Chapter 3
Least-squares-solver Based Machine Learning Accelerator for Real-time Data Analytics in Smart Buildings

Hantao Huang and Hao Yu

3.1 Introduction

Among various energy consumers, it is reported that over 70% electricity is consumed by more than 79 million residential buildings and 5 million commercial buildings in the USA [1]. There is an increasing need to develop cyber-physical energy management system (EMS) for modern buildings composed of both micro-grid and smart IoT hardware [2]. For smart energy management system, collecting information from IoT devices can help recognize energy consumption profile and perform accurately load forecasting with consideration of occupants behavior. As such, load balance can be achieved based on demand-response strategy for better energy efficiency [3].

One direct application of demand-response strategy in energy management system (EMS) is the real-time dynamic electricity price [4] based on the demand. An accurate load forecasting can help schedule the energy demand to reduce the electricity cost. However, energy data analytics for load forecasting is challenging since it is greatly affected by occupants behavior and environmental factors [5]. Occupants behavior is of random nature and very hard to predict [6]. Using real-time sensed data from occupation location, power meters and various sensors can capture occupants behavior for more accurate data analytics. However,uploading data to the cloud and processing backend take latency and edge device such as smart-gateway is computational resource limited. Therefore, a computationally efficient

H. Huang
Nanyang Technological University, 50 Nanyang Avenue, Block S3.2, Level B2, Singapore 639798, Singapore

H. Yu (✉)
School of Electrical and Electronic Engineering, Nanyang Technological University, 50 Nanyang Avenue, Block S3.2, Level B2, Singapore 639798, Singapore
e-mail: haoyu@ntu.edu.sg

A. Chattopadhyay et al. (eds.), *Emerging Technology and Architecture for Big-data Analytics*, DOI 10.1007/978-3-319-54840-1_3

data analytics (machine learning algorithm) is greatly needed for real-time smart building energy management system.

Machine learning algorithms can be broadly classified into: supervised learning, unsupervised learning, and reinforcement learning [7]. Supervised learning based neural network is widely applied for energy data analytics. Supervised learning will learn the connection between two subset of data, inputs and outputs, to build a model. Two central problems under supervised learning are classification and regression. Both problems share the same goal to build a mode to predict the output based on the input. However, the difference between two problems is the fact the dependent attribute (output) is categorical for classification and numerical for regressions [8].

In the smart building EMS, the pre-trained model from machine learning algorithms will be loaded in the embedded system to perform data analytics such as short-term load forecasting. However, previous works [3, 9, 10] have limitations in twofold. Firstly, since various factors affect load forecasting, the pre-trained model cannot be adjusted with the new arrival data. Moreover, traditional supporting vector machine and neural network based algorithms [3] consume large hardware resource to analyze energy data with poor efficiency and latency. Secondly, previous energy data analytics [9, 10] ignores the real-time occupant profile, whose distribution at different functionalized location (office, resting area, kitchen, etc.) can significantly affect the short-term energy load forecasting accuracy. As such, the energy management system of building towards comfort and energy-efficiency is still not optimized.

In this work, we present a fast machine learning accelerator for smart building data analytics. A computational efficient machine learning is developed using a regularized least-squares solver with incremental square-root-free Cholesky factorization. A scalable and parameterized hardware architecture is developed in a pipeline and parallel fashion for both regularized least-squares and matrix-vector multiplication. With the high utilization of the FPGA hardware resource, our implementation has 128-PE in parallel operated at 50-MHz. Experimental results have shown that the proposed machine-learning accelerator (on FPGA) has good forecasting accuracy with an average speed-up of 4.56× and 89.05×, when compared to general CPU and embedded CPU. Moreover , 450.2×, 261.9× and 98.92× energy saving can be achieved comparing to general CPU, embedded CPU and GPU.

The rest of this chapter is organized as follows. The Internet of Things (IoT) based smart-grid and smart building are presented in Sect. 3.2. The machine learning algorithm based on least-squares and backwards propagation is discussed in Sect. 3.3. Then Sect. 3.4 elaborates the Cholesky decomposition based least-squares solver. In Sect. 3.5, detailed implementation on FPGA hardware is elaborated. Experimental results regarding accuracy, speed-up, and energy consumption by FPGA implementation are presented in Sect. 3.6 with conclusion drawn in Sect. 3.7.

3.2 IoT System Based Smart Building

3.2.1 Smart-Grid Architecture

The overall Internet of Things (IoT) based smart-grid and smart building system is illustrated in Fig. 3.1. The key components from smart gird are the two-directional main electricity power grid and additional renewable energy based electricity power grid. By utilizing smart-grid, customers cannot only buy electricity from main power grid but also sell electricity from renewable solar energy to generate profits with dynamic prices. Smart building is the main element of the smart-grid for power consumption. Therefore, accurately predicting the energy demand of building can support the balance between supply and demand of smart-grid.

3.2.2 Smart Gateway for Real-Time Data Analytics

Smart gateway is the major control center, harboring the ability in storage and computation. Our smart gateway will be BeagleBoard-xM. As Fig. 3.1 shows, smart building is an IoT based system with various connected sensors. Environment sensors can collect information on light intensity, humidity, and temperature and send the data to micro-controller to understand environment. Energy sensor are used to collect the current of each appliance and through smart sockets, on-off control can be performed according to save energy. Moreover, occupancy provides information about the activity of occupants and location base services such as lighting and air-con can be provided accordingly. All these smart control is operated based on the pattern defined in the micro-controller and learnt by supervised learning process. Therefore, it is important to recognize (classify) the environment and occupants behavior to respond accordingly for customized services. Moreover, accurately predicting the energy demand of next minute or hour and then adjusting the supply

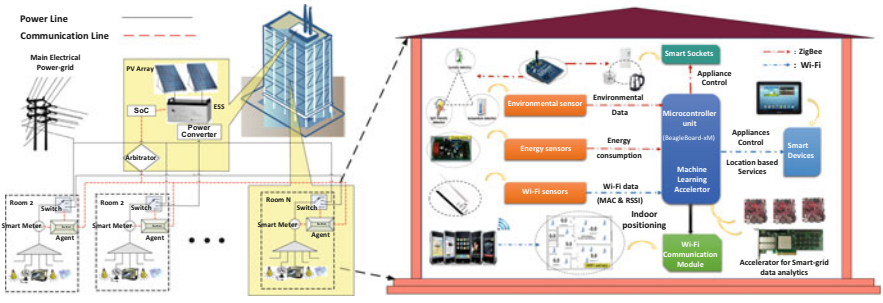

Fig. 3.1 Internet of things (IoT) based smart-grid and smart building system with renewable solar energy

are the key to achieve load balancing. However, due to the limited computation resource of smart-gateway, an FPGA based machine learning accelerator is designed to perform fast interference, model update, and re-train the machine learning model.

3.2.3 Problem Formulation for Data Analytics

In this chapter, data analytics refers to supervised machine learning, which is classification problem and regression problem. The classification problem is used for recognitions and regression problem is for prediction such as load forecasting. Details of each problem formulation are shown as below.

Objective 1: Minimize the error rate of classification.

$$\begin{aligned} \min e &= \sum_{i}^{N} x_i / N \\ \text{s.t. } x_i &= f(fe_1, fe_2, \ldots) = \{0, 1\} \end{aligned} \tag{3.1}$$

where $f(\cdot)$ represents the trained model from training data and $fe_1, fe_2 \ldots$ represent input data for this model. $x_i = 0$ represents the accurate prediction, $x_i = 1$ represents the false prediction, and N represents the number of predictions.

Objective 2: Improve the accuracy of energy demand forecasting with time interval t.

$$\min \ er = \sum_{t=0}^{t=23} (y_t - f(E_t, M_t, T_t))^2 \tag{3.2}$$

where y_t is actual energy demand at time t and $f(E_t, M_t, T_t)$ is the model predicted result with input features: energy consumption data E_t, occupants motion profile M_t and environmental T_t until time t. $f(\cdot)$ is the machine learning trained model. Once the new energy consumption data is ready, the machine learning model will be re-trained with new arrival data to build up customized energy forecasting model.

3.3 Background on Neural Network Based Machine Learning

In this section, the fundamental of neural network based machine learning is introduced with comparison of two training methods.

Neural network (NN) is a family of network models inspired by biological neural network to build the link for a large number of input–output data pair. It typically has two computational phases: training phase and testing phase.

