Design and Development of Deep Learning Convolutional Neural Network on an Field Programmable Gate Array

Y.C.Wong, Y.Q.Lee
Micro and Nano Electronic (MINE) Research Group, Centre for Telecommunication Research & Innovation (CeTRI)
Faculty of Electronic and Computer Engineering, Universiti Teknikal Malaysia Melaka,
Hang Tuah Jaya, 76100 Durian Tunggal, Melaka, Malaysia
ycwong@utem.edu.my

***Abstract*—This paper presents the design and development of Convolutional Neural Network on Field Programmable Gate Array. In the recent work of deep learning Convolutional Neural Network, CNN is a challenging research area in both software and hardware implementation. Software implementations tend to be prohibitively slow considering that most of the neural networks run on sequentially operation architecture. Thus, the objective of this work is to design and develop deep learning CNN on FPGA based on the premise that hardware implementations that perform parallel computation of each neuron in the layers can be made faster. This work focuses on handwriting recognition where the machine has the ability to receive and interpret intelligible handwritten input from the sources. The speed of the CNN implemented on an FPGA was analyzed. Digits and numbers were successfully recognized by the developed system.**

***Index Terms*— Convolutional Neural Network, Field Programmable Gate Array.**

I. INTRODUCTION

Convolutional Neural Network (CNN) consists of one or more convolutional layers, followed by one or more fully connected layers in a standard multilayer neural network. The architecture of a CNN is designed to take the advantages of the two-dimensional structure of an input image with local connections and tied weights followed by some forms of pooling which results in translation invariant features. CNN is easy to train and has fewer parameters than the fully connected networks with the same number of hidden units. A typical CNN has three types of layer arranged in feed forward structure, namely the convolutional layer, subsampling layer and fully connected layer. CNN extracts simple features at higher resolution and converts them to complex features at lower resolution [1]. Considering the slow process involved in the implementation of CNN in software, a hardware implementation of CNN in FPGA is introduced in order to speed up the process in CNN. FPGA is the construction of programmable logic, which is not only erasable but also flexible for design. Deep learning CNN on FPGA can be applied to many applications, such as handwritten digit recognition and handwritten document recognition. It also can be applied as facial recognition system on chip, in which the design methodology can be used to integrate the entire components of a target system into a single chip so that it can be applied to one chip implementation of face recognition for wearable or mobile applications with a compact size and weight. Facial recognition on chip for wearable or mobile application can allow users to authenticate themselves by looking at the camera, allowing financial transactions. Besides that, a policeman who wears the device around the neck can automatically check the information of the person in front of him by accessing a registered database. A complete real-time face recognition system consists of a face detection, recognition and down-sampling module using FPGA [2]. According to the research [3], deep learning shows good ability in solving complex learning problem as the emerging field of machine learning. Unfortunately, the size of the networks becomes increasingly large due to the demands of the practical applications, which subsequently pose significant challenge for constructing high-performance implementations of deep learning neural networks. Recently, significant researches have been carried out in the implementation of CNN on an FPGA. The research in [8] proposed a completed FPGA-based real time face recognition system that runs at 45 frames per second with Virtex-5 FPGA.

A. Project Application

The design and development of CNN on an FPGA can be applied to many applications, such as the recognition of handwritten digits and handwritten documents. Other than digit recognition, the implementation of CNN on an FPGA has been widely used in many tasks such as image classification and object detection [4]. The proposed design is suitable for low power embedded system applications with limited memory. The general application for this work is the digits recognition implemented on an FPGA, which can be applied to an autonomous car. As FPGA is portable, it can be applied to an autonomous car, which can detect the digit or number of the available parking slot in the car park. This work contributes to the sustainable and friendly environment due to low power consumption to operate FPGA. FPGA explores high optimized reconfigurable architectures where speed up can be provided by exploring wide parallelism, deep pipeline, fast and efficient data paths. The CNN frameworks require very high computational power and large amounts of memory, while the GPU performs well on expensive machine, it is not suitable for portable devices and embedded systems [5].

Figure 1 shows the block diagram of the DE1-SoC computer, while Figure 2 shows the DE1-SoC board. Ethernet cable and mini-USB cable are needed for

connecting the DE1-SoC board. Altera Monitor Program is a good way to begin working with the DE1-SoC Computer and the ARM A9 processor.

