS'18)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3229762.3229765

ReQuEST at ASPLOS'18, March 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-5923-8...$15.00
https://doi.org/10.1145/3229762.3229765

1 Extended Abstract

Musical Chair [3] is a technique for distributing deep neural network (DNN) models over several devices. DNN models have a variety of layers such as fully connected (`fc`), convolution (`conv`), batch normalization (`norm`), max pooling (`maxpool`), and activation (`act`) layers. Among these layers, fully connected and convolution layers are one the most resource-hungry and compute-intensive. Therefore, Musical Chair [3] aims at alleviating the compute cost and overcome the resource barrier of these layers by distributing their computation. In the paper, we discuss two modes of distributing these layers: data parallelism and model parallelism. Model parallelism is splitting parts of a given layer or group of layers over multiple devices, whereas data parallelism is providing the next input to multiple devices in a network. Note that we are examining this problem in the context of real-time data processing, in which we have a continuous stream of input data. In summary, for `conv` layers, data parallelism yields a higher performance in comparison with model parallelism. On the other hand, for `fc` layers, depending on the size of the input and the `fc` layer, the choice between model- and data-parallelism varies. As discussed in the paper, this is because of the resource-constrained nature of IoT devices. In this section, first, we explain our devices, then, describe Alexnet [1] and VGG16 [2] models, and finally, present Musical Chair task assignments for these models.

1.1 Technical Description

Hardware/Software Overview: In our implementation of a distributed IoT network, we use several Raspberry PIs [5], the specification of which is shown in Table 1. We choose Raspberry PI since it is a cheap and accessible platform that represents low-end IoT devices. On each PI, with the Ubuntu 16.04 operating system,

Table 1. Raspberry PI 3 specification [5]

CPU	1.2 GHz Quad Core ARM Cortex-A53	
Memory	900 MHz 1 GB RAM LPDDR2	
GPU	No GPGPU Capability	
Price	$35 (Board) + $5 (SD Card)	
Power Consumption	Idle (No Power Gating)	1.3 W
	%100 Utilization	6.5 W
	Averaged Observed	3 W

Table 2. Nvidia Jetson TX2 specifications [6].

CPU	2.00 GHz Dual Denver 2 + 2.00 GHz Quad Core ARM Cortex-A57	
Memory	1600 MHz 8 GB RAM LPDDR4	
GPU	Pascal Architecture - 256 CUDA Core	
Total Price	$600	
Power Consumption	Idle (Power Gated)	5 W
	%100 Utilization	15 W
	Averaged Observed	9.5 W

we use Keras 2.1 [7] with the TensorFlow 1.5 [8] backend. Some arbitrary Raspberry PIs are equipped with a camera [9] as well for demo purposes. But, for measurement purposes, these cameras are not required since inputs can be populated with random numbers instead of images. For power measurements, using a power analyzer, we measured the consumed power of a powered USB 3.0 hub that powers all Raspberry PIs. Moreover, timing measurements are done in the source code. To compare our collaborative IoT implementations, we also measure the performance and power consumption of our models on Jetson TX2 [6]. the specification of which is in Table 2.

Models Overview: We study single-stream Alexnet and VGG16 image recognition models, the architecture of which is shown in Figures 1 and 2, respectively.

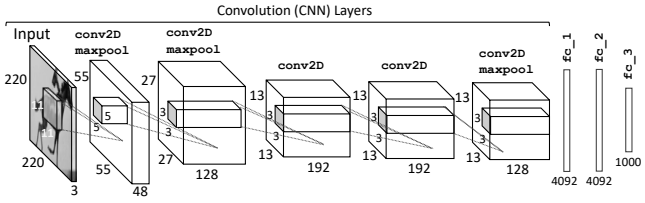

Figure 1. Single stream AlexNet model.

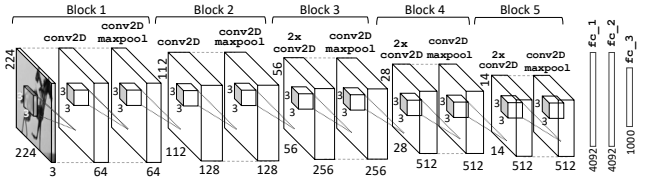

Figure 2. VGG16 model.

