Journal of Ambient Intelligence and Humanized Computing (2020) 11:2309–2323
https://doi.org/10.1007/s12652-019-01357-4

ORIGINAL RESEARCH

A hybrid GPU-FPGA based design methodology for enhancing machine learning applications performance

Xu Liu[1] · Hibat-Allah Ounifi[2] · Abdelouahed Gherbi[2] · Wubin Li[3] · Mohamed Cheriet[1]

Received: 2 January 2019 / Accepted: 5 May 2019 / Published online: 13 June 2019

Abstract
The high-density computing requirements of machine learning (ML) is a challenging performance bottleneck. Limited by the sequential instruction execution system, traditional general purpose processors are not suitable for efficient ML. In this work, we present an ML system design methodology based on GPU and FPGA to tackle this problem. The core idea of our proposal is when designing an ML platform, we leverage the graphics processing unit (GPU)'s high-density computing to perform model training and exploit field programmable gate array (FPGA)'s low-latency to perform model inferencing. In between, we define a model converter, which enable transforming the model used by the training module to one that is used by inferencing module. We evaluated our approach through two use cases. The first is a handwritten digit recognition with convolutional neural network while the second use case is for predicting data center's power usage effectiveness with deep neural network regression algorithm. The experimental results indicate that our solution can take advantages of GPU and FPGA's parallel computing capacity to improve the efficiency of training and inferencing significantly. Meanwhile, the solution preserves the accuracy and the mean square error while converting the models between the different frameworks.

Keywords Machine learning · High performance computing · Heterogeneous computing · Hybrid platform · GPU computing · FPGA computing · CNN · DNN · Model converting · PUE

1 Introduction

Recent massive adoption of machine learning applications, e.g., prediction of PM2.5 (Ganesh et al. 2018) and mural deterioration detection (Huang et al. 2017), are commonly based on neural network algorithms such as multilayer perceptron (MLP) or CNN. These algorithms, in general, have to face substantial training data and consuming much time to achieve high accuracy, which is a challenge for GPPs (such as CPU) which have few computing cores and execute instructions in sequence and thus are worse at high-density computing tasks.

Meanwhile, people found that matrix operations almost dominate the ML algorithms. We can decompose the complex matrix operations into duplicated and straightforward atom operations such as additions and multiplications. Base on these facts and previous GPPs' shortages, people, try to find some other hardware devices which have much more parallel computing units. GPUs and FPGAs become the natural choice as they all have lots of computing units which can be re-programmed to work in parallel to increase the speed of machine learning.

A classical machine learning process includes two main phases, a training phase, and an inferencing phase. During the training phase, we pour tens of thousands of training data sets into the neural network, and the distance between the ground truth value and the prediction value will decrease continuously, and when the accuracy reach our goal, we stop

✉ Abdelouahed Gherbi
Abdelouahed.Gherbi@etsmtl.ca

Xu Liu
xu.liu.1@ens.etsmtl.ca

Hibat-Allah Ounifi
hibat-allah.ounifi.1@ens.etsmtl.ca

Wubin Li
wubin.li@ericsson.com

Mohamed Cheriet
mohamed.cheriet@etsmtl.ca

[1] Synchromedia Laboratory, University of Québec (ÉTS), Montréal, Canada

[2] University of Québec (ÉTS), Montréal, Canada

[3] Ericsson Research, Ericsson, Montréal, QC, Canada

the training and get the model. In the inferencing phase, we use the previous model to do prediction or estimation. When the training and inferencing are not on the same framework, for example, training on the Tensorflow and inferencing on Caffe, a model converter is necessary to convert the model from one framework to another.

Currently, there have been extensive studies on how to use only GPUs to accelerate the training phase (Raina et al. 2009; Bergstra et al. 2011; Sharp 2008; Potluri et al. 2011) or how to use only FPGAs to accelerate the inferencing phase (Motamedi et al. 2016; Qiu et al. 2016; Nagarajan et al. 2011; Aydonat et al. 2017), however, the investigation on how to combine the GPUs and FPGAs to improve the performance of the ML system, as well as what kind of hardware devices combination has the best performance remain mostly unexplored.

GPUs and FPGAs have different architectures, different characteristics, and performance. In our paper, we first analyzed the advantages and disadvantages when using GPUs and FPGAs individually to implement ML's various components, training, and inferencing. Then we gave out our design methodology of the ML system, and next, we performed two real ML cases, hand digital recognization, and prediction of datacenter's PUE based on our design methodology and tested and compared their performances. In the end, we made a discussion about the two experiments' results and gave out our viewpoints.

We organize the rest of our paper as follows: Sect. 2 presents related work about ML acceleration with different kinds of hardware devices. Sect. 3 elaborates the details of our hybrid design methodology of the ML system. Sects. 4 and 5 describe the implementations and performances of a CNN digital handwriting classification and a DNN PUE prediction based on our design methodology individually. Section 6 discusses the two cases' differences and problems we met and gives some explanations. In Sect. 7, we conclude our work and briefly discuss our future research plan.

2 Related work

People have studied ML acceleration with GPU extensively. For example, Steinkraus et al. (2005) implemented a full connected two layers neural network on ATI Radeon X800 graphic card and achieved 3× speedup in training and testing. However, they published the paper in 2005; the primary method of using pixel shaders for ML computation is outdated. Similarly, Raina et al. (2009) used GPU to accelerate the two unsupervised learning algorithms, including deep belief networks (DBNs) and sparse coding. They took advantages of GPUs' global memory to save the data and parameters from reducing the transfer time between the host machine and the GPU, resulting in performance improvement with 70 times faster than a dual-core CPU when implementing the DBNs algorithm. Their result demonstrated the potential of acceleration using GPUs, which is also confirmed by Potluri et al. (2011) in their work where Potluri et al. have used GeForce 9500 GT with 256MB memory graphic card to speed up GPU-based Universal Machine-CNN (UM-CNN).

On the adoption of FPGAs to accelerate ML algorithms, Motamedi et al. (2016) presented an accelerator for deep CNNs. Their accelerator can exploit the available parallelism resources to minimize the execution time, achieving a 1.9× speedup comparing with the state-of-the-art deep CNN accelerator. Aydonat et al. (2017) proposed a new architecture designed with OpenCL. They tried to increase data reusing to reduce external memory bandwidth consumption. They also used the Winograd transform to improve the FPGAs' performance. They managed to speed up executing the AlexNet CNN benchmark on Intel's Arria 10 FPGA, 10× faster than the state-of-the-art on FPGAs and the power efficiency is similar to the best implementation of AlexNet on TitanX GPU. Nagarajan et al. (2011) has proposed a method to implement a multi-dimensional PDF estimation algorithm which used Gaussian kernels on the FPGA. They used ActiveHDL to develop their platform. It is a hardware description language (HDL) and is not so popular and not so easy to use. They have got a 20× speedup over a 3.2 GHz CPU processor.

To the best of our knowledge, few works combined FPGA and GPU to improve the performance of ML systems in the past. The first work we found is a FPGA-GPU architecture for kernel SVM pedestrian detection by Bauer et al. (2010). They used GPU for model training and inferencing and used FPGA for feature extraction. In the second work, Zhu et al. (2016) have implemented a novel parallel framework for neural networks with GPU and FPGA. In their work, the neural network processing was decomposed into layers and scheduled either on the GPU or FPGA accelerators. Additionally, in a white paper (Rush et al. 2017), without giving detailed information, the authors only gave a hypothesis that the combination of CPU, GPU and FPGA would have the best performance, but did not give any verification.

In summary, few studies have combined GPUs and FPGAs together to implement ML systems, not to mention how to convert models from one framework to another.

3 Design methodologies

3.1 The whole architecture and workflow

A standard ML system usually has two main modules; one is for training model, the other is for inferencing model. Model

is the belt between the training module and the inferencing module.

Figure 1 presents the whole architecture of our ML system, which contains three modules, i.e., training module, inferencing module, and converting-model module. We implemented the training module on the GPU and built the inferencing module on the FPGA. As the FPGA inferencing framework is different from the GPU training framework, a model converting component is needed.