Table 3.1 A list of parameters definitions in machine learning

Parameter	Elements	Definitions
X	$[x_{11}, x_{12}, x_{13},\ldots,x_{Nn}]$	A set of n dimension data in N training samples
T	$[t_{11}, t_{12}, t_{13},\ldots,t_{NM}]$	A set of M target classes in N training samples
H	$[h_{11}, h_{12}, h_{13},\ldots,h_{NL}]$	A set of L hidden nodes in N training samples
A	$[a_{11}, a_{12}, a_{13},\ldots,a_{nL}]$	Input weight matrix between X and H
$\boldsymbol{\Gamma}$	$[\gamma_{11}, \gamma_{12}, \gamma_{13},\ldots,\gamma_{LM}]$	Output weight matrix between H and T
Y	$[y_1, y_2, y_3,\ldots,y_M]$	A set of M model outputs
$\boldsymbol{\beta}$	N.A.	Learning rate set by designers

– In the training phase, the weight coefficients of the neural network model are first determined using training data by minimizing the squares of error difference between trial solution and targeted data in a so-called ℓ_2-norm method.
– In the testing phase, the neural network model with determined weight coefficients is utilized for classification or calculation given the new input of data.

Formally, the detailed descriptions of each parameter are summarized in Table 3.1. Given a neural network with n inputs and M outputs shown in Fig. 3.2, a dataset $(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)$ is composed of paired input data $\mathbf{X}$ and training data $\mathbf{T}$ with N number of training samples, n dimensional input features and M classes. During the training, one needs to minimize the ℓ_2-norm error function with determined weights: $\mathbf{A}$ (at input layer) and $\boldsymbol{\Gamma}$ (at output layer):

$$E = ||\mathbf{T} - F(\mathbf{A}, \boldsymbol{\Gamma}, \mathbf{X})||_2 \tag{3.3}$$

where $F(\cdot)$ is the mapping function from the input to the output of the neural network.

The output function of this neural network classifier is

$$\begin{aligned} &\mathbf{Y} = \mathbf{H}F(\mathbf{A}, \boldsymbol{\Gamma}, \mathbf{X}), \quad \mathbf{Y} = \{y_1, y_2, \ldots y_m\} \\ &\text{Label}(X) = \underset{i\in\{1,2,\ldots,m\}}{\arg\max}\ y_i \end{aligned} \tag{3.4}$$

where $\mathbf{Y} \in \mathbb{R}^{N\times m}$. Here N represents the number of testing samples. The index of maximum value $\mathbf{Y}$ is found and identified as the predicted class.

3.3.1 Backward Propagation for Training

The first method to minimize the error function E is the Backward Propagation (BP) method. As shown in Fig. 3.2a, the weights are firstly initially guessed for forward

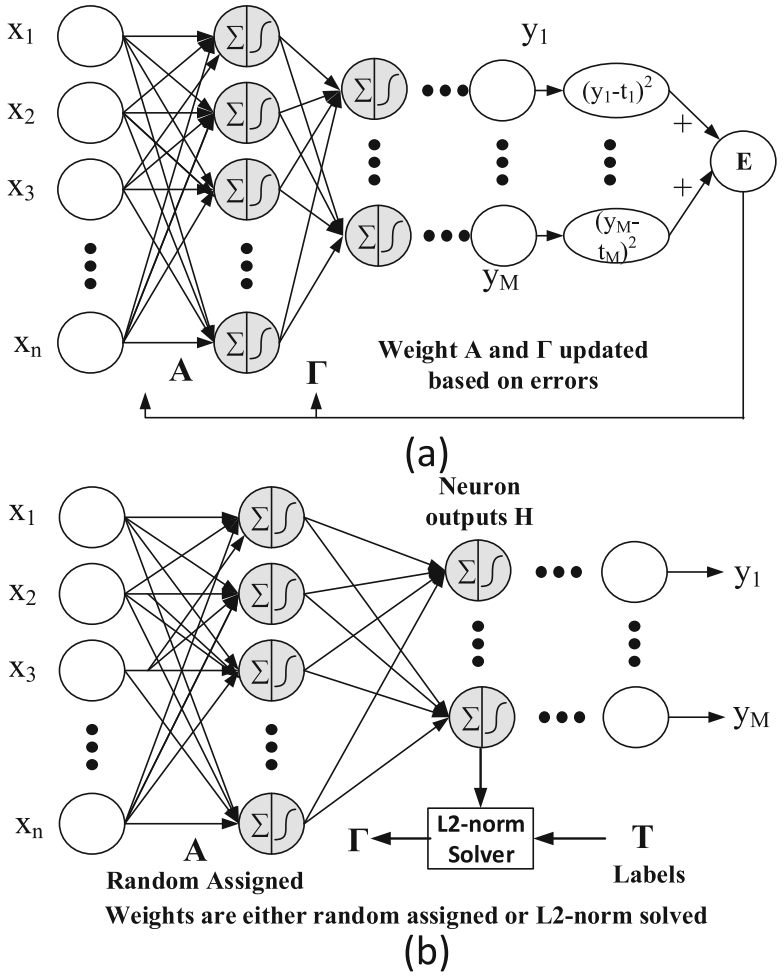

Fig. 3.2 Trainings of neural network: (**a**) backward propagation; and (**b**) least-square solver

propagation. Based on the trial error, the weights are further calculated backward with derivatives of weights calculated by

$$\nabla E = \left(\frac{\partial E}{\partial a_{11}}, \frac{\partial E}{\partial a_{12}}, \frac{\partial E}{\partial a_{13}} \cdots \frac{\partial E}{\partial a_{DL}} \right) \tag{3.5}$$

where D is the output dimension of previous layer and L is the input dimension of the next layer. For the current layer, each weight can be updated as

$$a_{dl} = a_{dl} - \beta * \frac{\partial E}{\partial a_{dl}},\ d = 1, 2, \ldots, D,\ l = 1, 2, \ldots, L \tag{3.6}$$

where β is the learning constant that defines the step length of each iteration in the negative gradient direction. Note that the BP method requires to store the derivatives of each weight. It is expensive for hardware realization. More importantly, it may be trapped on local minimal with long converging time. Hence, the BP based training is usually performed off-line and has large latency when analyzing the real-time sensed data.

3.3.2 Least-Squares Solver for Training

One can directly solve the least-squares problem using the least-squares solvers of the ℓ_2-norm error function E [11–13]. As shown in Fig. 3.2b, the input weight $\mathbf{A}$ can be first randomly assigned and one can directly solve output weight $\mathbf{\Gamma}$ as follows.

We first find the relationship between the hidden neural node and input training data as

$$\mathbf{preH} = \mathbf{XA} + \mathbf{B},\ \mathbf{H} = \frac{1}{1 + e^{-\mathbf{preH}}} \tag{3.7}$$

where $\mathbf{X} \in \mathbb{R}^{N \times n}$. $\mathbf{A} \in \mathbb{R}^{n \times L}$ and $\mathbf{B} \in \mathbb{R}^{N \times L}$ is random generated input weight and bias formed by a_{ij} and b_{ij} between $[-1, 1]$. N and n are the training size and the dimension of training data, respectively. The output weight $\boldsymbol{\Gamma}$ is computed based on pseudo-inverse ($L < N$):

$$\mathbf{\Gamma} = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{T} \tag{3.8}$$

However, performing pseudo-inverse is also expensive for hardware realization.

The comparison of BP and least-squares solver can be summarized as follows. BP is a relative simple implementation by gradient descent objective function with good performances. However, it suffers from the long training time and may get stuck in the local optimal point. On the other hand, least-squares solver can learn very fast, but pseudo-inverse is too expensive for calculating. Therefore, solving ℓ_2-norm minimization efficiently becomes the bottleneck of the training process.

3.3.3 Feature Extraction with Behavior Cognition

Input features are very important to train an accurate machine learning model. In this chapter, occupants behavior is analyzed based on the active occupant motion in each room since it indicates the potential behavior of occupants in the room [14]. Rooms inside the same house have vastly different occupants behavior profiles due to different functionalities. Therefore, we extracted behavior profiles for different

rooms, respectively. For each room i, there are four states represented by S for occupants positioning:

$$S = \begin{cases} s_1 : 0 & \text{no occupant in the room } i \\ s_2 : 0 \longrightarrow 1 & \text{occupants entering the room } i \\ s_3 : 1 & \text{occupants in the room } i \\ s_4 : 1 \longrightarrow 0 & \text{occupants leaving the room } i \end{cases} \tag{3.9}$$

where motion state S is detected by indoor positioning system via WiFi data every minute. The probability of occupants motion for room i can be expressed as:

$$M_i(t) = \frac{Ti(s_2) + Ti(s_3)}{Ti}, \quad t = 1, 2, 3, \ldots, 96 \tag{3.10}$$

where $Ti(s_j)$ represents the time duration with corresponding state s_j. $M_i(t)$ is occupant motion probability of room i in Ti time interval. Figure 3.3 presents an example of motion probability in different rooms. As a conclusion, all the features and their descriptions for data analytics are summarized in Table 3.2.

Our work differs from previous works [15, 16] such as sequential learning or recursive learning from two manifold. Firstly, the training data size in our work is fixed size with adding new arrival data and removing old data. This is preferred since environment and occupants change with several levels of seasonality [3]. Old data from months ago tend to bias the new change of load demand. Secondly, our learning algorithm focuses on tuning the size of neural network. Since training data is changed, it is more effective to re-train the model than using sequential learning method to update the out-dated model.