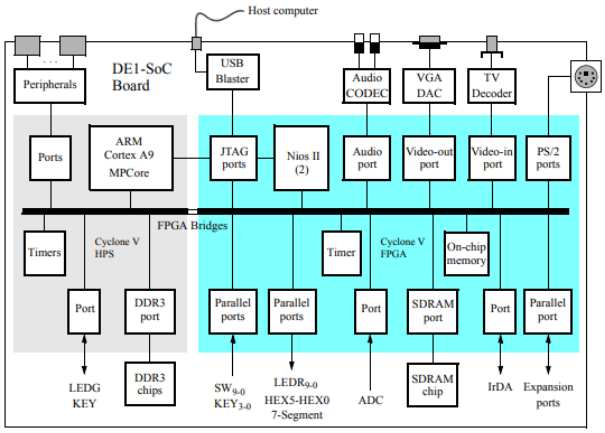

Figure 1: Block diagram of the DE1-SoC computer.

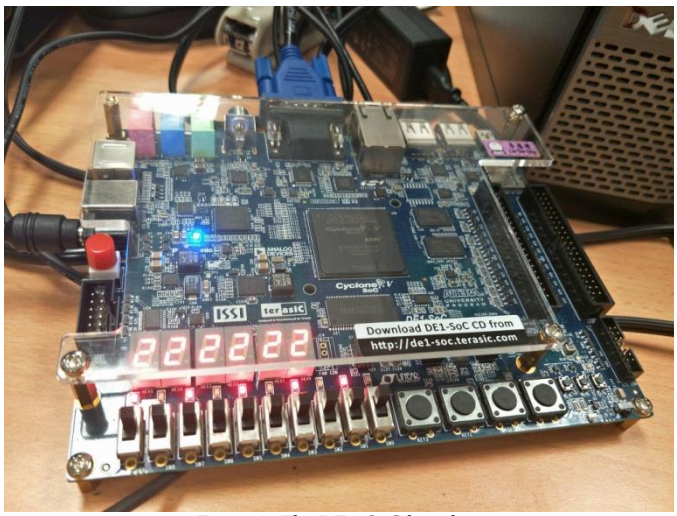

Figure 2: The DE1-SoC board.

II. METHODOLOGY

The methods used in this work is illustrated in Figure 3. This work focused on the design and development of deep learning CNN on an FPGA. First and foremost, the Linux Ubuntu 16.04 LTS was installed and it was started with Linux on the DE1-SoC board. The developed CNN in C code was analyzed and enhanced. The purpose of the C code was for the training and inference of CNN in a general way. As such, it could be used for any traditional computer vision tasks such as object classification or detection and handwritten digits recognition. The long-term goal of this code is to provide a low level, efficient and very lightweight deep learning framework to make it easy to deploy in constrained environment. The currently implemented layers include convolution, transposed convolution, fully connected, max-pooling and batch normalization. The neural network was fitted into an FPGA implemented circuit. The tools required were Altera DE1-SoC development and education board, host computer, ethernet cable, Mini-USB cable for connecting the DE1-SoC board to the host computer and the MicroSD card. The host computer was used for developing software programs that run under Linux on the DE1-SoC board. The CNN code was simulated and run successfully. After that, configuring the DE1-SoC Board was used with Linux to connect the board to the host computer. After the process of configuring the FPGA from Linux, Linux Applications was developed using FPGA hardware devices. Lastly, Linux Drivers was developed for FPGA hardware. The programs were compiled on the host computer and then the resulting executable was transferred onto the Linux filesystem, which is microSD card.