The entire flow is as follows. The training module loads training data from the database first and then uses ML to train the model. When the model is ready, the system converts the model from the training framework to the inferencing framework. In the end, the inferencing module loads the model file and the inferencing data and does classification or prediction.

3.2 The GPU training module

The goal of the training phase is to train a model to obtain the maximum test accuracy within minimum time. Based on two reasons, we select the GPU to implement the training module. One reason is, comparing with FPGAs, GPUs often own lower price/performance ratio in model training. So most research projects select GPUs to do training (Steinkraus et al. 2005; Raina et al. 2009; Potluri et al. 2011), and few projects choose FPGAs to train their model (Zhao et al. 2016). The other reason is the training module usually has complex architectures (forward and backward propagation, gradient descent, and so on) and its goal is to get the model, once get the model, the training module is useless, so people hope to build the training module quickly. On this point, GPUs are much easier to program than FPGAs. As GPUs have similar instruction system as CPUs, the programs run on CPUs can be easily transplanted to GPUs. Furthermore, Nvidia corporation has created compute unified device architecture (CUDA), which is a GPU computing program standard based on the C programming language. Moreover, many companies have already developed a series of frameworks which support CUDA standard and hide CUDA implementation details, allowing users to focus on the design of ML algorithms. TensorFlow is almost the best one among those frameworks, which is developed by Google Brain (Google 2019).

Figure 2 is our training module development flow. Firstly we use Tensorflow to design our training algorithm, and then the Tensorlfow will translate the python code into CUDA code, and depart the code into two parts, and put one on the GPU to run, put the other on the CPU to execute. After finishing GPU computing, the CPU will collect the results from the GPU and combine the results.

3.3 The FPGA inferencing module

The goal of the inferencing phase is to do inferencing with minimum latency using the model generated from the training phase. Unlike training, we often do inferencing multiple times. Any small latency of each cycle can accumulate to a considerable amount. So it is valuable to reduce the inferencing latency.

FPGAs are composed of logical elements (LEs). We just programmed these LEs into different hardware electrical circuits to meet our requirements. FPGAs' performances are near application-specific integrated circuits (ASICs), and they can be reconfigured many times, so their prices are usually much higher than ASICs, CPUs, and GPUs. In most FPGA designs, there is no fetching and decoding instruction system, which make FPGAs are much faster than GPUs and CPUs. However, this advantage also becomes the FPGAs'disadvantage. As the FPGAs do not have traditional instruction system, when we have a new function requirement, we can not use the general software programming

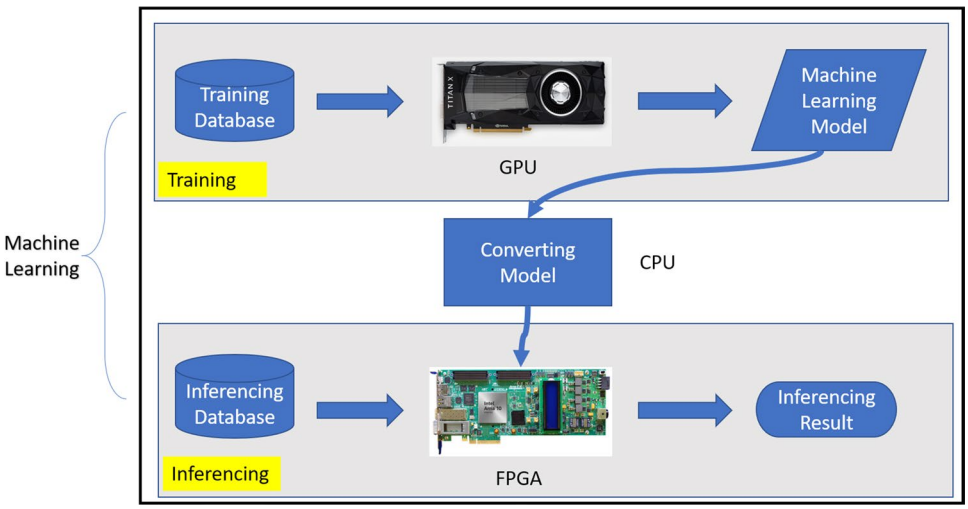


Fig. 1 The architecture of heterogeneous machine learning system (Liu et al. 2018)

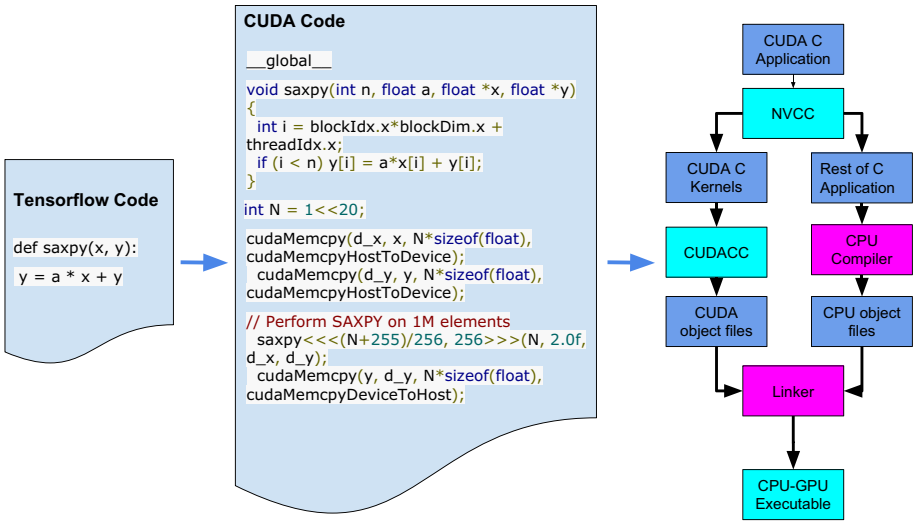


Fig. 2 The flow of the GPU training development (Lanfear 2013)

language such as C to describe it. We have to use some HDLs such as Verilog or VHDL (VHSIC very high-speed hardware description language) to re-design the whole hardware electrical circuit. The HDLs are similar to the assembly languages, and even a small function needs many HDL sentences to describe. If a system is as complex as the DNN's training module, it is hard to use HDLs to implement it, which restricts the scales of FPGA designs most.

Now, we can compensate the difficulties of FPGA design by using high-level programming languages such as open computing language (OpenCL) which do not require too much electrical circuits knowledge (Aydonat et al. 2017; Zhao et al. 2016; Bettoni et al. 2017; Li et al. 2018). OpenCL is a framework for developing programs which can run on different heterogeneous platforms such as GPU + CPU or FPGA + CPU.

Figure 3 shows its development flow. We can divide the development into two phases: the host program and the kernels. When designing the kernels, we should consider how to take the full parallel computing capacity of the FPGA. The kernels will be synthesized into hardware logic circuits and uploaded to the FPGA development board. The host program, which is similar to the traditional C program that runs on the CPU, is in charge of allocating parallel computing jobs on the FPGA development board and collecting the computing results from the FPGA.

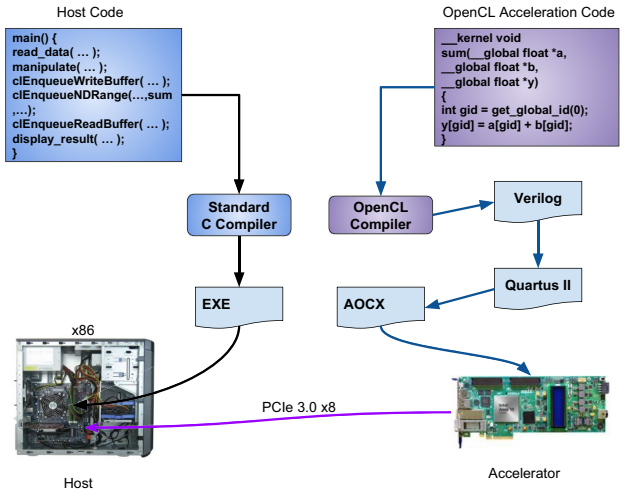


Fig. 3 The flow of the FPGA inferencing development (Intel 2018)

3.4 Model converter

Training and inferencing are usually put on different platforms, as the training process is high-density computing and time-consuming, it needs a large number of computing resources which is not affordable for most inferencing platforms. Moreover, the training is usually a one-off process in a fixed time. On the contrary, the inferencing happened often. So it is a smart and efficient way to use different platforms to implement the training and inferencing phases.