Distributed Models: After applying Musical Chair, for Alexnet, Figure 3 illustrates generated system architectures for five- and six-devices. In the five-device system, model parallelism is applied on the `fc_1` layer, whereas, in six-device configuration, an additional data parallelism is performed on `conv` layers. For VGG16, Figure 4 depicts two generated systems for nine and 11 devices. In both systems, `fc_1` is divided since its input size is extremely large, while, since the computation of `fc_2` and `fc_3` are not a bottleneck, Musical Chair prioritize the distribution of other layers such as `conv` layers using data parallelism.

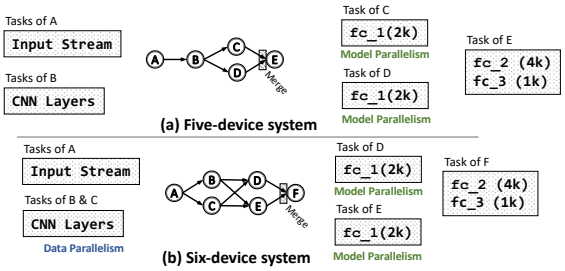

Figure 3. System architectures for AlexNet.

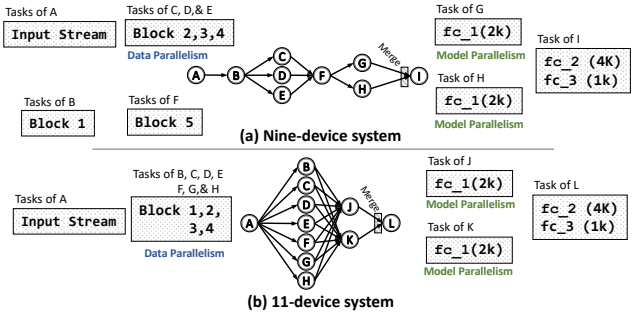

Figure 4. System architectures for VGG16.

1.2 Empirical Evaluation

Figure 5a depicts performance in terms of inference per second (IPS) for Alexnet. The performance of six-device system is similar to TX2 with CPU, while its performance is only 30% worst than TX2 with GPU. Figure 5b and c shows power consumption of all systems. As shown, the static energy consumption of the systems with Raspberry PIs are significantly higher, this is because (i) Raspberry PIs have several several unnecessary peripherals enabled, (ii) TX2 is a low-power design with power gating capability, and (iii) Raspberry PIs utilize more number of communication modules. Even so, we observe that systems with Raspberry PIs achieve better dynamic energy consumption. Figure 6 illustrates similar metrics for VGG16 model. Since VGG16 is more computationally intensive, we use more devices in our systems to achieve similar performance with TX2. When the number of devices increases from nine to 11, we achieve 2.3x better performance by reassigning all CNN blocks and performing more optimal data parallelism. In fact, compared to the TX2 with GPU, the 11-device system achieves comparable IPS (15% degradation).

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification With Deep Convolutional Neural Networks," in

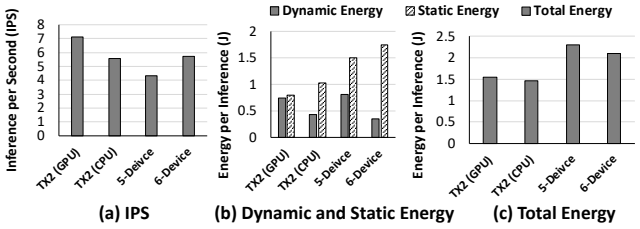

Figure 5. AlexNet: Measured IPS (a), static and dynamic energy consumption (b), and total energy consumption (c).

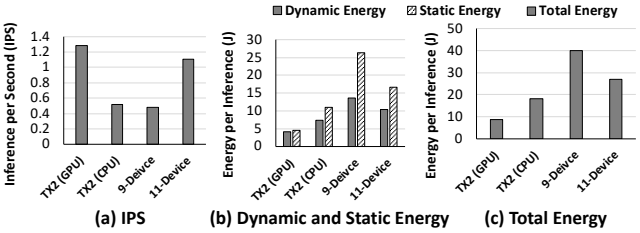

Figure 6. VGG16: Measured IPS (a), static and dynamic energy consumption (b), and total energy consumption (c).