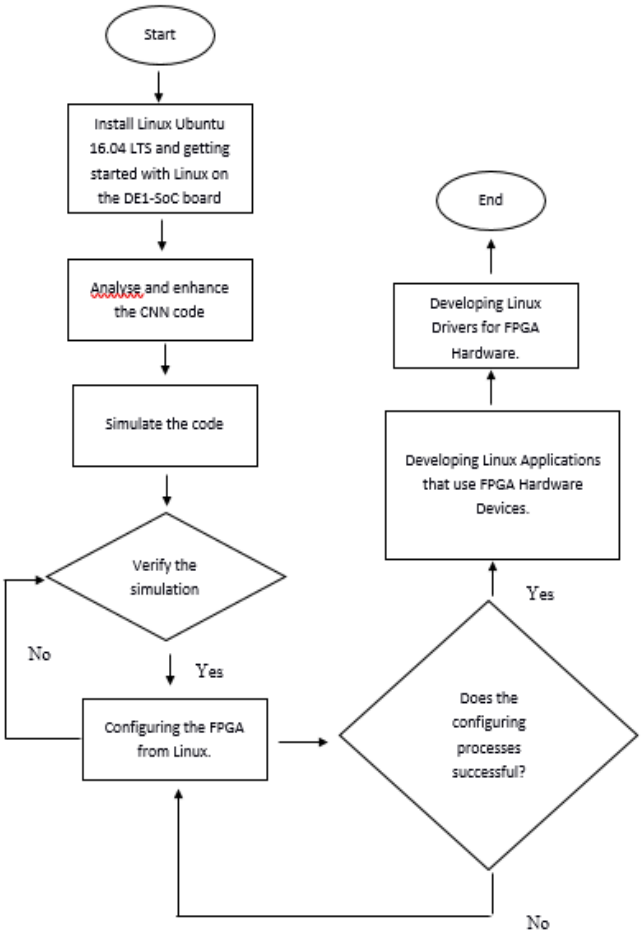

Figure 3: CNN on FPGA design flow

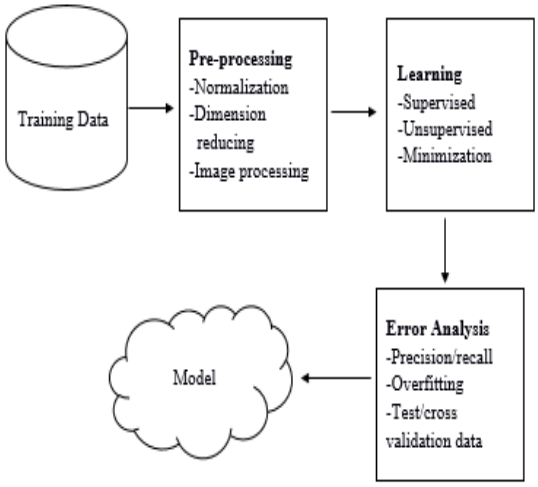

Figure 4: The steps of training data

Figure 4 shows the steps of the training data. Machine learning can be divided into two phases. The first phase is devoted for learning and the next state is for prediction. Machine recognition, description, classification and image

processing are the significant problems in variety of engineering and scientific disciplines such as biology, psychology, medicine, marketing, computer vision and artificial intelligence. Handwritten recognition is the ability of the machines that receive and interpret intelligible handwritten input from the sources. Neural network is the way people used to realize the pattern classification and image recognition. Basically, handwriting recognition system was implemented using software technology.

Once the model had been trained, the validation and testing subsets were used to predict the classification and recognition result. The prediction process was implemented to enhance the performance of the classification and detection tasks as shown in Figure 5.

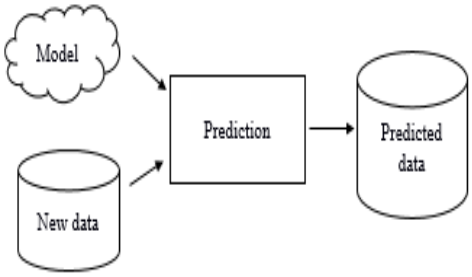

Figure 5: The steps of prediction

III. RESULT

There are two ways of implementing computations in the hardware or the software [6-8]. The software approach is the most straightforward, and the development skills are widely available. Meanwhile, the hardware approach involves the custom design of a circuit dedicated to a particular need of the application. FPGA based acceleration solution for DNN inference in [6], is realized on a SoC device where software controls the execution and off loads compute intensive operations to the hardware accelerator.

The MNIST database [7] contains 70000 standardized images of handwritten digits. The idea was to train the neural network first using the training set. After the training ended successfully, it was switched off and the effectiveness of the trained network was tested using the testing set. Each MNIST image has a size of 28 x 28 = 784 pixels. Each pixel was provided as a number between 0-255 indicating its density. Each pixel was treated as either 'ON' or 'OFF', that means black and white.