However, generally, different platforms adopt different frameworks and model definitions. We can not use the model generated by one framework on another directly. A model converter which can convert a model from one framework to another is needed. Although there are already some tools such as MMdnn (Chen et al. 2019) for converting the model file among the main frameworks (Tensorflow, Caffee, PyTorch and so on), it is useless for personal custom-made cases, including our case.

Suppose we should turn a model from the source framework to the target framework. For converting a model, here is a general process.

1. Understand and capture the date structure of the source model completely.
2. Understand and capture the model file definition of the target framework fully.
3. Design a mapping function from the source model parameters and weights to the target model.

A model data structure or a model file of machine learning contains parameters such as weights and biases generated during the training process, it is the core and goal of machine learning, and we save it after training and load it before inferencing.

It is not accessible to understand a model data structure or file well, as there are many items, such as the number of The results of CNN training timelayers, the size of each layer, the size of each filter, the activation functions used between layers, the order of matrix dimensions, and the flattening ways. In particular, if the model definition is not open source, we have to guess the order of matrix dimensions, which is almost the hardest part, as the matrices involved usually have 4-dimensions, leading to $4 \times 3 \times 2 \times 1 = 24$ possible flattening ways. However, only one way is correct.

When we design the model mapping function, we should obey all of the above definitions in the target model, which is the secret to guarantee the correctness of model converting.

Figure 4 is a simple model converting example. There are two frameworks, 1 and 2, which have the same machine learning architectures and the numbers of parameters. However, their model's data structures and flattening ways are different. W_1 in model 1 is stored as $\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$ and flattened as [1, 1, 0, 0] while W_2 in model 2 is stored as $\begin{bmatrix} 0.1 & 0.1 \\ 0 & 0 \end{bmatrix}$ and is flattened as [0.1, 0, 0.1, 0]. After understanding the two model's data structures and flattening ways well, it is easy to convert the model from framework 1 to framework 2.

In our case, as we implemented the training module on the Tensorflow framework, and implemented the inferencing module on the OpenCL-FPGA framework, we design a particular model mapping function which can map weights and bias from the Tensorflow framework to the FPGA framework.

3.5 Summary

After analyzing and comparing different hardware device combinations, we found the best solution to implement a machine learning system that is to use GPU to implement the training module and the FPGA to implement the inferencing module and add a model converter in charge of converting the model from one framework to another.

To verify this design methodology, we implemented two ML use cases and tested their performance. The first is a CNN for digital hand-writing recognition, the other is a DNN regression for estimating the PUE of the data center.

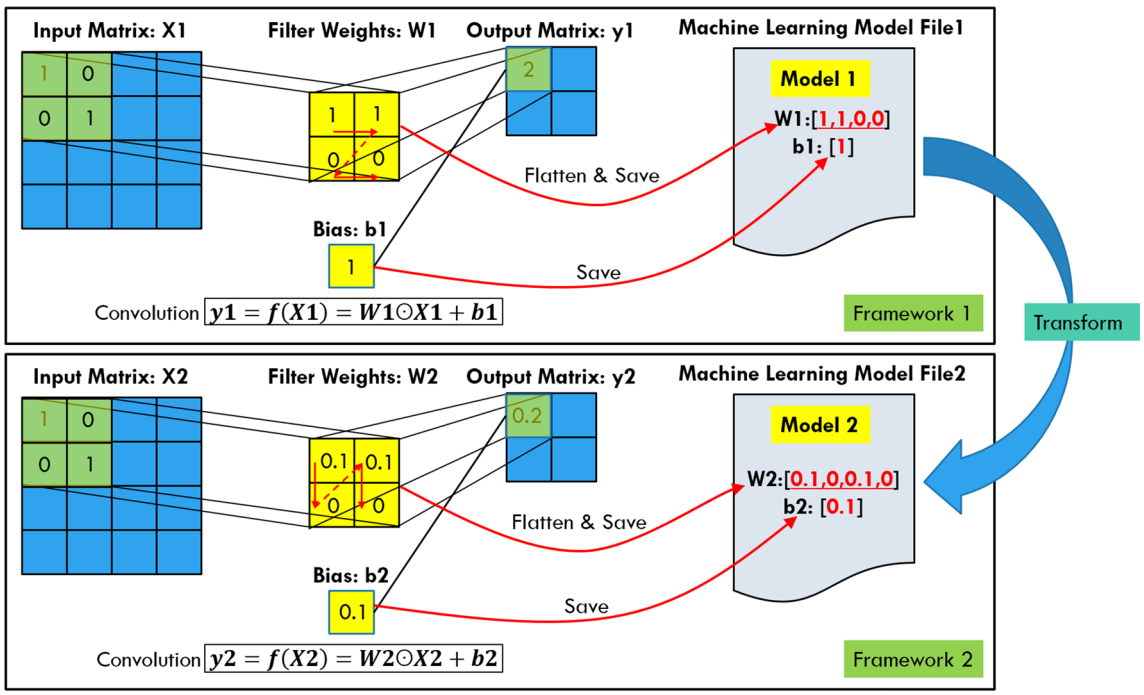


Fig. 4 Schematic of model converting (Liu et al. 2018)

Table 1 Hardware devices list (Liu et al. 2018)

Device	Type	Number
CPU	Intel(R) Xeon(R) CPU E5-1620 v4 @ 3.50 GHz	1
Memory	Samsung DDR4 8 g	4
Solid state disk drive	INTEL SSDSC2BB48 480 g	1
Mechanical hard disk drive	WDC WD40EFRX-68N 4 TB	1
GPU	NVIDIA TITAN Xp	1
FPGA	Intel Arria 10 GX FPGA development kit	1

Table 2 Software environment (Liu et al. 2018)

Software	Version
Ubuntu	16.04.3 LTS
Python	3.5.2
Tensorflow	1.4.0
Tensorflow-GPU	1.4.0
CUDA	8.0
Intel(R) FPGA SDK for OpenCL	17.1.0 Build 240

4 A case study on digital handwriting recognition with CNN classification

4.1 Experiment environment

The following two cases use the same experiment environment.

4.1.1 Hardware environment

Our devices' list is shown in Table 1. We use the motherboard to host every hardware device together. The Titan XP graphics card communicates with the host board through PCIe gen 3 × 16 (with the bandwidth of 15760 MB/s). The Arria 10 development board transfers data to the host board by PCIe gen 3 × 8 (with the bandwidth of 7880 MB/s). When the amount of training data is huge, GPU cannot load the whole data in one-time and has to break the data into many small batches, which will lead to transferring data frequently between the host and the device.

On the contrary, the data for inferencing is usually small, that is no need to transfer data frequently. From this perspective, the bandwidth becomes a performance bottleneck of the system. Therefore, this becomes another reason to use the GPU to do the training and the FPGA to do the inferencing.

4.1.2 Software environment

Table 2 is our software environment. The operating system is Ubuntu 16.04. We use the Tensorflow to implement the GPU training module and the OpenCL to design the FPGA inferencing module.

4.2 The algorithm of CNN case

Our goal is to verify the performance of the heterogeneous machine learning system. We select LeNet-5 and MNIST (LeCun et al. 2018) as our algorithm and data set, which are not too complicated but enough to show the effects of hardware acceleration.

The MNIST is a training dataset for digital handwriting recognition. Each example is a pixel value matrix whose size is 28 × 28, and each pixel value's range is from 0 to 255. In our case, for easy processing, we divide the pixel value by 255 (Fig. 5). MNIST dataset totally includes 55,000 training examples, 5000 validation examples, and 10,000 test examples.