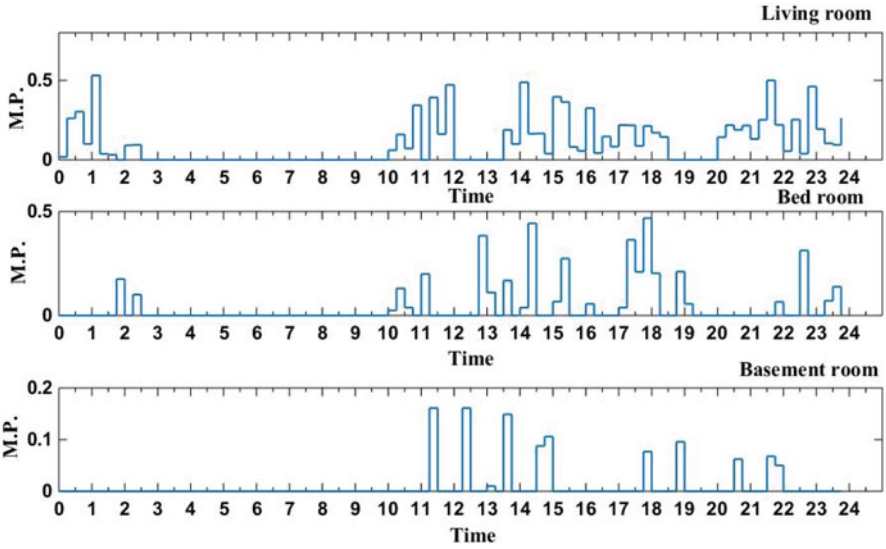

Fig. 3.3 Motion probability within 15 min interval in three different rooms (Living room, bed room, and basement)

Table 3.2 Input features for short-term load forecasting

Inputs	Descriptions
1	Date type: weekday is represented by 1 and weekend is represented by 0
2–25	Eg(d-7,t), Eg(d-6,t), Eg(d-5,t), Eg(d-4,t), Eg(d-3,t), Eg(d-2,t), Eg(d-1,t): Energy of the 7 days preceding to the forecasted day at the same hour
26–121	Mo(d-7,t), Mo(d-6,t), Mo(d-5,t), Mo(d-4,t), Mo(d-3,t), Mo(d-2,t), Mo(d-1,t): Occupants motion of the 7 days preceding to the forecasted day at the same hour
122–169	Te(d-7,t), Te(d-6,t), Te(d-5,t), Te(d-4,t), Te(d-3,t), Te(d-2,t), Te(d-1,t): Temperature and Humidity of the 7 days preceding to the forecasted day at the same hour
169-t	C(t), C(t-1), C(t-2),. . . ,: Prior to *time t*, new collected data temperature, humanity, energy and occupants motion

3.4 Least-Squares Solver Based Training Algorithm

In this section, we firstly reformulate a regularized least-squares problem. Then square-root-free Cholesky decomposition is discussed to reduce the complexity. Final, an incremental least-squares method is introduced to further simplify the operation to basic linear algebra subprograms (BLAS).

3.4.1 Regularized ℓ_2-Norm

Considering (3.8), a better generalized training method is to minimize the training error and the norm of the output weights, which can be defined as a regularized ℓ_2-norm as follows:

$$\min ||\mathbf{H}\boldsymbol{\Gamma} - \mathbf{T}||_2 + \lambda||\boldsymbol{\Gamma}||_2 \tag{3.11}$$

where $\mathbf{H}$ is the hidden-layer output matrix generated from the Sigmoid function for activation; and λ is a user defined parameter that biases the training error and output weights [11]. This problem can be reformulated as

$$\begin{aligned} &\min ||\tilde{\mathbf{H}}\boldsymbol{\Gamma} - \tilde{\mathbf{T}}||_2 \\ &\text{where } \tilde{\mathbf{H}} = \begin{pmatrix} \mathbf{H} \\ \sqrt{\lambda}\mathbf{I} \end{pmatrix} \quad \tilde{\mathbf{T}} = \begin{pmatrix} \mathbf{T} \\ \mathbf{0} \end{pmatrix} \end{aligned} \tag{3.12}$$

where $\mathbf{I} \in \mathbb{R}^{L\times L}$ and $\tilde{\mathbf{H}} \in \mathbb{R}^{(N+L)\times L}$. This is a standard least-squares problem with general solution:

$$\boldsymbol{\Gamma} = (\tilde{\mathbf{H}}^T\tilde{\mathbf{H}})^{-1}\tilde{\mathbf{H}}^T\tilde{\mathbf{T}}, \ \tilde{\mathbf{H}} \in \mathbb{R}^{N\times L} \tag{3.13}$$

where $\tilde{\mathbf{T}} \in \mathbb{R}^{(N+L)\times M}$ and M is the number of classes. The new training algorithm is summarized in Algorithm 1. The complexity of solving output weight will be reduced by the square-root-free Cholesky decomposition and incremental least-squares solutions.

Algorithm 1 The proposed training algorithm of neuron network

Input: Training Set $(x_i, t_i), x_i \in \mathbf{R}^n, t_i \in \mathbf{R}^M, i = 1, \ldots N$, activation function $G(a_i, b_i, x_j)$, maximum number of hidden neuron node L and accepted training error ϵ.

Output: Neuron Network output weight Γ

1: Randomly assign hidden-node parameters
$(a_{ij}, b_{kj}),\ i = 1, 2, \ldots, n,\ j = 1, \ldots, l,\ k = 1, 2, \ldots, N$;
2: Calculate the hidden-layer output matrix $\mathbf{H}$
$\mathbf{preH} = \mathbf{XA} + \mathbf{B},\ \mathbf{H} = \mathbf{1}/(\mathbf{1} + e^{-\mathbf{preH}})$
3: Form regularized ℓ_2-norm
$\tilde{\mathbf{H}} = \begin{pmatrix} \mathbf{H} \\ \sqrt{\lambda}\mathbf{I} \end{pmatrix} \tilde{\mathbf{T}} = \begin{pmatrix} \mathbf{T} \\ \mathbf{0} \end{pmatrix}$
4: Calculate the output weight
$\mathbf{\Gamma} = (\tilde{\mathbf{H}}^T\tilde{\mathbf{H}})^{-1}\tilde{\mathbf{H}}^T\tilde{\mathbf{T}}$
5: IF ($l \le L$ or $error > \epsilon$)
Increase number of hidden node
$l = l + 1$, repeat from Step 1
6: ENDIF

3.4.2 Square-Root-Free Cholesky Decomposition

The main step for a direct solution of the training problem is the standard least-squares problem of minimizing $||\tilde{\mathbf{T}} - \tilde{\mathbf{H}}\boldsymbol{\Gamma}||_2$. This can be the solution using SVD, QR, and Cholesky decomposition. The computational cost of SVD, QR, or Cholesky decomposition for the problem is $O(4(N+L)L^2 - \frac{4}{3}L^3)$, $O(2(N+L)L^2 - \frac{2}{3}L^3)$, and $O(\frac{1}{3}L^3)$, respectively [17]. Therefore, we use Cholesky decomposition to solve the least-squares problem. Moreover, its incremental and symmetric property reduces the computational cost and hence saves half of memory required [17]. Here, we use H_L to represent the matrix with L number of hidden neuron nodes, which decomposes the symmetric positive definite matrix $\tilde{\mathbf{H}}^T\tilde{\mathbf{H}}$ into

$$\tilde{\mathbf{H}}^T\tilde{\mathbf{H}} = \mathbf{QDQ}^{\mathbf{T}} \tag{3.14}$$

where $\mathbf{Q}$ is a lower triangular matrix with diagonal elements $q_{ii} = 1$ and $\mathbf{D}$ is a positive diagonal matrix. Such method can maintain the same space as Cholesky

factorization but avoid the extracting the square roots as the square root of $\mathbf{Q}$ is resolved by diagonal matrix $\mathbf{D}$ [18].

$$\begin{aligned}\tilde{\mathbf{H}}_L^T\tilde{\mathbf{H}}_L &= \left[\tilde{\mathbf{H}}_{L-1}\ h_L\right]^T\left[\tilde{\mathbf{H}}_{L-1}\ h_L\right] \\ &= \begin{pmatrix}\tilde{\mathbf{H}}_{L-1}^T\tilde{\mathbf{H}}_{L-1} & \mathbf{v}_L \\ \mathbf{v}_L^T & g\end{pmatrix}\end{aligned} \tag{3.15}$$

where $(\mathbf{v}_L, g)$ is a new column generated from new data $h_L^T h_L$, compared to $\tilde{\mathbf{H}}_{L-1}^T\tilde{\mathbf{H}}_{L-1}$. Therefore, we can find

$$\mathbf{Q}_L\mathbf{D}_L\mathbf{Q}_L^T = \begin{pmatrix}\mathbf{Q}_{L-1} & 0 \\ \mathbf{z}_L^T & 1\end{pmatrix}\begin{pmatrix}\mathbf{D}_{L-1} & 0 \\ 0 & d\end{pmatrix}\begin{pmatrix}\mathbf{Q}_{L-1}^T & \mathbf{z}_L \\ 0 & 1\end{pmatrix} \tag{3.16}$$

Therefore, we can easily calculate the vector $\mathbf{z}_L$ and scalar d for Cholesky decomposition as

$$\mathbf{Q}_{L-1}\mathbf{D}_{L-1}\mathbf{z}_L = \mathbf{v}_L,\ d = g - \mathbf{z}_L^T\mathbf{D}_{L-1}\mathbf{z}_L \tag{3.17}$$

where $\mathbf{Q}_L$ and $\mathbf{v}_L$ are known from (3.15), which means that we can continue to use previous factorization result and only update according part. Algorithm 2 shows more details on each step. Note that Q_1 is 1 and D_1 is $\tilde{\mathbf{H}}_1^T\tilde{\mathbf{H}}_1$.