Advances in Neural Information Processing Systems (NIPS), pp. 1097–1105, 2012.

[2] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[3] R. Hadidi, J. Cao, M. Woodward, M. Ryoo, and H. Kim, "Musical Chair: Efficient Real-Time Recognition Using Collaborative IoT Devices," *ArXiv e-prints:1802.02138*.

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[5] R. P. Foundation, "Raspberry Pi 3." https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2018. [Online; accessed 5/1/18].

[6] NVIDIA, "NVIDIA Jetson TX." http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html, 2017. [Online; accessed 5/1/18].

[7] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.

[8] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. Software available from tensorflow.org.

[9] R. P. Foundation, "Raspberry Pi 3." https://www.raspberrypi.org/products/camera-module-v2/, 2018. [Online; accessed 5/1/18].

[10] T. A. S. Foundation, "Apache Avro." https://avro.apache.org, 2018. [Online; accessed 5/1/18].

[11] NVIDIA, "NVIDIA JetPack." https://developer.nvidia.com/embedded/jetpack, 2018. [Online; accessed 5/1/18].

[12] ReQuEST at ASPLOS'18:, "1st Reproducible Tournament on Pareto-efficient Image Classification." http://cknowledge.org/request-cfp-asplos2018.html, 2018. [Online; accessed 5/1/18].

[13] Thierry Moreau, Anton Lokhmotov, Grigori Fursin, "Introducing ReQuEST: an Open Platform for Reproducible and Quality-Efficient Systems-ML Tournaments," *arXiv 1801.06378*, 2018.

A Artifact Appendix

Submission and reviewing methodology:
http://cTuning.org/ae/submission-20171101.html

A.1 Abstract

This Artifact Appendix describes experimental workflow, artifacts and results from this paper evaluated during the 1st reproducible ReQuEST tournament at the ACM ASPLOS'18:

- **Original artifact:** https://github.com/parallel-ml/asplos2018-workshop
- **Latest CK workflow:** https://github.com/ctuning/ck-request-asplos18-iot-farm
- **CK results:** https://github.com/ctuning/ck-request-asplos18-results-iot-farm
- **Artifact DOI:** https://doi.org/10.1145/3229771
- **ReQuEST submission and reviewing guidelines:** http://cknowledge.org/request-cfp-asplos2018.html ([12])
- **ReQuEST goals:** [13]
- **ReQuEST workflows:** https://github.com/ctuning/ck-request-asplos18-results
- **ReQuEST scoreboard:** http://cKnowledge.org/request-results

Our artifact provides source code for all of our implementations on a public Github repository. The source codes allows the evaluation of our results on a network of connected Raspberry PI 3s and Nvidia Jetson TX2. Note that for measuring the energy consumption, a power analyzer is needed. Additionally, to measure the energy consumption of several Raspberry PIs, we utilize a powered USB 3.0 hub and measure its energy consumption with the power analyzer.

A.2 Artifact check-list

Details: http://cTuning.org/ae/submission_extra.html

- **Algorithm:** Image recognition models of Alexnet and VGG16.
- **Program:** Written scripts in Keras framework.
- **Compilation:** Python $\geq$ 2.7.
- **Binary:** will be compiled on a target platform.
- **Data set:** Randomly generated images with Numpy (thus will not be able to test accuracy).
- **Run-time environment:** Ubuntu 16.04 ; Python version $\geq$ 2.7; Keras $\geq$ 2.1.3 with Tensorflow-gpu $\geq$ 1.5 for the backend; *(for Raspberry PI systems)* Apache Avro [10] $\geq$ 1.8.2; *(for TX2 GPU)* CUDA 8.0 with cuDNN $\geq$ 5.1.
- **Hardware:** Nvidia Jetson TX2 ; up to 11 Raspberry PI 3 with 16 GB SD cards; power analyzer; Wifi router (we use 300Mbps, 2.4 GHz 802.11n).
- **Execution:** Automated via CK command line
- **Metrics:** Inference per second; static and dynamic energy consumption.
- **Output:** Scripts output end-to-end latency. User measures power consumption during idle state and inference operations.
- **Experiments:** Performing inference on different hardware.
- **How much disk space required (approximately)?**
- **How much time is needed to prepare workflow (approximately)?**
- **How much time is needed to complete experiments (approximately)?**
- **Publicly available?:** Yes
- **Code license(s)?:** Apache 2.0
- **CK workflow framework used?** Yes
- **CK workflow URL:** https://github.com/ctuning/ck-request-asplos18-iot-farm
- **CK results URL:** https://github.com/ctuning/ck-request-asplos18-results-iot-farm
- **Original artifact:** https://github.com/parallel-ml/asplos2018-workshop