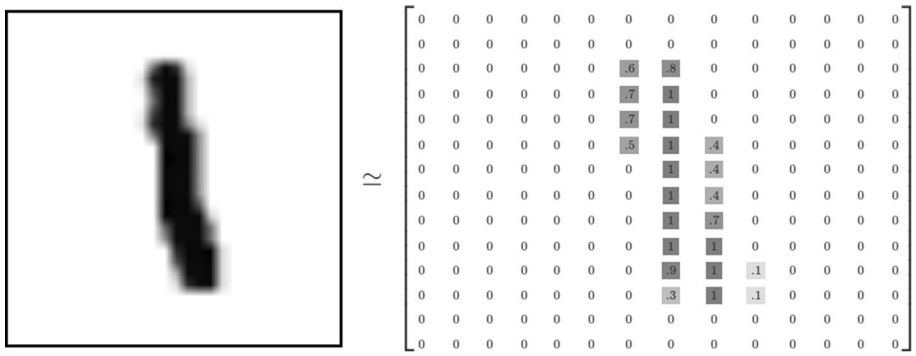


Fig. 5 The pixel value matrix (Google 2018)

The LeNet-5 is designed by Yann LeCun for handwritten and machine-printed character recognition (LeCun et al. 1998), belongs to CNNs. It has almost all classical CNN structures (convolution layers, pooling layers, and full connection layers, and so on) and has high accuracy in digital handwriting recognition. Most importantly, it contains a lot of high-density computing jobs which are suitable for testing the acceleration effects of parallel designs.

Figure 6 is our tailor-made version of LeNet-5 for this experiment. It has seven layers in total, including three convolution layers, two sub-sampling layers, and two full connection layers. These layers are orderly arranged, as illustrated in Fig. 6.

4.3 The training module of CNN case

4.3.1 The implementation

Figure 7a is our training phase control flow diagram. It includes two main phases, i.e., forward propagation and backward propagation. The difference of value calculated by the forward propagation and label value is passed into the backward propagation to calculate the weights and biases of each layer. After several computing loops, the difference converges within a permissible range, the training process terminates. We store weights and bias in a model file.

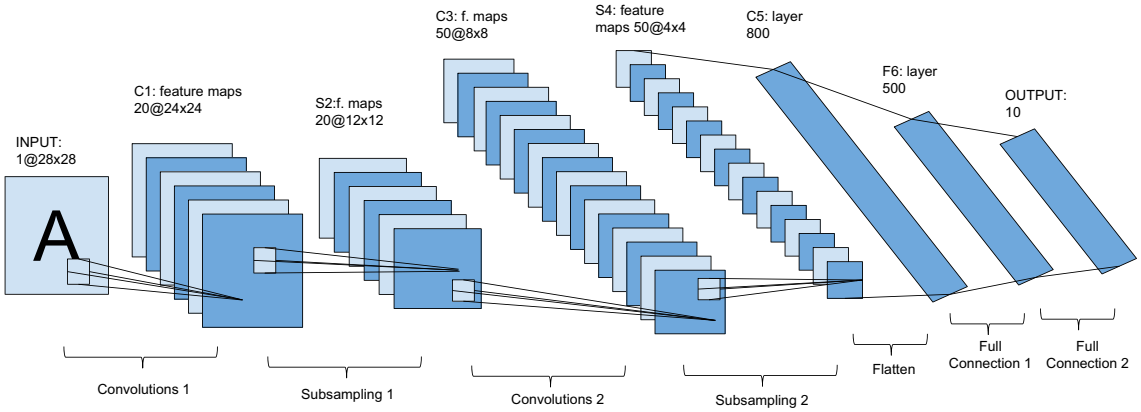


Fig. 6 The architecture of LeNet-5 improved for experiment (Liu et al. 2018)

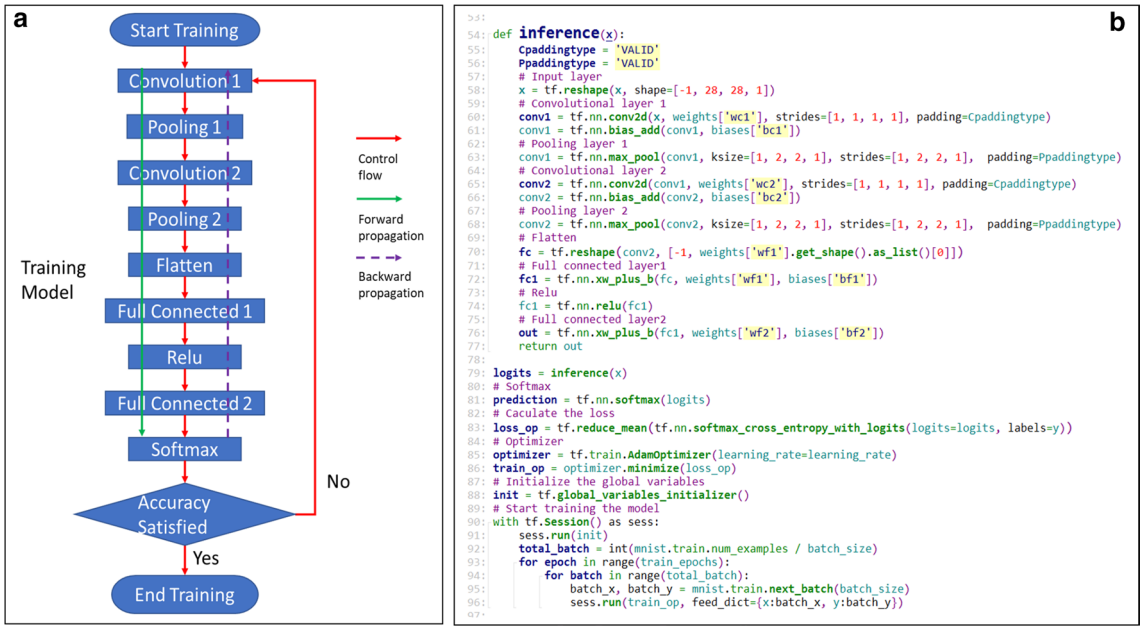


Fig. 7 **a** The control flow of training; **b** the TensorFlow implementation of the training module (Liu et al. 2018)

We use NVIDIA Titan Xp graphics card to accelerate the training process. The GPU has 3840 CUDA cores which can be programmed to do parallel computing. Its floating computing performance can reach 12 TFLOPS.

To do the NVIDIA graphics card computing development, it should use the CUDA programming language mentioned previously. That will mean we should build the acceleration kernel from scratch. Thanks to the TensorFlow, which packages the CUDA libraries, now we only need to focus on designing the architecture. The Fig. 7b is part of our TensorFlow implementation code.

4.3.2 The experiment results

In this experiment, we aim to find a better device for training model by comparing the training speed of CPU and GPU.

For achieving this goal, we designed two use cases. One only has a CPU-E5-1620, the other has a CPU-E5-1620 and a GPU-TitanXp. We chose the same 55,000 training examples and measured their training time, respectively. We did six times the same tests and calculated their average times.

The results are in Table 3. We can conclude that the average speed of TitanXp is about 8.8× faster than the average speed of CPU E5-1620 with the same accuracy. This result is similar to Kind (2018) work whose GPU speed is about 9× faster than CPU.

4.4 The inferencing module of CNN case

4.4.1 The implementation

Figure 8a is our inferencing phases control flow. It only has forward propagation. It loads the model generated from the training module and then does inferencing only with the forward propagation algorithm.