3.4.3 Incremental Least-Squares Solution

The optimal residual for least-squares problem $\tilde{\mathbf{H}}\boldsymbol{\Gamma} = \mathbf{T}$ is defined as r:

$$r = \mathbf{T} - \tilde{\mathbf{H}}\Gamma_{ls} = (\tilde{\mathbf{H}}(\tilde{\mathbf{H}}^T\tilde{\mathbf{H}})^{-1}\tilde{\mathbf{H}}^T - \mathbf{I})\mathbf{T} \tag{3.18}$$

Therefore, r is orthogonal to $\tilde{\mathbf{H}}$, where the projection of r to $\tilde{\mathbf{H}}$ is

$$< r, \tilde{\mathbf{H}} >= \mathbf{T}^T(\tilde{\mathbf{H}}(\tilde{\mathbf{H}}^T\tilde{\mathbf{H}})^{-1}\tilde{\mathbf{H}}^T - \mathbf{I})^T\tilde{\mathbf{H}} = 0 \tag{3.19}$$

Similarly, for every iteration of Cholesky decomposition, x_{l-1} is the least-squares solution of $\mathbf{T} = \tilde{\mathbf{H}}_{\Lambda_{l-1}} * \boldsymbol{\Gamma}$ with the same orthogonality principle, where Λ_l is the selected column sets for matrix $\tilde{\mathbf{H}}$. Therefore, we have

$$\begin{aligned}\mathbf{T} &= r_{l-1} + \tilde{\mathbf{H}}_{\Lambda_{l-1}} * x_{l-1} \\ \tilde{\mathbf{H}}_{\Lambda_l}^T\tilde{\mathbf{H}}_{\Lambda_l}x_l &= \tilde{\mathbf{H}}_{\Lambda_l}^T(r_{l-1} + \tilde{\mathbf{H}}_{\Lambda_{l-1}} * x_{l-1})\end{aligned} \tag{3.20}$$

where x_{l-1} is the least-squares solution in the previous iteration. By utilizing superposition property of linear systems, we can have

$$\begin{bmatrix} \tilde{\mathbf{H}}_{\Lambda_l}^T \tilde{\mathbf{H}}_{\Lambda_l} x_{tp1} \\ \tilde{\mathbf{H}}_{\Lambda_l}^T \tilde{\mathbf{H}}_{\Lambda_l} x_{tp2} \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{H}}_{\Lambda_l}^T r_{l-1} \\ \tilde{\mathbf{H}}_{\Lambda_l}^T \tilde{\mathbf{H}}_{\Lambda_{l-1}} * x_{l-1} \end{bmatrix} \tag{3.21}$$

$$x_l = x_{tp1} + x_{tp2} = x_{tp1} + x_{l-1}$$

where the second row of equation has a trivial solution of $[x_{l-1}\ 0]^T$. Furthermore, this indicates that the solution of x_l is based on x_{l-1} and only x_{tp} is required to be computed out from the first row of (3.21), which can be expanded as

$$\tilde{\mathbf{H}}_{\Lambda_l}^T \tilde{\mathbf{H}}_{\Lambda_l} x_{tp1} = \begin{bmatrix} \tilde{\mathbf{H}}_{\Lambda_{l-1}}^T r_{l-1} \\ h_l^T r_{l-1} \end{bmatrix} = \begin{bmatrix} 0 \\ h_l^T r_{l-1} \end{bmatrix} \tag{3.22}$$

Due to the orthogonality between the optimal residual $\tilde{\mathbf{H}}_{\Lambda_{l-1}}$ and r_{l-1}, the dot product becomes 0. This clearly indicates that the solution x_{tp1} is a sparse vector with only one element. By substituting square-root-free Cholesky decomposition, we can find

$$\mathbf{Q}^T dx_{tp} = h_l^T r_{l-1} \tag{3.23}$$

where x_{tp} is the same as x_{tp1}. The other part of Cholesky factorization $\mathbf{Q}$ for multiplication of x_{tp1} is always 1 and hence is eliminated. The detailed algorithm including Cholesky decomposition and incremental least-squares is shown in Algorithm 2. By utilizing Cholesky decomposition and incremental least-squares techniques, the computational complexity is reduced with only 4 basic linear algebra operations per iterations.

3.5 Least-Squares Based Machine Learning Accelerator Architecture

3.5.1 Overview of Computing Flow and Communication

The top level of proposed VLSI architecture for training and testing is shown in Fig. 3.4. The description of this architecture will be introduced based on testing flow. The complex control and data flow of the neural network training and testing is enforced by a top level finite state machine (FSM) with synchronized and customized local module controllers.

For the neural network training and testing, an asynchronous first-in first-out (FIFO) is designed to collect data through AXI4 light from PCIe Gec3X8. Two buffers are used to store rows of the training data $\mathbf{X}$ to perform ping-pong operations.

Algorithm 2 Fast incremental least-squares solution

Input: Activation matrix $\tilde{\mathbf{H}}_L$, target matrix $\tilde{\mathbf{T}}$ and number of hidden nodes L
Output: Neuron Network output weight x

1: Initialize $r_0 = \tilde{\mathbf{T}}$, $\Lambda_0 = \emptyset$, $d = \mathbf{0}$, $x_0 = \mathbf{0}$, $l = 1$,
2: While $||r_{l-1}||_2^2 \leq \epsilon^2$ or $l \leq L$
3: $c(l) = h_l^T r_{l-1}$, $\Lambda_l = \Lambda_{l-1} \cup l$
4: $\mathbf{v}_l = \tilde{\mathbf{H}}_{\Lambda_l}^T h_l$
5: $\mathbf{Q}_{l-1} w = \mathbf{v}_l(1 : l-1)$. $\mathbf{z}_l = w./diag(\mathbf{D}_{l-1})$
6: $d = g - \mathbf{z}_l^T w$
7: $\mathbf{Q}_l = \begin{bmatrix} \mathbf{Q}_{l-1} & \mathbf{0} \\ \mathbf{z}_l^T & 1 \end{bmatrix}$, $\mathbf{D}_l = \begin{bmatrix} \mathbf{D}_{l-1} & \mathbf{0} \\ \mathbf{0} & d \end{bmatrix}$
$(\mathbf{Q}_1 = 1, \mathbf{D}_1 = h_1 * h_1^T)$
8: $\mathbf{Q}_l^T x_{tp} = \begin{bmatrix} \mathbf{0} \\ c(l)/d \end{bmatrix}$
9: $x_l = x_{l-1} + x_{tp}$, $r_l = r_{l-1} - \tilde{\mathbf{H}}_{\Lambda_l} x_{tp}$, $l = l + 1$
10: END While

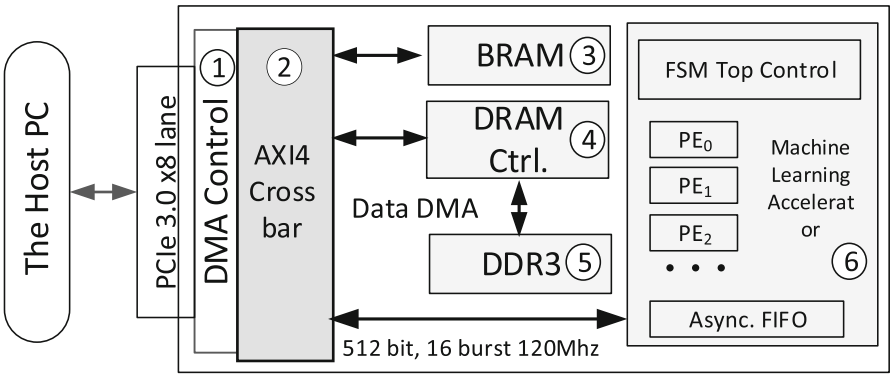

Fig. 3.4 Accelerator architecture for training and testing

These two buffers will be re-used when collecting the output weight data. To maintain high training accuracy, floating point data is used with parallel fixed point to floating point converter. As the number indicated on each block in Fig. 3.4, data will be firstly collected through PCIe to DRAM and Block RAM. Block RAM is used to control the core to indicate the read/write address of DRAM during the training/testing process. The core will continuously read data from block RAM for configurations and starting signal. Once data is ready in DRAM and the start signal is asserted, the core will process computation for neural network testing or training process. An implemented FPGA block design on Vivado is shown in Fig. 3.9.