A.3 Description

A.3.1 How delivered

Our source code and scripts are available on Github: https://github.com/parallel-ml/asplos2018-workshop. A brief guide, similar to this artifact, is also available at `README.md` in the repository.

A.3.2 Hardware dependencies

We use Nvidia Jetson TX2 for the first part of the experiments. In the second part, we utilize up to 11 Raspberry PI 3s. In addition, a wifi router is necessary for the connection between Raspberry PIs. For both part, a conventional power analyzer is needed to measure the power consumption of its output. To easily measure the power consumption of Raspberry PIs, the usage of a powered USB 3.0 hub is recommended.

A.3.3 Software dependencies

Both our hardware, TX2 and Raspberry PI, are AArch64 architectures. We use Ubuntu 16.04 on both systems, however, similar Linux distribution should also work. On TX2, we use NVIDIA JetPack 3.0 to install CUDA 8.0 and cuDNN 5.1. Then, we utilize `pip`, a Python package manager, to install Keras with a Tensorflow-gpu backend, this installation procedure is similar on Raspberry PI as well. In addition, for Raspberry PI, we install Apache Avro through `pip` for managing remote procedure calls (RPC).

A.4 Installation

TX2: After installing a Linux distribution on TX2 (we use Ubuntu 16.04), install Nvidia JetPack [11] for enabling GPU support (CUDA and cuDNN). After installing Python and `pip`, install Keras through `pip`. Keras should be able to install its dependencies with `pip` automatically. If not, follow the Keras guide, the url of which is in the `README.md`.
Raspberry PI: For connivence of not repeating every step on all 11 Raspberry PIs, we suggest that performing all steps on a single Raspberry PI and then cloning its SD card. We have provided dependency file in the repo for CPU-based installation. You can execute it with bellow command to install packages:

```
pip install -r requirements.txt
```

Moreover, make sure ports number 12345 and 9999 are open on all Raspberry PIs and your router. Then, get all of the IP addresses of Raspberry PIs in your network, and fill them in `resource/ip` files under `mutiple-devices/{experiment}` dir based on Figures 3 and 4. The files are in the JSON format and each task is assigned with a list of IPs. For instance, for VGG16, in nine-device system (Figure 4a), you should have three devices for `block234` and two devices for `fc_1`.

A.5 Experiment workflow

TX2: Go to the `single-device` directory, the `predict.py` script in each model executes 50 inferences. Then, it reports the average time per inference. For energy measurements, first, measure the idle power of TX2 (static power), then, during inference time measure the consumed power (static and dynamic power). Use the commands below to execute CPU and GPU versions:

```
#GPU version
python predict.py
#CPU version
CUDA_VISIBLE_DEVICE= python predict.py
```

Raspberry PI: Go to the `multiple-devices/{experiment}` directory on each Raspberry PI. On all of devices except the initial sender, execute:

```
python node.py
```

Then, start the data sender with:

```
python initial.py
```

The initial node receives the inference responses back from the end node and prints end-to-end latency and per-layer latency for each inference. For energy measurements, first, measure the idle power of the system (we use a 14-port powered USB 3.0 hub) (static power), then, during inference time measure the consumed power (static and dynamic power).

A.6 Evaluation and expected result

The expected results should be similar to Figures 5 and 6. In these figures, IPS, shown in subfigures a, is derived by dividing one by average end-to-end latency. Note that in Raspberry systems, since network congestion affects the latency, expect around 10%–20% variation in the results. For energy measurements, our power analyzer reports

consumed power (W/s), so we multiply this reported number by end-to-end latency to derive total energy consumption per inference (J). To derive dynamic energy consumption, simply subtract static energy consumption (recorded during idle state) from total energy consumption (recorded during inference).