In CNN terminology, the 3x3 matrix is called a filter, kernel or feature detector. The matrix formed by sliding the filter over the image and computing the dot product is called 'Convolved Feature' or 'Feature Map'. Filter also acts as feature detectors from the original input image. CNN learns the values of these filters on its own training process. The more number of filters were used, the more image features were extracted and the network becomes better at recognizing patterns in unseen images.

Figure 6 shows the training data in the training process. Batch size is the total number of training examples presented in a single batch. The maximum batch size of this work was 400000. Batch size and number of batches are two different things. Iteration is the number of batches needed to complete one epoch. In this design, the dataset of 4000000 examples were divided into batches of 200 then it took 2000 iterations to complete 1 epoch.

```
iter= 320000 train-error= 0.014094 test-error= 12.800000 training-time= 29.349890 sec
iter= 322000 train-error= 0.013687 test-error= 12.800000 training-time= 29.516635 sec
iter= 324000 train-error= 0.013000 test-error= 12.800000 training-time= 29.221635 sec
iter= 326000 train-error= 0.014719 test-error= 12.800000 training-time= 29.300117 sec
iter= 328000 train-error= 0.013125 test-error= 12.800000 training-time= 30.289065 sec
iter= 330000 train-error= 0.012844 test-error= 12.800000 training-time= 30.212116 sec
iter= 332000 train-error= 0.013594 test-error= 12.800000 training-time= 29.654600 sec
iter= 334000 train-error= 0.013719 test-error= 16.000000 training-time= 29.238797 sec
iter= 336000 train-error= 0.013531 test-error= 12.800000 training-time= 29.746419 sec
iter= 338000 train-error= 0.012563 test-error= 16.000000 training-time= 29.475178 sec
iter= 340000 train-error= 0.013594 test-error= 12.800000 training-time= 29.561627 sec
iter= 342000 train-error= 0.012469 test-error= 12.800000 training-time= 29.785169 sec
iter= 344000 train-error= 0.013063 test-error= 12.800000 training-time= 29.454066 sec
iter= 346000 train-error= 0.013250 test-error= 12.800000 training-time= 29.544877 sec
iter= 348000 train-error= 0.014125 test-error= 12.800000 training-time= 29.642759 sec
iter= 350000 train-error= 0.012312 test-error= 12.800000 training-time= 29.796877 sec
iter= 352000 train-error= 0.014562 test-error= 12.800000 training-time= 29.706440 sec
iter= 354000 train-error= 0.012469 test-error= 12.800000 training-time= 29.824016 sec
iter= 356000 train-error= 0.013719 test-error= 16.000000 training-time= 29.604065 sec
iter= 358000 train-error= 0.013656 test-error= 12.800000 training-time= 29.553713 sec
iter= 360000 train-error= 0.013344 test-error= 12.800000 training-time= 29.588724 sec
iter= 362000 train-error= 0.012938 test-error= 12.800000 training-time= 29.590169 sec
iter= 364000 train-error= 0.013844 test-error= 12.800000 training-time= 30.079900 sec
iter= 366000 train-error= 0.013125 test-error= 12.800000 training-time= 29.685372 sec
iter= 368000 train-error= 0.012750 test-error= 12.800000 training-time= 29.664731 sec
iter= 370000 train-error= 0.013438 test-error= 12.800000 training-time= 29.587000 sec
iter= 372000 train-error= 0.013031 test-error= 16.000000 training-time= 29.588430 sec
iter= 374000 train-error= 0.013875 test-error= 12.800000 training-time= 29.610835 sec
iter= 376000 train-error= 0.013375 test-error= 16.000000 training-time= 29.540623 sec
iter= 378000 train-error= 0.013250 test-error= 12.800000 training-time= 29.611244 sec
iter= 380000 train-error= 0.011969 test-error= 16.000000 training-time= 29.495788 sec
iter= 382000 train-error= 0.013656 test-error= 12.800000 training-time= 30.396249 sec
iter= 384000 train-error= 0.012750 test-error= 12.800000 training-time= 29.977081 sec
iter= 386000 train-error= 0.014562 test-error= 12.800000 training-time= 29.045160 sec
iter= 388000 train-error= 0.012344 test-error= 16.000000 training-time= 29.017070 sec
iter= 390000 train-error= 0.013531 test-error= 12.800000 training-time= 29.015059 sec
iter= 392000 train-error= 0.012375 test-error= 12.800000 training-time= 29.056282 sec
iter= 394000 train-error= 0.014062 test-error= 12.800000 training-time= 29.012579 sec
iter= 396000 train-error= 0.012625 test-error= 16.000000 training-time= 29.125821 sec
iter= 398000 train-error= 0.013656 test-error= 12.800000 training-time= 29.014347 sec
[INFO] Training ended successfully
```