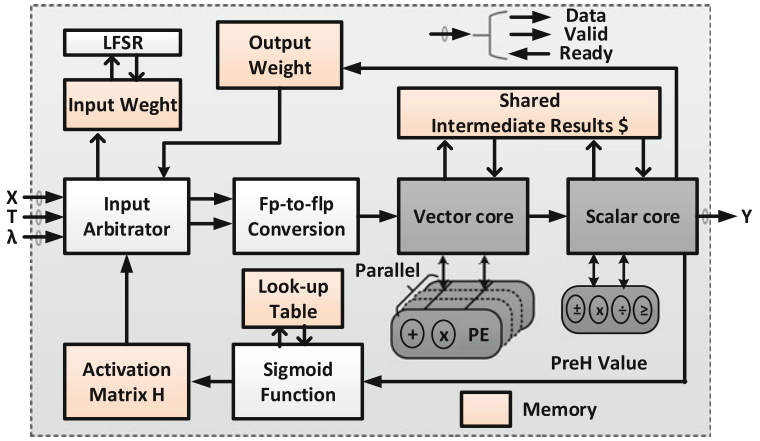

Fig. 3.5 Detailed architecture for online learning

3.5.2 FPGA Accelerator Architecture

As mentioned in the Sect. 3.3, operations in neural network are performed serially from one layer to the next. This dependency reduces the level of parallelism of accelerator and requires more acceleration in each layer. In this chapter, a folded architecture is proposed as shown in Fig. 3.5. Firstly, the *input arbitrator* will take input training data and input weight. A pipeline stage is added for activation after each multiplication result. Then depending on the mode of training and testing, the *input arbitrator* will decide to take label or output weight. For testing process, output weight is selected for calculation neural network output. For training, label will be taken for output weight calculation based on Algorithm 2. To achieve similar software-level accuracy, floating-point data is used during the computation process and 8-bit fixed point is used for data storage.

3.5.3 ℓ_2-Norm Solver

As mentioned in the reformulated ℓ_2-norm Algorithm 2, Step 5 requires forward substitutions. Figure 3.6 provides the detailed mapping for forward substitutions on our proposed architecture. For the convenient purposes, we use $\mathbf{QW} = \mathbf{V}$ to represent Step 5, where $\mathbf{Q}$ is a triangular matrix. Figure 3.7 provides the detailed equations in each PEs and stored intermediate values. To explore the maximum level of parallelism, we can perform multiplication at the same time on each row to compute $w_i, i \neq 1$ as shown in the right of Fig. 3.7. However, there is different number of multiplication and accumulations required for different w_i. In the first round, to have the maximum level of parallelism, intuitively we require $L-1$ parallel

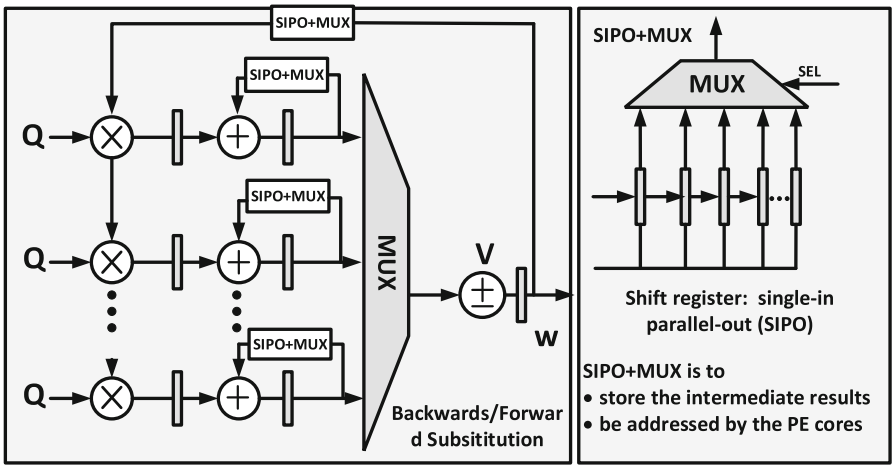

Fig. 3.6 Computing diagram of forward/backward substitution in L2-norm solver

```
For j=1:L-1                          // Pseudo-code for column oriented forward subsitution
    v(j+1:L)= v(j+1:L)-v(j)·q(j+1:L,j)  // Overwrites v with the solution x
End                                  // Parallel computation for dot product (MAC)
```

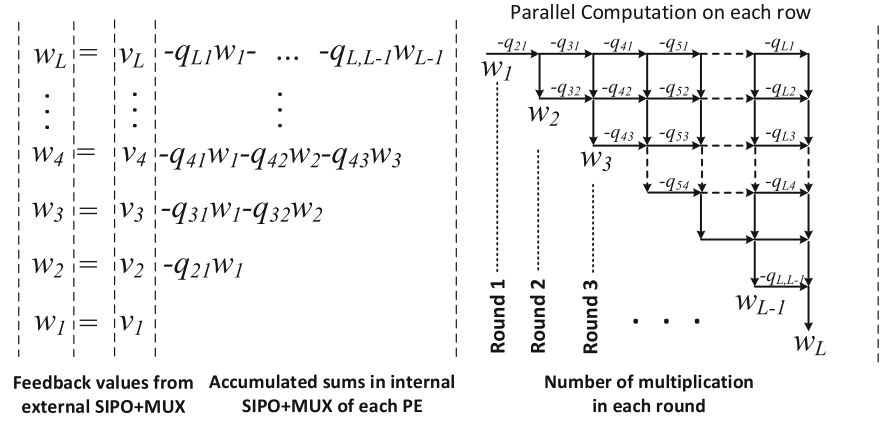

Fig. 3.7 Detailed mapping of forward substitution

PEs to perform the multiplication. After knowing w_2, we need $L-2$ parallel PEs for the same computations in the second round. However, if we add a shift register, we can store the intermediate results in the shift register and take it with a decoder of MUX. For example, if we have parallelism of 4 for $L = 32$, we can perform 8 times parallel computation for the round 1 and store them inside registers. This helps improve the flexibility of the process elements (PEs) with better resource utilization.

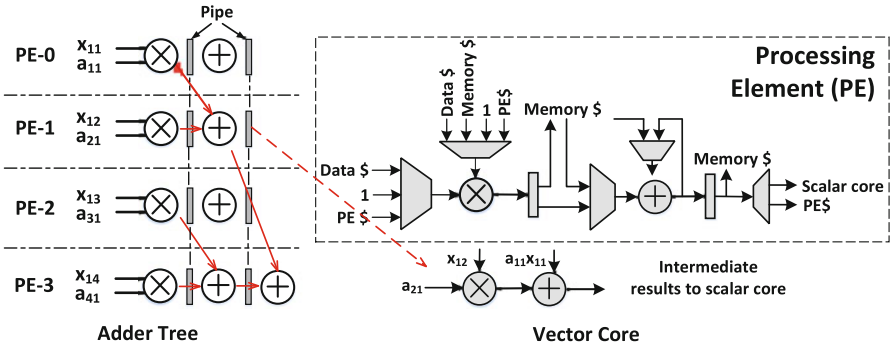

Fig. 3.8 Computing diagram of matrix–vector multiplication

3.5.4 Matrix–Vector Multiplication

All the computation relating to vector operation is performed on processing elements (PEs). Our designed PE is similar as [19] but features direct instruction to perform vector–vector multiplications for neural network. Figure 3.8 gives an example of vector–vector multiplication (dot product) for (3.7) with parallelism of 4. If the vector length is 8, the folding factor will be 2. The output from PE will be accumulated twice based on the folding factor before sending out the vector–vector multiplication result. The adder tree will be generated based on the parallelism inside vector core. The output will be passed to scalar core for accumulations. In the PE, there is a bus interface controller. It will control the multiplicand of PE and pass the correct data based on the top control to PE.

3.6 Experiment Results

In this section, we firstly discuss the machine learning accelerator architecture and resource usage. Then details of FPGA implementation with CAD flow are discussed. The performance of proposed scalable architecture is evaluated for regression problem and classification problem, respectively. Finally, the energy consumption and speed-up of proposed accelerator are evaluated in comparison with CPU, embedded CPU and GPU.