A.7 Unified installation and evaluation for the ReQuEST scoreboard using Collective Knowledge framework

A.7.1 Installation

Install global prerequisites (Ubuntu and similar)

```
$ sudo apt-get install libhdf5-dev
$ sudo apt-get install cython
$ sudo apt-get install python-h5py
$ sudo apt-get install python-pip
$ pip install matplotlib
$ pip install h5py
```

Minimal CK installation

The minimal installation requires:

- Python 2.7 or 3.3+ (limitation is mainly due to unitests)
- Git command line client.

You can install CK in your local user space as follows:

```
$ git clone http://github.com/ctuning/ck
$ export PATH=$PWD/ck/bin:$PATH
$ export PYTHONPATH=$PWD/ck:$PYTHONPATH
```

You can also install CK via PIP with sudo to avoid setting up environment variables yourself:

```
$ sudo pip install ck
```

Install this CK repository with all dependencies (other CK repos to reuse artifacts)

```
$ ck pull repo:ck-request-asplos18-iot-farm
```

A.7.2 Install this CK workflow from the ACM Digital Library snapshot

It is possible to install and test the snapshot of this workflow from the ACM Digital Library without interfering with your current CK installation. Download related file "request-asplos18-artifact-?-ck-workflow.zip" to a temporary directory, unzip it and then execute the following commands:

```
$ . ./prepare_virtual_ck.sh
$ . ./start_virtual_ck.sh
```

All CK repositories will be installed in your current directory. You can now proceed with further evaluation as described below.

Install or detect TensorFlow via CK

We tested this workflow with TF 1.5.

You can try to detect and use already installed TF on your machine as follows:

```
$ ck detect soft --tags=lib,tensorflow
```

Alternatively, you can install pre-built CPU version via CK as follows (please select Python 2 if several Python installations are automatically detected by CK):

```
$ ck install package --tags=lib,tensorflow,v1.5.0,vcpu,vprebuilt
```

If you plan to use NVIDIA GPU, you can install CUDA version instead:

```
$ ck install package --tags=lib,tensorflow,v1.5.0,vcuda,vprebuilt
```

If you want to build TF from sources, you can install it different versions as follows (you may need to limit the number of used processors on platforms with limited memory):

```
$ ck install package --tags=lib,tensorflow,v1.5.0,vsrc --env.CK_HOST_CPU_NUMBER_OF_PROCESSORS=1
```

Finally, you can install all available TF packages via CK as follows:

```
$ ck install package --tags=lib,tensorflow
```

Now you can install Keras via CK with all sub-dependencies for this workflow:

```
$ ck install package:lib-keras-2.1.3-request
```

A.7.3 Benchmarking on a single device (CPU)

AlexNet:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-alexnet-single-device-cpu
```

VGG16

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-vgg16-single-device-cpu
```

A.7.4 Benchmarking on a single device (GPU)

First test that CUDA-powered GPU is detected by CK:

```
$ ck detect platform.gpgpu --cuda
```

AlexNet

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-alexnet-single-device-gpu
```

VGG16

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-vgg16-single-device-gpu
```

A.7.5 Benchmarking on a farm of machines (AlexNet)

First you need to describe configuration of your farm via CK.

For example, for 5 device configuration for AlexNet, prepare JSON file with any name such as "'farm-5.json"' describing all IP addresses of your nodes:

```
{
    "node":
    {
        "initial": [
            "192.168.1.8"
        ],
        "block1": [
            "192.168.1.3"
        ],
        "block2": [
            "192.168.1.4", "192.168.1.5"
        ],
        "block3": [
            "192.168.1.6"
        ]
    }
}
```

Note that IP of "initial" node is the one where you will run benchmarking.