Table 3 The results of CNN training time (Liu et al. 2018)

Experiments	CPU-E5-1620		GPU-TitanXp		Accelerate times
	Training time (s)	Accuracy (%)	Training time (s)	Accuracy (%)	Acceleration (GPU/CPU)
1	448.60	98.70	50.61	98.80	8.86
2	447.79	98.88	51.17	98.58	8.75
3	448.35	98.90	50.73	98.80	8.84
4	448.62	98.94	50.46	98.88	8.89
5	447.62	98.59	50.94	98.82	8.79
6	447.88	98.94	50.78	98.79	8.82
Average	448.10	98.80	50.80	98.80	8.80

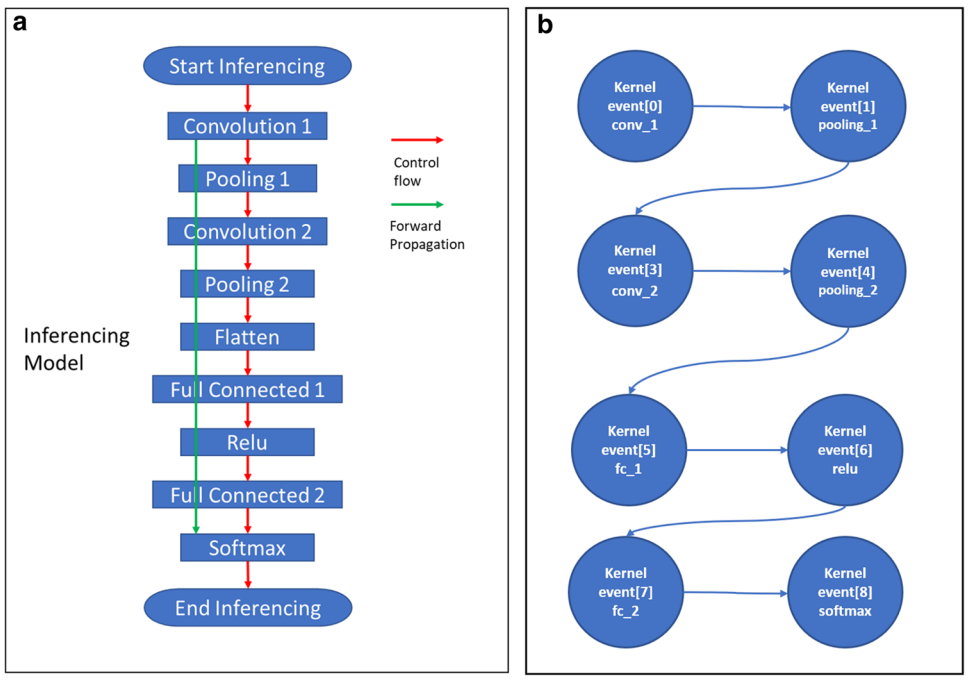


Fig. 8 **a** The control flow of inferencing; **b** the kernels of OpenCL implementation (Liu et al. 2018)

We use Intel Arria 10 FPGA development board to implement the inferencing module. Arria 10 is made by Intel with the 20 nm process. It has high performance and low power consumption. Although its 1.5 TFLOPS is still slower than TITAN Xp 12 TFLOPS, its power consumption below 100 W is much better than TITAN Xp 250 W.

We use OpenCL for FPGA development. Its development includes two parts: a host program and kernels. The host program runs on the CPU, and the kernels run on the FPGA. Figure 8b shows our kernel events implemented with OpenCL. It has eight kernel events which map with the control flow's eight functions. These logical kernel events will be programmed on the FPGA and executed one by one to implement the inferencing function.

4.4.2 The experiment results

In this experiment, we use the CPU, GPU, and FPGA to execute the same inferencing algorithm with the same model and compare their efficiencies with each other.

First, we measure the time of inferencing 10,000 images on the CPU, GPU, and FPGA devices, respectively. Then we execute the measurement six times and calculate the average time of inferencing 10,000 images on different hardware devices.

The results of the experiments are presented in Table 4. From the table, we can see that the average speed of Arria 10 is about 10.9 times faster than the average rate of CPU E5-1620 and is about 7.1 times faster than the GPU TitanXp.

4.5 The model converter of CNN case

4.5.1 The experiment results

For verifying our model converter works well, we conduct two experiments on the same 10K MNIST test examples respectively. In Experiment_1, we only test FPGA inferencing accuracy with the original model while in Experiment_2, and we change CNN's configuration, retrain the model by TensorFlow with GPU, convert the model to the FPGA and then test the inferencing accuracy on FPGA and Tensorflow respectively.

Table 5 presents the statistics of accuracy collected from the two experiments. The Experiment_2's accuracy is better than Experiment_1's. In Experiment_2, the FPGA has preserved the same precision as the TensorFlow, which proved our model convert successfully.

5 A case study on data center PUE with DNN regression

5.1 The overview of the use case

To fully evaluate our hybrid design methodology of the ML system, we select another algorithm to implement and test its performance on CPU, GPU, and FPGA. This algorithm belongs to one of DNN regressions, and we use it for estimating the PUE of Data Center (DC).

A DC is a physical space that groups together IT systems (servers, storage, and so on), mechanical systems [computer room air conditioner (CRACs), chillers, and so on], and electrical systems [uninterruptible power supply (UPS), power distribution unit (PDU), transformers, and so on], for storing, processing and protecting data. The energy consumption takes the main part of operating a DC and can reach up to 75% of operating costs, which is one of the major reasons why industrial and environmental organizations have focused on improving energy performance while ensuring continuity of services. For this reason, many energy-related metrics are defined, such as PUE. The Eq. (1) is the definition of the PUE (David Wright

Table 5 Converting model experiments of CNN

Experiment_1	Accuracy(%)	Experiment_2	Accuracy(%)
N/A	N/A	TensorFlow	99.13
FPGA	99.05	FPGA	99.13

Table 4 Results of 10,000 image inferencing time

Experiments	CPU-E5-1620	GPU-TitanXp		FPGA-Arria10		
	Inferencing time (us)	Inferencing time (us)	Acceleration (GPU/CPU)	Inferencing time (us)	Acceleration (FPGA/CPU)	Acceleration (FPGA/GPU)
1	14.881587	9.713557	1.5320	1.38002	10.8	7.0
2	14.995880	9.202130	1.6296	1.34051	11.2	6.9
3	14.685811	9.419498	1.5591	1.36456	10.8	6.9
4	14.785795	9.762074	1.5146	1.34304	11.0	7.3
5	15.064704	9.854970	1.5286	1.32339	11.4	7.4
6	14.307688	9.440861	1.5155	1.37961	10.4	6.8
Average	14.786910	9.565520	1.5459	1.35520	10.9	7.1

2017). The IT equipment power includes all the actual load of IT equipment such as workstations, servers, storage, switches, printers, and other service delivery equipment (David Wright 2017):

$$PUE = \frac{Total\ data\ center\ power}{IT\ equipment\ power}. \quad (1)$$

The PUE stands for how energy is efficiently used to keep the DCs running without service interruption. It is used to evaluate over a year the total amount of energy consumed by the DC, compared to the amount of energy necessary for the operation of the IT equipment. The closer the result is to 1.0, the less power the non-IT equipment consumes, and the more it is considered "eco-responsible".

Moreover, the interactions of DC systems are complicated. According to Ounifi et al. (2018)'s work, the DC systems (IT, electrical and mechanical systems) interactions and the different feedback loops make it difficult to estimate and predict the DCs' energy efficiency accurately.

To capture such complexities, we try to find an estimation model to calculate the PUE metric values with the help of DNN.

DNN borrowed the concept of the deep neural network of the brain and will mainly analyze and process the input data through a succession of several neurons that take the input signals from the previous neurons. DNNs are good at modeling non-linearity and have characteristics such as the ability to model real-time operation and fault tolerance.

In this case study, we will try to use our machine learning design methodologies to implement a DNN regression system which can train a model for estimating the PUE of a data center.

5.2 The dataset and the DNN regression algorithm of DNN case

5.2.1 The dataset

The dataset we used is from the "ITEA3 RISE SICS Data Center" located in Sweden. It has 2881 sets. Each set is collected from the DC every 60 s from 9 a.m. to 9 p.m. containing 415 kinds of DC features such as fan speed, input DC's power, and average cold Aisle temperature (Ounifi et al. 2018). Table 6 shows part of the 415 features. Besides, the dataset also has a time series column and a ground truth PUE column. So our working dataset is a 2881×417 matrix. We apply cross-validation by dividing the 2881 data sets into two parts: 2656 for training sets, and 225 for test sets.