Figure 6: Part of the process of training data

Figure 7 shows the training error against the iteration. The training error is the error that emerges when the trained model is run back on the training data. According to Figure 7, the train error continued to decrease with the increase of iteration.

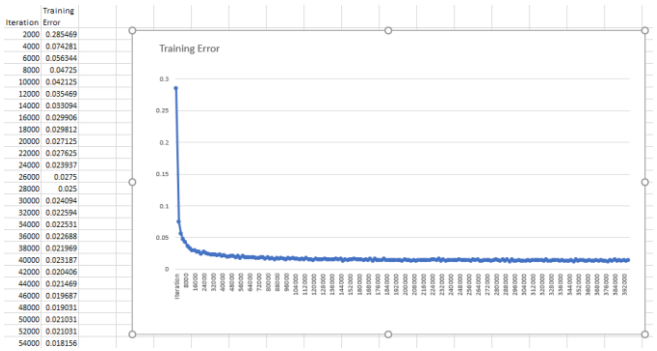

Figure 7: Training error against the iteration

Figure 8 shows the test error against the iteration. Test error is the error when the trained model is run on a set of data that has never been exposed. This data is usually used to measure the accuracy of the model before it is shipped to prediction.

Figure 8: Test error against the iteration

Figure 9 shows the process of testing images. The python script was run to fold all the pictures and categories into single binary pictures. Then it appeared as ubyte files ready to tar. Figure 10 shows the selected testing images in

portable network graphics (PNG) files.

```
yanqing@yanqing-S551LN:~/jpg_to_mnist$ python convert-images-to-mnist-format.py
./training_images/1/0.png
./testing_images/1/2.png
./testing_images/1/7.png
./testing_images/1/3.png
./testing_images/1/9.png
./testing_images/1/0.png
gzip: train-images.idx3-ubyte.gz already exists; do you wish to overwrite (y or n)? y
gzip: train-labels.idx1-ubyte.gz already exists; do you wish to overwrite (y or n)? y
gzip: test-images.idx3-ubyte.gz already exists; do you wish to overwrite (y or n)? y
gzip: test-labels.idx1-ubyte.gz already exists; do you wish to overwrite (y or n)? y
yanqing@yanqing-S551LN:~/jpg_to_mnist$
```

Figure 9: The process of testing images

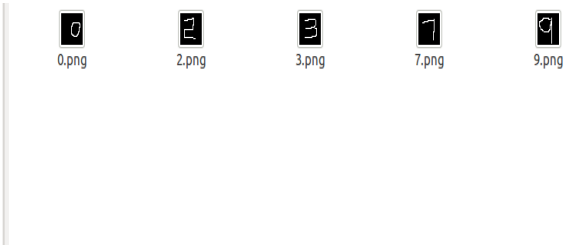

Figure 10: The selected testing images in portable network graphics (PNG) files

As shown in Figure 11, each column represents one image file, so the prediction file appears five columns. Each row represents one number. There are ten rows which represent zero until nine. The testing images that matched with the results are shown in the prediction file. The accuracy for each number is different. The datasets for the training examples can be increased to improve the accuracy.

```
0.000015 0.000260 0.999705 0.000001 0.000001 0.000001 0.000002 0.000004 0.000011 0.000001
0.000006 0.000049 0.000095 0.000039 0.000006 0.000016 0.000006 0.999562 0.000116 0.000102
0.000366 0.129854 0.047737 0.817148 0.000217 0.000217 0.000217 0.001123 0.002807 0.000314
0.000070 0.000014 0.000011 0.000004 0.000369 0.000004 0.000004 0.000572 0.000014 0.998939
0.467127 0.002623 0.058895 0.005034 0.002623 0.024397 0.032717 0.011134 0.272263 0.123188
```

```
yanqing@yanqing-S551LN: ~/jpg_to_mnist
n)? n
        not overwritten
gzip: test-images.idx3-ubyte.gz already exists; do you wish to overwrite (y or n
)? n
        not overwritten
gzip: test-labels.idx1-ubyte.gz already exists; do you wish to overwrite (y or n
)? n
        not overwritten
yanqing@yanqing-S551LN:~/jpg_to_mnist$ python convert-images-to-mnist-format.py
./training_images/1/0.png
./testing_images/1/2.png
./testing_images/1/7.png
./testing_images/1/3.png
./testing_images/1/9.png
./testing_images/1/0.png
```