Now you must register this configuration in the CK with some name such as "farm-5" as follows:

```
$ ck add machine:farm-5 --access_type=avro --avro_config=farm-5.json
```

Select linux-32 or linux-64 depending on your nodes. You can view all registered configurations of target platforms as follows:

```
$ ck show machine
```

Now must log in to all your nodes and perform all above installation steps to install Python, CK, TensorFlow and Keras. Then you can start servers on all nodes (apart from "initial") as follows:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-alexnet-farm-5-nodes-start-server \
    --target=farm-5
```

Now you can run benchmark for distributed inference as follows:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-alexnet-farm-5-nodes --target=farm-5 \
   --env.STAT_REPEAT=5
```

You can change the number of repetitions using STAT_REPEAT environment variable.

A.7.6 Benchmarking on a farm of machines (VGG16, 9 nodes)

For VGG16 with 9 nodes, create "farm-9.json" and register farm-9 machine:

```
{
    "node":
    {
        "initial": [
            "192.168.1.8"
        ],
        "block1": [
            "192.168.1.3"
        ],
        "block234": [
            "192.168.1.4", "192.168.1.5", "192.168.1.6"
        ],
        "block5": [
            "192.168.1.7"
        ],
        "fc1": [
            "192.168.1.9", "192.168.1.10"
        ],
        "fc2": [
            "192.168.1.11"
        ]
    }
}
```

```
$ ck add machine:farm-9 --access_type=avro --avro_config=farm-9.json
```

Now start server on all nodes as follows:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-vgg16-farm-9-nodes-start-server \
     --target=farm-9
```

Now you can run benchmark for distributed inference as follows:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-vgg16-farm-9-nodes --target=farm-9 \
    --env.STAT_REPEAT=5
```

A.7.7 Benchmarking on a farm of machines (VGG16, 11 nodes)

For VGG16 with 11 nodes, create "farm-11.json" and register farm-11 machine:

```
{
    "node":
    {
        "initial": [
            "192.168.1.8"
        ],
        "block12345": [
            "192.168.1.3","192.168.1.4","192.168.1.5","192.168.1.6",
            "192.168.1.7","192.168.1.9","192.168.1.10"
        ],
        "fc1": [
            "192.168.1.11", "192.168.1.13"
        ],
        "fc2": [
            "192.168.1.12"
```

```
        ]
    }
}
```

```
$ ck add machine:farm-11 --access_type=avro --avro_config=farm-11.json
```

Now start server on all nodes as follows:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-vgg16-farm-11-nodes-start-server \
    --target=farm-11
```

Now you can run benchmark for distributed inference as follows:

```
$ ck run program:request-iot-benchmark --cmd_key=benchmark-vgg16-farm-11-nodes --target=farm-11 \
    --env.STAT_REPEAT=5
```

A.7.8 Scripts for unified benchmarking for ReQuEST scoreboard

You can now perform unified benchmarking and collect statistics in the CK format using scripts in the following CK entry:

```
$ cd `ck find script:benchmark-request-iot-farm`
```

If you plan to benchmark workflow on your host machine (CPU,GPU) while you already added targets for distributed inference, you must also add a "host" target to the CK as follows:

```
$ ck add machine:host --use_host
```

You can now benchmark inference on your host as follows:

```
$ python benchmarking.py --cmd_key=benchmark-alexnet-single-device-cpu
$ python benchmarking.py --cmd_key=benchmark-alexnet-single-device-gpu
$ python benchmarking.py --cmd_key=benchmark-vgg16-single-device-cpu
$ python benchmarking.py --cmd_key=benchmark-vgg16-single-device-gpu
```

You can also benchmark distributed inference using target machines farm-5, farm-9 and farm-11: (you must start servers on each node as described in previous section)

```
$ python benchmarking.py --cmd_key=benchmark-alexnet-farm-5-nodes --target=farm-5
$ python benchmarking.py --cmd_key=benchmark-vgg16-farm-9-nodes --target=farm-9
$ python benchmarking.py --cmd_key=benchmark-vgg16-farm-11-nodes --target=farm-11
```

CK will record experimental data in a unified format in the following entries:

```
$ ck ls local:experiment:ck-request-asplos18-iot-farm*
```

You can pack them and send "ckr-local.zip" to ReQuEST organizers as follows:

```
$ ck zip local:experiment:ck-request-asplos18-iot-farm*
```