Table 6 A part of selected DC features of the ITEA3 RISE Sics DC

DC features	Units
Indoor/outdoor temperature	°C
Input data center power	Mw
Whole data center humidity	%
Energy consumption/rack	Mw
Workload (electrical)/server	Mw
Workload (CPU usage)/server	Mgbit
Power consumption after the PDU	Mw
Average cold Aisle temperature	°C
Fan speed	RPM
Fan power	Kw
CRAC fan power	Kw
Power used by chilled liquid	Kw
Chilled water entering temperature	°C
CRAC energy consumption	KVA
Total rack IT load	KVA
Hot Aisle temperature	°C
hline Outside air dry bulb temperature	°C

5.2.2 The architecture of DNN regression algorithm

According to the structure of Sect. 5.2.1's data sets, we design a tailor-made version of DNN regression which is composed of one input layer, five hidden layers and one output layer. The details architecture of the five hidden layers is as follows.

- Input layer: 415 neurons.
- Full connected layer 1: 512 neurons.
- Activation function: Relu.
- Full connected layer 2: 1024 neurons.
- Activation function: Relu.
- Full connected layer 3: 1024 neurons.
- Activation function: Relu.
- Full connected layer 4: 1024 neurons.
- Activation function: Relu.
- Full connected layer 5: 512 neurons.
- Activation function: Relu.
- Output layer: 1 neuron.

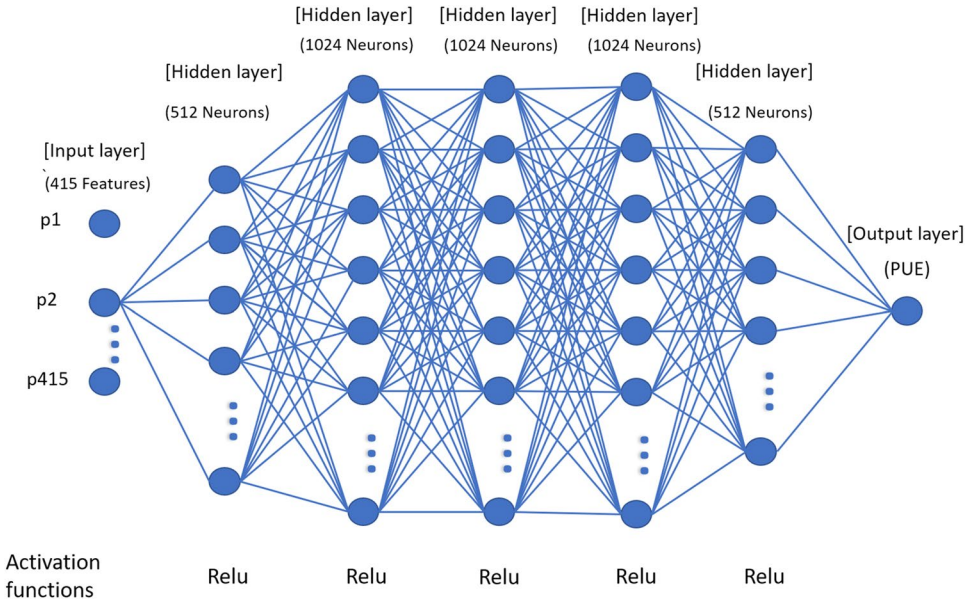
And all these have been shown on the Fig. 9.

5.3 The training module of DNN case

5.3.1 The implementation

Figure 10a is our training control flow which is similar to Fig. 7a. It includes two main phases, i.e., forward propagation and backward propagation. The difference between CNN case and DNN case is the DNN case has no convolutional layers,

Fig. 9 DNN proposed model



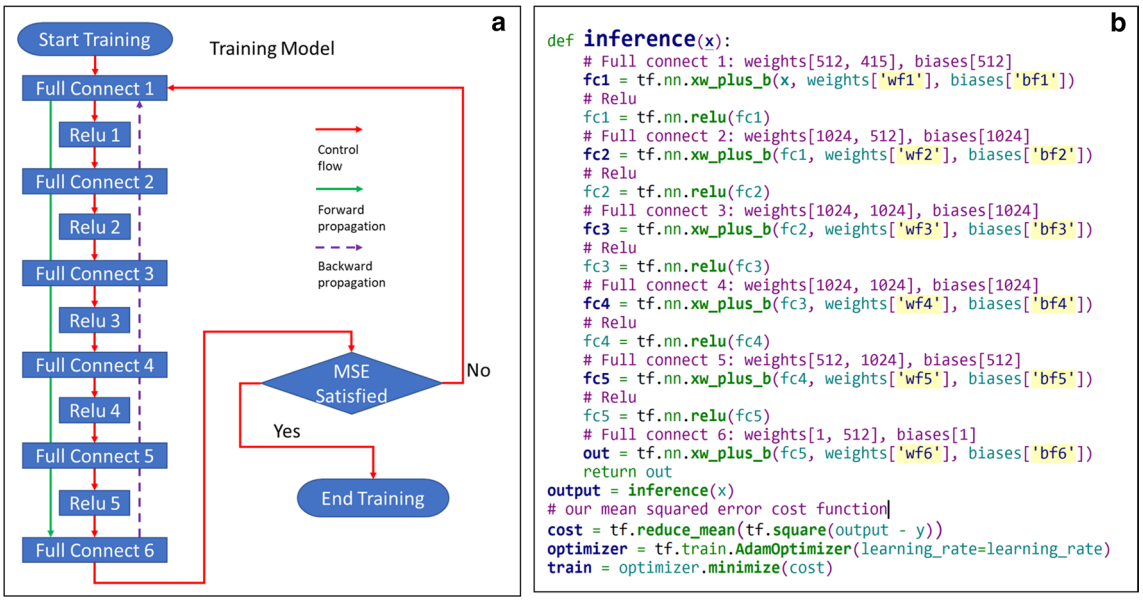


Fig. 10 **a** The control flow of DNN training; **b** the TensorFlow implementation of the DNN training module

and pooling layers and the cost function is mean square error (MSE) which defined in Eq. (2) instead of mean cross entropy (MCE). After several computing loops, the MSE will converge within a permissible range, the training process terminates. We will store the final weights and bias in a particular model file:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i')^2, \tag{2}$$

where y_i is the real output, y_i' is the calculated output, and n is the number of the input data. We still use Tensorflow to implement this training part. Figure 7b is part of our TensorFlow code.

5.3.2 The experiment results

During this experiment, we run the same DNN regression training program with the same training dataset on GPU-TitanXp and CPU-E5-1620 respectively and then measure their MSEs and execution time.

Figure 11 is the result that we use the model trained by GPU and the model trained by the CPU to do PUE

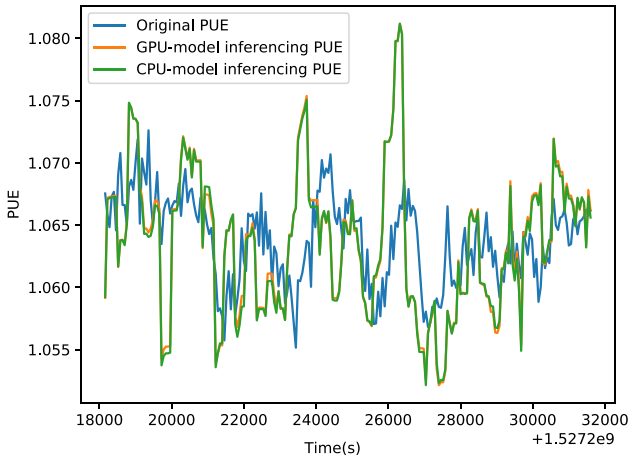


Fig. 11 The comparison of GPU-model and CPU-model inferencing results

inferencing on the same test set. They look almost the same. They have the same MSE: 0.000029. However, their execution times are different. Table 7 shows that the GPU's training speed is about 12.4× faster than the CPU's.