3.6.1 Experiment Setup and Benchmark

To verify our proposed architecture, we have implemented in on Xilinx Virtex 7 with PCI Express Gen3x8 [20]. The HDL code is synthesized using Synplify and

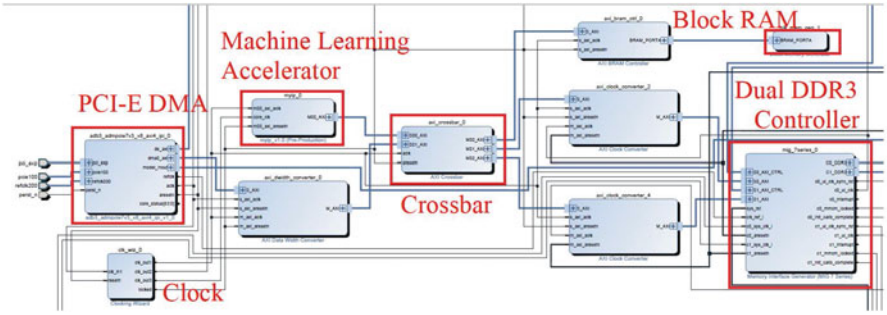

Fig. 3.9 Vivado block design for FPGA least-squares machine learning accelerator

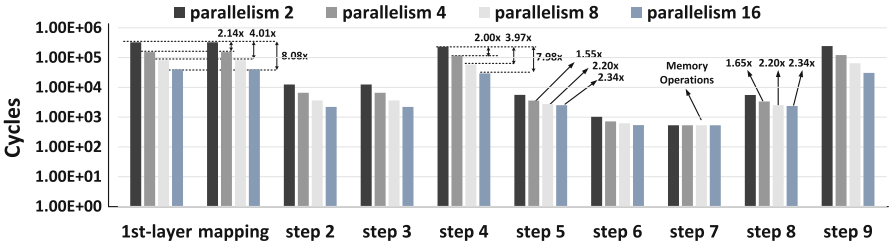

Fig. 3.10 Training cycles at each step of the proposed training algorithm with different parallelisms ($N = 74$; $L = 38$; $M = 3$ and $n = 16$)

the maximum operating frequency of the system is 53.1 MHz under 128 parallel PEs. The critical path is identified as the floating-point division, where 9 stages of pipeline are inserted for speedup. We develop three baselines (x86 CPU, ARM CPU, and GPU) for performance comparisons.

Baseline 1: General Processing Unit (x86 CPU). The general CPU implementation is based on C program on a computer server with Intel Core -i5 *3.20GHz* core and *8.0GB* RAM.

Baseline 2: Embedded processor (ARM CPU). The embedded CPU (BeagleBoard-xM) [21] is equipped with 1GHz ARM core and 512MB RAM. The implementation is performed using C program under Ubuntu 14.04 system.

Baseline 3: Graphics Processing Unit (GPU). The GPU implementation is performed by CUDA C program with cuBLAS library. A Nvidia GeForce GTX 970 is used for the acceleration of learning on neural network.

The dataset for residential load forecasting is collected by Singapore Energy Research Institute (ERIAN). The dataset consists of 24-henergy consumptions, occupants motion, and environmental records such as humidity and temperatures from 2011 to 2015. Features for short-term load forecasting is summarized in Table 3.2. Please note that we will perform hourly load forecasting using real-time environmental data, occupants motion data, and previous hours and days energy consumption data. Model will be retrained sequentially after each hour with new generated training data.

3.6.2 FPGA Design Platform and CAD Flow

The ADM-PCIE-7V3 is a high-performance reconfigurable computing card intended for high speed performance applications, featuring a Xilinx Virtex-7 FPGA. The key features of ADM-PCIE 7V3 are summarized as below [20]

- Compatible with Xilinx OpenCL compiler
- Supported by ADM-XRC Gen 3 SDK 1.7.0 or later and ADB3 Driver 1.4.15 or later.
- PCIe Gen1/2/3 x1/2/4/8 capable
- Half-length, low-profile x8 PCIe form factor
- Two banks of DDR3 SDRAM SODIMM memory with ECC, rated at 1333 MT/s
- Two right angle SATA connectors (SATA3 capable)
- Two SFP+ sites capable of data rates up to 10 Gbps
- FPGA configurable over JTAG and BPI Flash
- XC7VX690T-2FFG1157C FPGA

The development platform is mainly on Vivado 14.4. The direct memory access (DMA) bandwidth is 4.5GB/s. The DDR3 bandwidth is 1333 MT/s with 64 bits width.

The CAD flows for implementing the machine learning accelerator on the ADM-PCIE 7V3 are illustrated in Fig. 3.11. The Xilinx CORE Generator System is first used to generate the data memory macros that are mapped to the BRAM resources on the FPGA. The generated NGC files contain both the design netlist, constraints files and Verilog wrapper. Then, these files together with the RTL codes of the machine learning accelerator are loaded to Synplify Premier for logic synthesis.

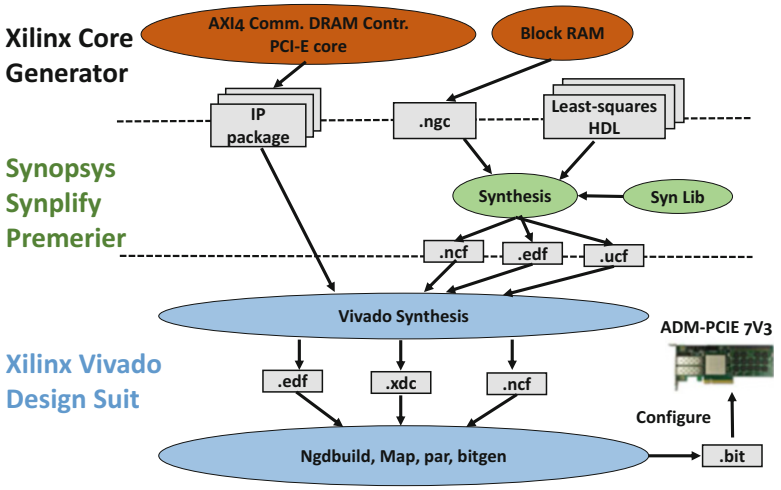

Fig. 3.11 CAD flows for implementing least-squares on ADM-PCIE 7V3

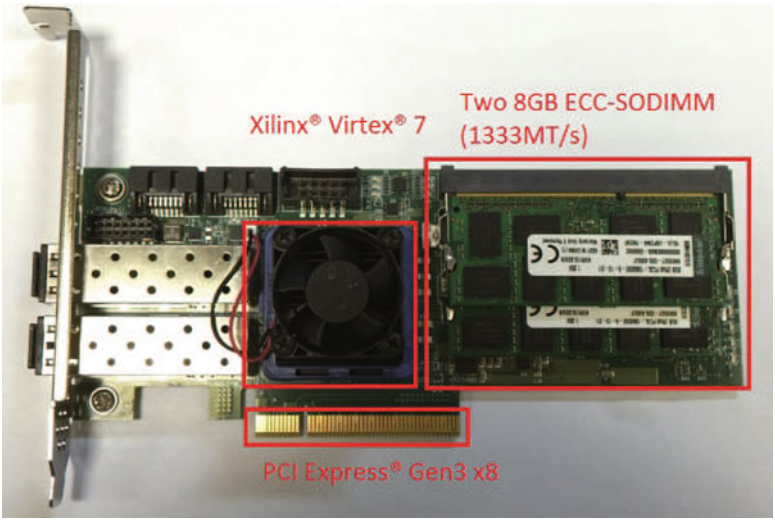

Fig. 3.12 Alpha-Data PCIe 7V3 FPGA board

Note that the floating-point arithmetic units used in our design are from the Synopsys DesignWare library. The block RAM is denoted as black box for Synplify synthesis. The EDF file stores the gate-level netlist in an electronic data interchange format (EDIF), and the UCF file contains user-defined design constraints. Next, the generated files are passed to Xilinx Vivado Design Suite to merge with other IP core such as DRAM controller and PCI-E core. In the Vivado design environment, each IP is packaged and connected. Then, we synthesize the whole design again under Vivado environment. Specifically, the "ngbbuild" command reads in the netlist in EDIF format and creates a native generic database (NGD) file that contains a logical description of the design reduced to Xilinx NGD primitives and a description of the original design hierarchy. The "map" command takes the NGD file, maps the logic design to a specific Xilinx FPGA, and outputs the results to a native circuit description (NCD) file. The "par" command takes the NCD file, places and routes the design, and produces a new NCD file, which is then used by the "bitgen" command for generating the bit file for FPGA programming. Figure 3.12 shows Alpha-Data PCIe FPGA board.