Figure 11: The results in prediction file

```
yanqing@yanqing-S551LN: ~/bcnn/examples/mnist_cl
[Activation] input_shape= 28x28x16 type= Relu output_shape= 28x28x16
[Maxpool] input_shape= 28x28x16 size= 2 stride= 2 ouput_shape= 14x14x16
[Convolutional] input_shape= 14x14x16 nb_filters= 16 kernel_size= 3 stride= 1 pa
dding= 1 output_shape= 14x14x16
[Activation] input_shape= 14x14x16 type= Relu output_shape= 14x14x16
[Maxpool] input_shape= 14x14x16 size= 2 stride= 2 ouput_shape= 7x7x16
[Connected] input_shape= 7x7x16 output_shape= 1x1x256
[Activation] input_shape= 1x1x256 type= Relu output_shape= 1x1x256
[Connected] input_shape= 1x1x256 output_shape= 1x1x10
[Activation] input_shape= 1x1x10 type= Relu output_shape= 1x1x10
[Softmax] input_shape= 1x1x10 output_shape= 1x1x10
lr= 0.003000 m= 0.900000 decay= 0.000500 seen= 6400000
layer= 0 nbread_bias= 16 bias_size_expected= 16
layer= 0 nbread_weight= 144 weight_size_expected= 144
layer= 3 nbread_bias= 16 bias_size_expected= 16
layer= 3 nbread_weight= 2304 weight_size_expected= 2304
layer= 6 nbread_bias= 256 bias_size_expected= 256
layer= 6 nbread_weight= 200704 weight_size_expected= 200704
layer= 8 nbread_bias= 10 bias_size_expected= 10
layer= 8 nbread_weight= 2560 weight_size_expected= 2560
[INFO] Model model.dat loaded succesfully
[INFO] Start prediction...
time for proccessing all images = 38  ms[INFO] Prediction ended successfully
yanqing@yanqing-S551LN:~/bcnn/examples/mnist_cl$
```

Figure 12: The speed for processing all the images in GPU.

```
ubuntu@DE1_SoC: ~/FYP/bcnn/examples/mnist_cl
File  Edit  Tabs  Help
[Activation] input_shape= 28x28x16 type= Relu output_shape= 28x28x16
[Maxpool] input_shape= 28x28x16 size= 2 stride= 2 ouput_shape= 14x14x16
[Convolutional] input_shape= 14x14x16 nb_filters= 16 kernel_size= 3 stride= 1 pa
dding= 1 output_shape= 14x14x16
[Activation] input_shape= 14x14x16 type= Relu output_shape= 14x14x16
[Maxpool] input_shape= 14x14x16 size= 2 stride= 2 ouput_shape= 7x7x16
[Connected] input_shape= 7x7x16 output_shape= 1x1x256
[Activation] input_shape= 1x1x256 type= Relu output_shape= 1x1x256
[Connected] input_shape= 1x1x256 output_shape= 1x1x10
[Activation] input_shape= 1x1x10 type= Relu output_shape= 1x1x10
[Softmax] input_shape= 1x1x10 output_shape= 1x1x10
lr= 0.003000 m= 0.900000 decay= 0.000500 seen= 6400000
layer= 0 nbread_bias= 16 bias_size_expected= 16
layer= 0 nbread_weight= 144 weight_size_expected= 144
layer= 3 nbread_bias= 16 bias_size_expected= 16
layer= 3 nbread_weight= 2304 weight_size_expected= 2304
layer= 6 nbread_bias= 256 bias_size_expected= 256
layer= 6 nbread_weight= 200704 weight_size_expected= 200704
layer= 8 nbread_bias= 10 bias_size_expected= 10
layer= 8 nbread_weight= 2560 weight_size_expected= 2560
[INFO] Model model.dat loaded succesfully
[INFO] Start prediction...
time for processing all images =   967 ms[INFO] Prediction ended successfully
ubuntu@DE1_SoC:~/FYP/bcnn/examples/mnist_cl$
```

Figure 13: The speed for processing all the images in ARM Cortex A9 in DE1-SoC board.