5.4 The inferencing module of DNN case

5.4.1 The implementation

Figure 12a is our inferencing phases control flow, which only has two operations, full connecting and Relu. It loads the model generated from the training phase and then does inferencing only with the forward propagation algorithm.

We also use OpenCL for FPGA design. Like GPUs, the process of FPGA development has two parts: a host program and kernels. The host program which runs on the CPU is in charge of loading datasets and model, allocating computing jobs on the FPGA and collecting the results from the FPGA. The kernels run on the FPGA are the primary computing acceleration part.

Table 7 The DNN training time

Experiments	CPU-E5-1620		GPU-TitanXp		Accelerate times
	Training time (s)	MSE (%)	Training time (s)	MSE (%)	Acceleration (GPU/CPU)
1	154.36	0.000029	12.14	0.000029	12.72
2	154.58	0.000029	12.38	0.000028	12.48
3	154.66	0.000029	12.52	0.000029	12.35
4	154.74	0.000029	12.40	0.000029	12.48
5	154.49	0.000029	12.68	0.000029	12.12
6	154.78	0.000029	12.73	0.000029	12.16
Average	154.60	0.000029	12.50	0.000029	12.40

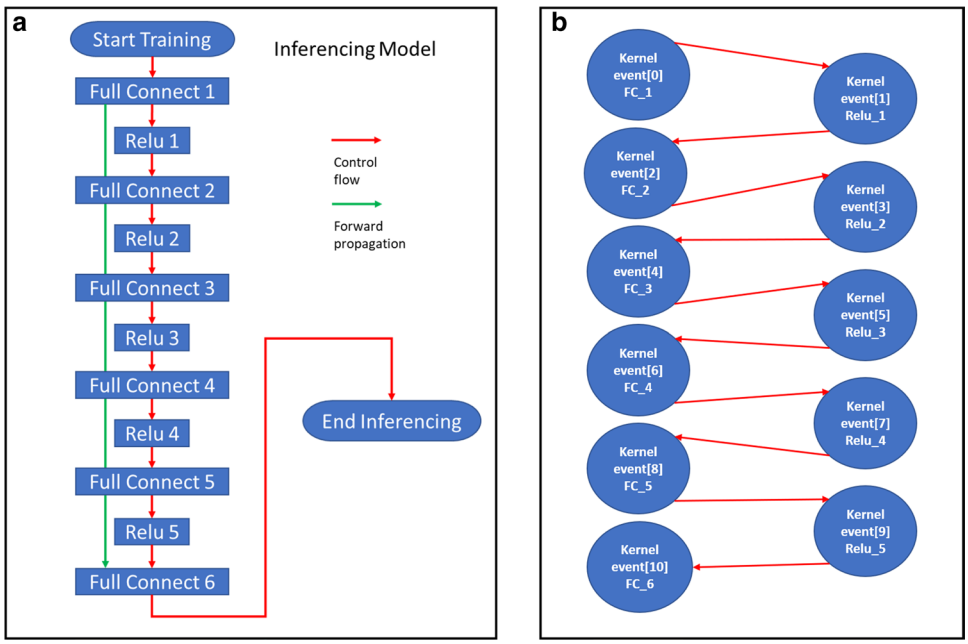


Fig. 12 **a** The control flow of DNN inferencing; **b** the kernels of OpenCL DNN implementation

Table 8 Results of 225 sets inferencing time

Experiments	CPU-E5-1620	GPU-TitanXp		FPGA-Arria10		
	Inferencing time (s)	Inferencing time (s)	Acceleration (GPU/CPU)	Inferencing time (s)	Acceleration (FPGA/CPU)	Acceleration (FPGA/GPU)
1	0.539186	0.160862	3.3519	0.0363316	14.8	4.4
2	0.488358	0.133327	3.6629	0.0374917	13.0	3.6
3	0.508489	0.119494	4.2554	0.0346079	14.7	3.5
4	0.500365	0.122232	4.0936	0.0382779	13.1	3.2
5	0.520802	0.129460	4.0229	0.0384238	13.6	3.4
6	0.451343	0.153649	2.9375	0.0362708	12.4	4.2
Average	0.501420	0.136500	3.6733	0.0369006	13.6	3.7

Figure 12b shows our kernel events implemented with OpenCL. It has 11 kernel events which map with the control flow's 11 functions. These logical kernel events will be programmed on the FPGA and executed one by one to implement the inferencing function.

5.4.2 The experiment results

Similar to the previous use case, we use the CPU, GPU, and FPGA to execute the same inferencing algorithm with the same model and compare their inferencing times with each other. First, we measure the time of inferencing 225 sets on the CPU, GPU, and FPGA devices, respectively. Then we execute the measurement six times and calculate the average time of inferencing 225 sets on different hardware devices.

The results of the experiments are presented in Table 8. From the table, we can see that the average speed of FPGA Arria-10 is about 13.6× times faster than the CPU E5-1620 and is about 3.7× times faster than the GPU TitanXp.

5.5 The model converter of DNN case

5.5.1 The experiment results

Since the framework of Tensorflow is different from the framework of our FPGA, we need to undertake the model converting. To verify that our model converter works correctly, we test the MSE of Tensorflow and the MSE of FPGA on the same 225 test data sets using the model generated by the Tensorflow and the model converted by our platform, respectively. Table 9 are the results. We have done six times training on the Tensorflow framework and got six models, converted these models for the FPGA, and with these converted models we do inferencing on the FPGA, and all got the same MSE.

Figure 13 is one of our test case results. From this figure, we can see that the inferencing results of Tensorflow are almost the same as the results of FPGA, which proves our model converter working well.

Table 9 Converting model experiments of DNN

Experiments	MSE of tensorlfow	MSE of FPGA
1	0.000028	0.000028
2	0.000029	0.000029
3	0.000028	0.000028
4	0.000029	0.000029
5	0.000029	0.000029
6	0.000029	0.000029

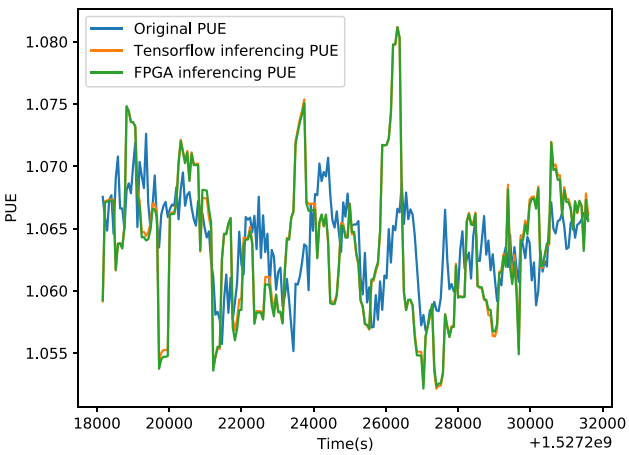


Fig. 13 Results of converting model in inferencing PUE

6 Discussions

We intentionally selected two different area use cases to verify that our hybrid ML platform design methodology is general. Although the two algorithms are different, one is CNN, the other is normal DNN, with our solution we got the same conclusion, that is the GPU is more suitable for training, and the FPGA is best in referencing, which confirms our initial hypothesis and analysis. Moreover, it proves our solution can take advantage of

Table 10 Results of one image inferencing time

Experiments	CPU-E5-1620	GPU-TitanXp		FPGA-Arria10		
	Inferencing time (us)	Inferencing time (us)	Acceleration (GPU/CPU)	Inferencing time (us)	Acceleration (FPGA/CPU)	Acceleration (FPGA/GPU)
1	3172	616045	0.0051	88.74	35.7	6942.4
2	5564	589114	0.0094	90.12	61.7	6536.7
3	4620	588444	0.0079	94.83	48.7	6205.3
4	3234	598652	0.0054	84.57	38.2	7079.0
5	4037	600288	0.0067	101.08	39.9	5938.8
6	4579	609913	0.0075	108.74	42.1	5609.0
Average	4201	600409	0.0070	94.70	44.4	6341.5

different high-performance devices to implement all kinds of machine learning jobs efficiently.