3.6.3 Scalable and Parameterized Accelerator Architecture

The proposed accelerator architecture features great scalability for different applications. Table 3.3 shows all the user-defined parameters supported in our architecture. At circuit level, users can adjust the stage of pipeline of each arithmetic to satisfy the speed, area, and resource requirements. At architecture level, the parallelism of PE

Table 3.3 Tunable parameters on proposed architecture

Parameters		Descriptions
Circuits	{MAN EXP}	Word-length of mantissa, exponent
	$\{P_A, P_M, P_D, P_C\}$	Pipe. stages of adder, mult, div and comp
Architectures	P	Parallelism of PE in VC
	n	Maximum signal dimensions
	N	Maximum training/test data size
	H	Maximum number of hidden nodes

Table 3.4 Resource utilization under different parallelism level ($N = 512$, $H = 1024$, $n = 512$ and 50 Mhz clock)

Paral.	LUT	Block RAM	DSP
8	52,614 (12%)	516 (35%)	51 (1.42%)
16	64,375 (14%)	516 (35%)	65 (1.81%)
32	89,320 (20%)	516 (35%)	96 (2.67%)
64	139,278 (32%)	516 (35%)	160 (4.44%)
128	236,092 (54%)	516 (35%)	288 (8.00%)

can be specified based on the hardware resource and speed requirement. The neural network parameters n, N, H can be also reconfigured for specific applications.

Figure 3.10 shows the training cycles on each step on proposed training algorithms for synthesized dataset. Different parallelism P is applied to show the speed-up of each steps. The speed-up of 1st-layer for matrix–vector multiplication is scaling up with the parallelism. The same speed-up improvement is also observed in the Step 3, 4, and 9 in Algorithm 2, where the matrix–vector multiplication is the dominant operation.

However, when the major operation is the division for the backward and forward substitution, the speed-up is not that significant and tends to saturate when the division becomes the bottleneck. We can also observe in Step 7, the memory operations do not scale with parallelism. It clearly shows that matrix–vector multiplication is the dominant operation in the training procedure (1st Layer, Step 3, Step 4, and Step 9) and our proposed accelerator architecture is scalable to dynamically increase the parallelism to adjust the speed-up.

The resource utilization under different parallelism is achieved from Xilinx ISE after place and routing. From Table 3.4, we can observe that LUT and DSP are almost linearly increasing with parallelism. However, Block RAM keeps constant with increasing parallelism. This is because Block RAM is used for data buffer, which is determined by other architecture parameters (N, H, n). Figure 3.13 shows the layout view of the FPGA least-squares solver.

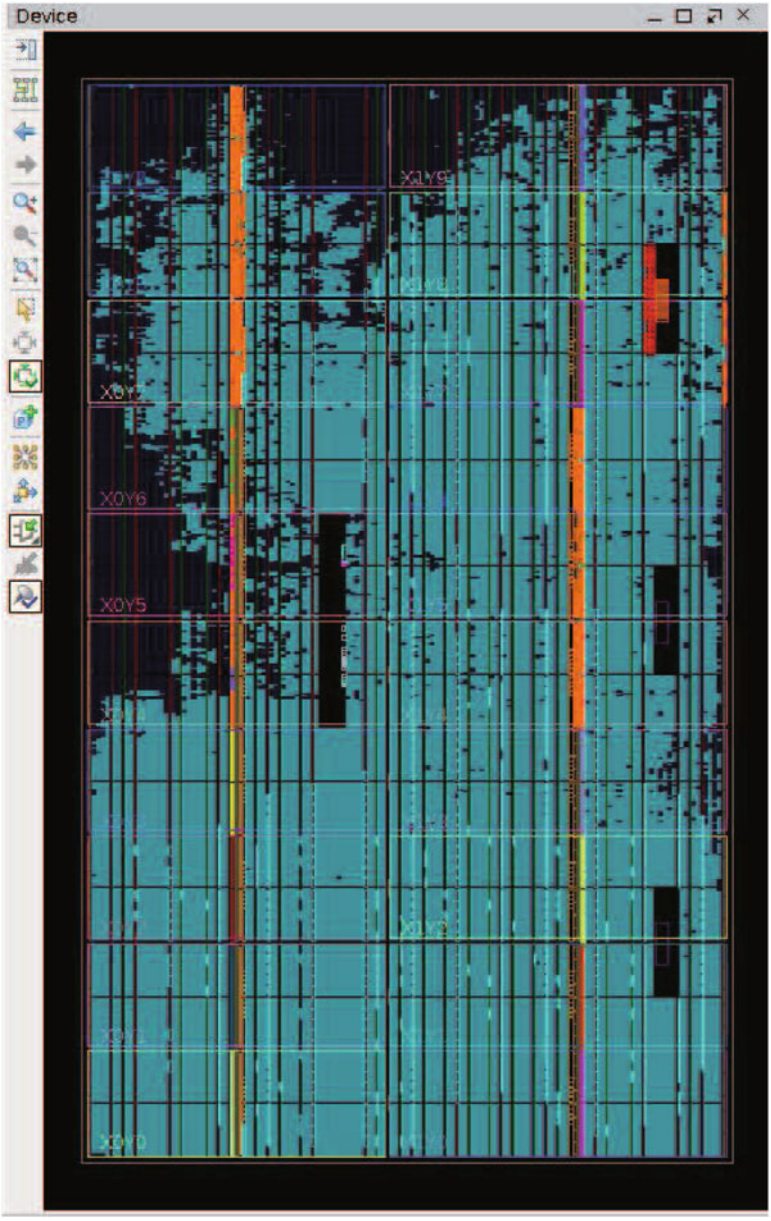

Fig. 3.13 Layout view of the FPGA with least-squares machine learning accelerator implemented

Table 3.5 UCI Dataset Specification and Accuracy

Benchmarks	Data size	Dim.	Class	Node No.	Acc. (%)
Car	1728	6	4	256	90.90
Wine	178	13	3	1024	93.20
Dermatology	366	34	6	256	85.80
Zoo	101	16	7	256	90.00
Musk1	476	166	2	256	69.70
Conn. Bench	208	60	2	256	70

3.6.4 Performance for Data Classification

In this experiment, six datasets are trained and tested from UCI dataset [22], which are wine, car, dermatology, zoo, musk and Connectionist Bench (Sonar, Mines vs. Rocks). The details of each dataset are summarized in Table 3.5. The architecture is set according to the training data set size and dimensions to demonstrate the parameterized architecture. For example, $N = 128(74)$ represents that the architecture parameter (training size) is 128 with the actual dataset wine size of 74. The accuracy of the machine learning is the same comparing to Matlab result since the single floating-point data format is applied for the proposed architecture.

For speed-up comparison, our architecture will not only compare to the time consumed by least-squares solver (DS) training method, but also SVM [23] and BP based method [24] on CPUs. For example, in dataset dermatology, the speed-up of training time is lower comparing to CPU based solution when the parallelism is 2. This is mainly due to the high clock speed of CPU. When the parallelism increases to 16, 4.70× speed-up can be achieved. For connectionist bench dataset, the speed-up of proposed accelerator is as high as 24.86×, when compared to the least-squares solver software solution on CPUs (Table 3.6). Furthermore, 801.20× and 25.55× speed-up can be achieved comparing to BP and SVM on CPUs.

3.6.5 Performance for Load Forecasting

Figure 3.14 shows the residential load forecasting with FPGA and CPU implementation. Clearly, all the peaks period are captured. It also shows that approximation by number representation (fixed point) will not degrade the overall performance. To quantize the load forecasting performance, we use two metrics: root mean square error (RMSE) and mean absolute percentage error (MAPE). Table 3.7 is the summarized performance with comparison of SVM. We can observe that our proposed accelerator has almost the same performance as CPU implementation. It also shows an average of 31.85% and 15.4% improvement in average on MAPE and RMSE comparing to SVM based load forecasting (Table 3.7).

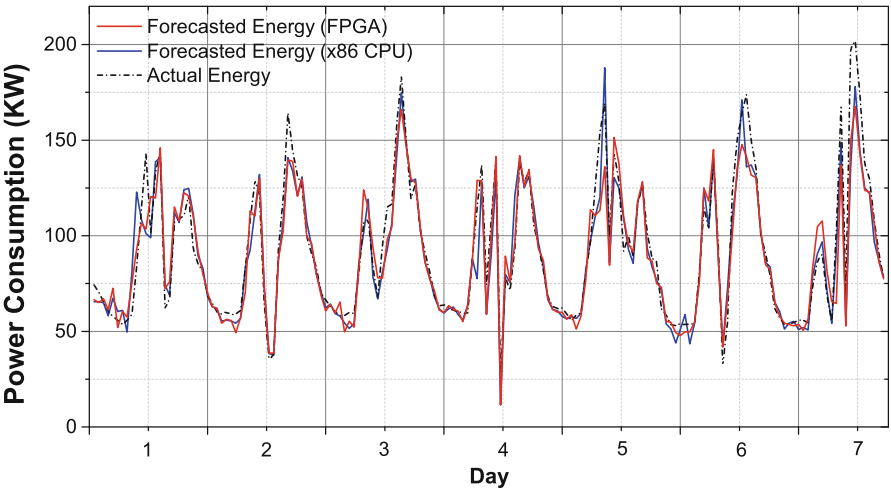

Fig. 3.14 7-Day residential load forecasting by proposed architecture with comparison of CPU implementation

Table 3.6 Parameterized and scalable architecture on different dataset with speed-up comparison to CPU

Benchmarks	FPGA (ms)	CPU (ms)	BP (ms)	SVM (ms)	Imp. (CPU)
Car	44.3	370	36,980	1182	8.35×
Wine	207.12	360	11,240	390	1.74×
Dermatology	19.45	160	17,450	400	8.23×
Zoo	22.21	360	5970	400	16.21×
Musk	24.09	180	340,690	3113	7.47×
Conn. Bench	14.48	360	11,630	371	24.86×

Table 3.7 Load forecasting accuracy comparison and accuracy improvement comparing to FPGA results to SVM result

Machine Learning	MAPE			RMSE		
	Max	Min	Avg	Max	Min	Avg
NN FPGA	0.12	0.072	0.92	18.87	6.57	12.80
NN CPU	0.11	0.061	0.084	17.77	8.05	12.85
SVM	0.198	0.092	0.135	20.91	5.79	15.13
Imp. (CPU)(%)	4.28	−18.03	−9.52	−6.19	18.39	0.39
Imp. (SVM)(%)	41.01	21.74	31.85	9.76	−13.47	15.40

3.6.6 Performance Comparisons with Other Platforms

In the experiment, the maximum throughput of proposed architecture is 12.68 Gflops with 128 parallelism for matrix multiplication. This is slower than GPU based implementation 59.78 Gflops but higher than x86 CPU based implementation 5.38 Gflops.