The speed for processing all the images in GPU was 38ms as shown in Figure 12, while the speed for processing all the images in ARM Cortex A9 in DE1-SoC board was 967ms, as shown in Figure 13. FPGA technology can be evolved fast. Theoretically, the FPGA with lower power consumption requires less thermal dissipation countermeasures; hence, it implements the solution in smaller dimensions.

The very basic of FPGA is more flexible than most microcontrollers. The term field programmable means the FPGA can be reprogrammed to do any task that can be fitted into the number of its gates. The power of FPGA is consumed more than the typical power of microcontrollers.

IV. Discussion

Recognizing digits is not an easy task. Deep learning CNN performed better than the other methods as it achieved higher accuracy. The idea is to take a large number of handwritten digits, known as training examples and then develop a system, derived from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. The accuracy for each number is different. Furthermore, by adding the number of training examples, the network can learn more and the accuracy could be improved.

The coding used in this work is C code. It can convert a set of jpg or png images into MNIST binary format. It can rescale all the jpg and png images in the folders the MNIST standard 28 x 28-pixel size. The python script is run to fold all the pictures and categories into single binary pictures. Then it will appear as ubyte files ready to tar. The long-term goal of this work is to provide a low level, efficient and a very lightweight deep learning framework to make it easy to be deployed in constrained environment.

The design and development of CNN on an FPGA can be applied to many applications such as the recognition of handwritten digits and handwritten documents. The general application for this work is focused on the digits or numbers recognition implemented on an FPGA, which can be applied to an autonomous car to detect the number of available parking slot in the car park.

V. CONCLUSION

In conclusion, the digits had been successfully recognized by the system since the results can be observed in the prediction file. However, the speed for processing all the images in ARM Cortex A9 in DE1-SoC board is lower than processing in GPU. The developed system uses only 38ms to recognize a numbers but longer time is needed in FPGA with 967ms. However, the FPGA is portable, hence it is suitable to apply to autonomous cars which can detect the to the numbers or digits to find the available parking slot in car park. This work can be further enhanced to object detection to detect the surrounding object, human and even animals.

ACKNOWLEDGEMENT

The authors acknowledge the technical and financial support by Universiti Teknikal Malaysia Melaka (UTeM) and Ministry of Science, Technology and Innovation Malaysia's grant no. 01-01-14-SF0133//L00029.

REFERENCES

[1] J. Wang, J. Lin and Z. Wang, "Efficient Hardware Architectures for Deep Convolutional Neural Network," in IEEE Transactions on Circuits and Systems I, vol. 65, no. 6, 2018, pp. 1941-1953.

[2] J. Matai, A. Irturk and R. Kastner, "Design and Implementation of an FPGA-based Real Time Face Recognition System". 19th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 2011, pp. 97-100.

[3] C. Wang, L. Gong, Q. Yu, X. Li, Y. Xie and X. Zhou, "DLAU: A Scalable Deep Learning Accelerator Unit on FPGA", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no.3, 2017, pp. 513-517.

[4] Y. Zhou, S. Redkar and X. Huang, "Deep Learning Binary Neural Network on an FPGA", 60th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2017, pp. 281-284.

[5] F. Yi, H. Xiao, S. Yongjie, "FPGA Accelerating Core Design Based on XNOR Neural Network algorithm", MATEC Web of Conference (SMIMA), 2018, pp. 1-5.

[6] L. Ruo, "A framework for FPGA-Based Acceleration of Neural Network Inference with Limited Precision via High-Level Synthesis with Streaming Functionality", M.S. theses, University of Toronto, 2016.

[7] Y. LeCun, C. Cortes, C.J.C. Burges, "MNIST handwritten digit database", [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 24- Aug- 2018].

[8] J. Matai, A. Irturk and R. Kastner, "Design and Implementation of an FPGA-based Real Time Face Recognition System", 19th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 2011, pp. 97-100.