Also, we have a few interesting findings. For instance, when we perform the CNN case study inferencing on one image, we obtain the result shown in Table 10. We can see that the average speed of Arria 10 is about 44.4 times faster than the CPU E5-1620 and is about 6342 times faster than the GPU TitanXp. In Wang et al. work (2017), FPGA was 36.1× faster than CPU. These results justify that our decision to use the FPGA to do inferencing is correct. Concerning the GPU's slight underperformance, we explain that the workload associated with the inferencing of one image is too small compared with the GPU initialization time and delay. Therefore, the total time of the GPU, which includes the initialization time and inferencing time is the longest.

Additionally, in the DNN case study, when the training sets' batch size is small, e.g., less than 32, the training speed of GPU is even slower than the training speed of CPU. Our explanation is CPU has better bandwidth and frequency than GPU, as the transfer speed of our ddr4-2400 is 19,200 MB/s, the transfer speed of GPU which equals the PCIe Gen 3 × 16 is 15,760 MB/s, and the max frequency of our CPU is 3800 MHz, the max frequency of our GPU is 1582 MHz. When the computing job is too small, although the GPU has more parallel computing cores, GPU does not have any further advantages over CPU.

7 Conclusion and future work

In this paper, we presented a hybrid, GPU-FPGA based design methodology for enhancing machine learning applications' performance. After carefully comparing and analyzing the characters and the structures of CPU, GPU and FPGA, the results of our investigations suggest that the GPU is more suitable for training while the FPGA is best for inferencing and a model converter is necessary when the training and inferencing frameworks are different. Therefore, to achieve higher machine learning performance, a better strategy would be to implement the training module on the GPU and the inferencing module on the FPGA.

According to the above design methodology, we implemented two machine learning systems. One is a CNN for handwriting digit recognition, and the other is a DNN regression for the estimation of the data center's PUE. The results of the two use cases confirm clearly that our hypothesis and analysis are correct. Also, it proves that our ML platform solution can take advantage of different high-performance devices to implement all kinds of machine learning jobs efficiently.

In our future work, we plan to do further investigation on the power analysis of the hybrid ML system. Besides, we will summarize the experience of model converting to identify standard rules for any model converting.

Acknowledgements This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Research Canada and the Canada Research Chair in Sustainable Smart Eco-Cloud. We would also like to thank Yves Lemieux for his insightful feedback during the research work.

References

Aydonat U, O'Connell S, Capalija D, Ling AC, Chiu GR (2017) An OpenCL deep learning accelerator on Arria 10. In: Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays, ACM, pp 55–64

Bauer S, Köhler S, Doll K, Brunsmann U (2010) FPGA-GPU architecture for Kernel SVM pedestrian detection. In: Proceedings of the 2010 IEEE computer society conference on computer vision and pattern recognition workshops (CVPRW), IEEE, pp 61–68

Bergstra J, Bastien F, Breuleux O, Lamblin P, Pascanu R, Delalleau O et al (2011) Theano: deep learning on gpus with python. In: NIPS 2011, BigLearning Workshop, Granada, Spain, Citeseer, vol 3

Bettoni M, Urgese G, Kobayashi Y, Macii E, Acquaviva A (2017) A convolutional neural network fully implemented on FPGA for embedded platforms. In: New generation of CAS (NGCAS), IEEE, pp 49–52

Chen C, Yao J, Zhang R, Zhou Y, Qin T, Zhan T, Wang Q (2019) MMdnn. GitHub repository. https://github.com/microsoft/MMdnn

David Wright (2017) Improving electrical efficiency in your data center. https://www.datacenterknowledge.com/archives/2014/09/23/improving-electrical-efficiency-data-center

Ganesh SS, Arulmozhivarman P, Tatavarti VSNR (2018) Prediction of pm2.5 using an ensemble of artificial neural networks and regression models. J Ambient Intell Hum Comput. https://doi.org/10.1007/s12652-018-0801-8

Google (2018) The MNIST matrix. https://www.tensorflow.org/versions/r1.1/get_started/mnist/beginners

Google (2019) TensorFlow. https://www.tensorflow.org/

Huang R, Feng W, Fan M, Guo Q, Sun J (2017) Learning multi-path cnn for mural deterioration detection. J Ambient Intell Hum Comput. https://doi.org/10.1007/s12652-017-0656-4

Intel (2018) Intel OpenCL development. http://www.innovatefpga.com/cgi-bin/innovate/teams.pl?Id=PR029&All=1

Kind T (2018) Tensorflow (TF) benchmarks. https://github.com/tobigithub/tensorflow-deep-learning/wiki/tf-benchmarks

Lanfear T (2013) High performancecomputing with CUDA and Tesla hardware. https://intranet.birmingham.ac.uk/it/teams/infrastructure/research/bear/documents/public/CUDA-2013-07-31/CUDA-Tutorial.pdf

LeCun Y, Bottou L, Bengio Y, Haffner P (1998) Gradient-based learning applied to document recognition. Proc IEEE 86(11):2278–2324

LeCun Y, Cortes C, Burges CJC (2018) The MNIST database. http://yann.lecun.com/exdb/mnist/

Li Y, Liu Z, Xu K, Yu H, Ren F (2018) A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. J Emerg Technol Comput Syst 14(2):18:1–18:16. https://doi.org/10.1145/3154839

Liu X, Ounifi HA, Gherbi A, Lemieux Y, Li W (2018) A hybrid gpu-FPGA-based computing platform for machine learning. Proc Comput Sci 141:104–111

Motamedi M, Gysel P, Akella V, Ghiasi S (2016) Design space exploration of FPGA-based deep convolutional neural networks. In: Proceedings of the 21st Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, pp 575–580

Nagarajan K, Holland B, George AD, Slatton KC, Lam H (2011) Accelerating machine-learning algorithms on FPGAs using pattern-based decomposition. J Signal Process Syst 62(1):43–63

Ounifi HA, Liu X, Gherbi A, Lemieux Y, Li W (2018) Model-based approach to data center design and power usage effectiveness assessment. Proc Comput Sci 141:143–150

Potluri S, Fasih A, Vutukuru LK, Al Machot F, Kyamakya K (2011) CNN based high performance computing for real time image processing on GPU. In: 2011 joint 3rd Int'l workshop on nonlinear dynamics and synchronization (INDS) and 16th Int'l symposium on theoretical electrical engineering (ISTET), IEEE, pp 1–7

Qiu J, Wang J, Yao S, Guo K, Li B, Zhou E, Yu J, Tang T, Xu N, Song S et al (2016) Going deeper with embedded FPGA platform for convolutional neural network. In: Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays, ACM, pp 26–35

Raina R, Madhavan A, Ng AY (2009) Large-scale deep unsupervised learning using graphics processors. In: Proceedings of the 26th annual international conference on machine learning, ACM, pp 873–880

Rush A, Sirasao A, Ignatowski M (2017) Unified deep learning with cpu gpu and fpga technologies. In: Advanced Micro Devices, Tech. Rep

Sharp T (2008) Implementing decision trees and forests on a GPU. In: European conference on computer vision. Springer, Berlin, Heidelberg, pp 595–608

Steinkraus D, Buck I, Y Simard P (2005) Using GPUs for machine learning algorithms. In: Proceedings of the 8th international conference on document analysis and recognition, IEEE, pp 1115–1120

Wang C, Gong L, Yu Q, Li X, Xie Y, Zhou X (2017) Dlau: a scalable deep learning accelerator unit on FPGA. IEEE Trans Comput Aided Design Integr Circ Syst 36(3):513–517

Zhao W, Fu H, Luk W, Yu T, Wang S, Feng B, Ma Y, Yang G (2016) F-CNN: an FPGA-based framework for training convolutional neural networks. In: Proceedings of the IEEE international conference on application-specific systems, architectures and processors, pp 107–114

Zhu M, Liu L, Wang C, Xie Y (2016) Cnnlab: a novel parallel framework for neural networks using gpu and FPGA—a practical study with trade-off analysis. CoRR arXiv:1606.06234