Table 3.8 Proposed architecture performance in comparison with other computation platform

Platform	Type	Format	Time (ms)	Power (W)	Energy	Speed-up	E. Imp.
x86 CPU	Train	Single	1646	84	138.26 J	2.59×	256.0×
	Test		1.54	84	0.129 J	4.56×	450.2×
ARM CPU	Train	Single	32,550	2.5	81.38 J	51.22×	150.7×
	Test		30.1	2.5	0.0753 J	89.05×	261.9×
GPU	Train	Single	10.99	145	1.594 J	0.017×	2.95×
	Test		0.196	145	0.0284 J	0.580×	98.92×
FPGA	Train	Single+ Fixed	635.4	0.85	0.540 J	–	–
	Test		0.338	0.85	0.287 mJ	–	–

To evaluate the energy consumptions, we calculate the energy for a given implementation by multiplying the peak power consumption of corresponding device. Although this is pessimistic analysis, it is still very likely to reach due to intensive memory and computation operations. Table 3.8 provides detailed comparisons between different platforms. Our proposed accelerator on FPGA has the lowest power consumption ($0.85W$) comparing to GPU implementation ($145W$), ARM CPU ($2.5W$) and x86 CPU implementation ($84W$). For training process, although GPU is the fastest implementation, our accelerator still has 2.59× and 51.22× speed-up for training comparing to x86 CPU and ARM CPU implementations. Furthermore, our proposed method shows 256.0×, 150.7×, and 2.95× energy saving comparing to CPU, ARM CPU, and GPU based implementations for training model. For testing process, it is mainly on matrix–vector multiplications. Therefore, GPU based implementations provide better speed-up performance. However, our proposed method still has and 4.56× and 89.05× speed-up for testing comparing to x86 CPU and ARM CPU implementations. Moreover, our accelerator is the most low-power platform with 450.1×, 261.9× and 98.92× energy saving comparing to x86 CPU, ARM CPU and GPU based implementations. In summary, our proposed accelerator provides a low-power and fast machine learning platform for smart-grid data analytics.

3.7 Conclusion

This chapter presents a fast machine learning accelerator for real-time data analytics in smart micro-grid of buildings with consideration of occupants behavior. An incremental and square-root-free Cholesky factorization algorithm is introduced with FPGA realization for training acceleration when analyzing the real-time sensed data. Experimental results have shown that our proposed accelerator on Xilinx Virtex-7 has a comparable forecasting accuracy with an average speed-up of 4.56× and 89.05×, when compared to x86 CPU and ARM CPU for testing. Moreover, 450.2×, 261.9×, and 98.92× energy saving can be achieved comparing to x86 CPU, ARM CPU, and GPU.

Acknowledgements This work is sponsored by grants from Singapore MOE Tier-2 (MOE2015-T2-2-013), NRF-ENIC-SERTD-SMES-NTUJTCI3C-2016 (WP4) and NRF-ENIC-SERTD-SMES-NTUJTCI3C-2016 (WP5).

References

1. L.D. Harvey, *Energy and the New Reality 1: Energy Efficiency and the Demand for Energy Services* (Routledge, London, 2010)
2. H. Ziekow, C. Goebel, J. Strüker, H.-A. Jacobsen, The potential of smart home sensors in forecasting household electricity demand, in *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)* (IEEE, New York, 2013), pp. 229–234
3. H.S. Hippert, C.E. Pedreira, R.C. Souza, Neural networks for short-term load forecasting: a review and evaluation. IEEE Trans. Power Systems **16**(1), 44–55 (2001)
4. W. Mielczarski, G. Michalik, M. Widjaja, Bidding strategies in electricity markets, in *Power Industry Computer Applications, 1999. PICA'99. Proceedings of the 21st 1999 IEEE International Conference* (IEEE, New York, 1999), pp. 71–76
5. E.A. Feinberg, D. Genethliou, Load forecasting, in *Applied Mathematics for Restructured Electric Power Systems* (Springer, Berlin, 2005), pp. 269–285
6. C. Sandels, J. Widén, L. Nordström, Forecasting household consumer electricity load profiles with a combined physical and behavioral approach. Appl. Energy **131**, 267–278 (2014)
7. S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, D.D. Edwards, *Artificial Intelligence: A Modern Approach*, vol. 2 (Prentice Hall, Upper Saddle River, NJ, 2003)
8. S. Theodoridis, K. Koutroumbas, Pattern recognition and neural networks, in *Machine Learning and Its Applications* (Springer, Berlin, 2001), pp. 169–195
9. S. Li, P. Wang, L. Goel, Short-term load forecasting by wavelet transform and evolutionary extreme learning machine. Electr. Power Syst. Res. **122**, 96–103 (2015)
10. A. Ahmad, M. Hassan, M. Abdullah, H. Rahman, F. Hussin, H. Abdullah, R. Saidur, A review on applications of ann and svm for building electrical energy consumption forecasting. Renew. Sust. Energ. Rev. **33**, 102–109 (2014)
11. G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: theory and applications. Neurocomputing **70**(1), 489–501 (2006)
12. Y.-H. Pao, G.-H. Park, D.J. Sobajic, Backpropagation, part iv learning and generalization characteristics of the random vector functional-link net. Neurocomputing **6**(2), 163–180 (1994). [Online]. Available http://www.sciencedirect.com/science/article/pii/0925231294900531
13. M.D. Martino, S. Fanelli, M. Protasi, A new improved online algorithm for multi-decisional problems based on mlp-networks using a limited amount of information, in *Proceedings of 1993 International Joint Conference on Neural Networks, 1993. IJCNN '93-Nagoya*, vol. 1 (1993), pp. 617–620
14. I. Richardson, M. Thomson, D. Infield, A high-resolution domestic building occupancy model for energy demand simulations. Energy Buildings **40**(8), 1560–1566 (2008)
15. N.-Y. Liang, G.-B. Huang, P. Saratchandran, N. Sundararajan, A fast and accurate online sequential learning algorithm for feedforward networks. IEEE Trans. Neural Netw. **17**(6), 1411–1423 (2006)
16. Y. Pang, S. Wang, Y. Peng, X. Peng, N.J. Fraser, P.H. Leong, A microcoded kernel recursive least squares processor using fpga technology. ACM Trans. Reconfigurable Technol. Syst. **10**(1), 5 (2016)
17. L.N. Trefethen, D. Bau III, *Numerical Linear Algebra*, vol. 50 (SIAM, Philadelphia, 1997)
18. A. Krishnamoorthy, D. Menon, Matrix inversion using cholesky decomposition. Preprint (2011). arXiv:1111.4144

19. F. Ren, D. Marković, A configurable 12237 kS/s 12.8 mw sparse-approximation engine for mobile data aggregation of compressively sampled physiological signals. IEEE J. Solid State Circuits **51**(1), 68–78 (2016)
20. Adm-pcie-7v3 [Online]. Available http://www.alpha-data.com/dcp/products.php?product=adm-pcie-7v3 (2016)
21. Beagleboard-xm [Online]. Available http://beagleboard.org/beagleboard-xm (2015)
22. M. Lichman, UCI machine learning repository (2013). [Online]. Available http://archive.ics.uci.edu/ml
23. J.A. Suykens, T. Van Gestel, J. De Brabanter, B. De Moor, J. Vandewalle, J. Suykens, T. Van Gestel, *Least Squares Support Vector Machines*, vol. 4 (World Scientific, Singapore, 2002)
24. R. Hecht-Nielsen, Theory of the backpropagation neural network, in *International Joint Conference on Neural Networks, 1989. IJCNN* (IEEE, New York, 1989), pp. 593–605