Speedup of Implementing Fuzzy Neural Networks With High-Dimensional Inputs Through Parallel Processing on Graphic Processing Units

Chia-Feng Juang, *Senior Member, IEEE*, Teng-Chang Chen, and Wei-Yuan Cheng

***Abstract*—This paper proposes the implementation of a zero-order Takagi–Sugeno–Kang (TSK)-type fuzzy neural network (FNN) on graphic processing units (GPUs) to reduce training time. The software platform that this study uses is the compute unified device architecture (CUDA). The implemented FNN uses structure and parameter learning in a self-constructing neural fuzzy inference network because of its admirable learning performance. FNN training is conventionally implemented on a single-threaded CPU, where each input variable and fuzzy rule is serially processed. This type of training is time consuming, especially for a high-dimensional FNN that consists of a large number of rules. The GPU is capable of running a large number of threads in parallel. In a GPU-implemented FNN (GPU-FNN), blocks of threads are partitioned according to parallel and independent properties of fuzzy rules. Large sets of input data are mapped to parallel threads in each block. For memory management, this research suitably divides the datasets in the GPU-FNN into smaller chunks according to fuzzy rule structures to share on-chip memory among multiple thread processors. This study applies the GPU-FNN to different problems to verify its efficiency. The results show that to train an FNN with GPU implementation achieves a speedup of more than 30 times that of CPU implementation for problems with high-dimensional attributes.**

***Index Terms*—Classification, compute unified device architecture (CUDA), fuzzy neural networks (FNNs), graphic processing unit (GPU), neural fuzzy systems.**

I. Introduction

RESEARCHERS have successfully applied fuzzy neural networks (FNNs) in several areas, such as classification and regression. The fuzzy IF–THEN rule derivation is often difficult and time consuming, and it requires expert knowledge. FNNs provide a solution to address the common bottleneck in fuzzy system design. Many researchers have proposed FNNs with parameter and/or structure learning [1]–[15]. In [1] an adaptive-network-based fuzzy inference system (ANFIS) is proposed. The ANFIS is learned through parameter learning. The structure of the ANFIS is fixed, and the input space is partitioned in a grid type. This kind of partition faces the curse of dimensionality because the number of fuzzy rules increases exponentially as the dimension of the input space increases. To address this problem, many FNNs with offline [3] or online [2], [4]–[15] structure-learning ability have been proposed. These structure-learning algorithms determine the number of rules using the idea of clustering. The objective is to design a well-performed FNN with the least number of rules. However, the problems considered in the FNNs [3]–[12], [15] are regression problems, where the network input dimensions are smaller than 10. For these low-dimensional regression problems, an FNN with a small rule set may achieve satisfactory performance. For many problems, such as pattern classification, tens or hundreds of attributes are fed as inputs to an FNN. These high-dimensional classification problems usually require an FNN with a large rule set to achieve good performance. Even an FNN with a small rule set may achieve good performance for some of these high-dimensional problems, and the user may need to try a larger rule set to see whether or not much better performance can be achieved. To train an FNN with a large rule set for high-dimensional problems is a computationally intensive task. The aforementioned FNNs are implemented on a single-threaded CPU, where datasets are processed in series instead of in parallel. For some problems, FNN training on a CPU may take several days. FNNs are suitable for implementation on parallel processing units because they can be expressed as data-parallel computations due to the parallel processing property of fuzzy rules and input variables. This property motivates the proposal to implement FNN using graphic processing units (GPUs) [16]. The GPU-implemented FNN (GPU-FNN) that are considered in this paper uses the structure and parameter learning in a self-constructing neural fuzzy inference network (SONFIN) [2]. Like the FNNs in studies [3]–[12] and [15], the structure and parameter learning algorithms in the SONFIN were proposed to design a well-performed fuzzy system with the least number of rules. This study uses SONFIN learning because of its powerful learning ability with a simple structure-learning approach. As mentioned earlier, for the problems with high-dimensional inputs, a large number of rules may be inevitable to achieve good performance even with network structure/parameter learning. The GPU-FNN is proposed to address this computationally intensive problem.

The GPU is a many-core multithreaded multiprocessor originally developed as a configurable graphics processor but recently used as a programmable parallel processor [16]. The

Manuscript received May 24, 2010; revised October 19, 2010 and January 17, 2011; accepted March 21, 2011. Date of publication April 7, 2011; date of current version August 8, 2011. This work was supported in part by the Ministry of Education, Taiwan, under the Aiming for Top University plan and in part by the National Science Council, Taiwan, under Grant NSC-98-2221-E-005-041-MY2.

The authors are with the Department of Electrical Engineering, National Chung-Hsing University, Taichung 402, Taiwan (e-mail: cfjuang@dragon.nchu.edu.tw; vilinchen@gmail.com; D9664206@mail.nchu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TFUZZ.2011.2140326

1063-6706/$26.00 © 2011 IEEE

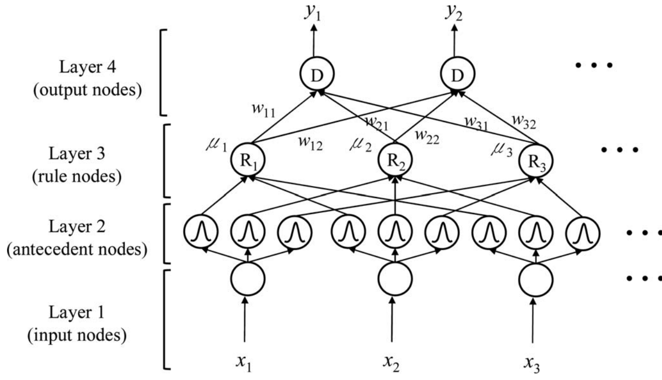


Fig. 1. Structure of the GPU-FNN.

GPU executes multiple threads in parallel that makes possible the parallel processing of enormous datasets and arithmetic operations in only one PC. GPU computing is becoming one of the mainstream computing systems [17]. The GPU is especially well suited to solve computationally intensive problems that can be expressed as data-parallel or task-parallel computations. Researchers have successfully applied GPU computing to problems that are traditionally addressed by the CPU [16], [18]–[21], with the GPU performance outpacing its CPU counterpart. Though the GPU provides a platform for parallel processing, performance of a problem that runs on it still depends heavily on how the developer manages blocks of thresholds and datasets. The proposed GPU-FNN uses a fuzzy rule-based approach for block–thread and dataset management. This approach efficiently partitions the GPU-FNN operations into several blocks of threads and the GPU-FNN datasets into chunks for sharing among threads. The software for the implementation of the GPU-FNN is the compute unified device architecture (CUDA) [22] that is developed by NVIDIA [23]. The CUDA makes it possible for developers to implement parallel algorithms on NVIDIA's GPU with high-level language. This paper verifies learning speed improvement of the GPU-FNN through comparisons with its CPU counterpart by the usage of four high-dimensional classification problems.

This paper is organized as follows. Section II describes the structure and learning of the GPU-FNN. Section III describes the basic concept of GPU hardware and CUDA software. Section IV introduces the block-thread and data management in the GPU-FNN by the use of the CUDA. Section V applies the GPU-FNN to solve different problems and demonstrates the learning-time speedup in comparison with its CPU counterpart. Section VI presents discussions on the performance of the GPU-FNN. Finally, Section VII presents the conclusion.

II. Graphic-Processing-Unit-Implemented Fuzzy Neural Network Structure and Learning

A. Structure of a Graphic-Processing-Unit-Implemented Fuzzy Neural Network

This section describes the GPU-FNN structure. The zero-order Takagi–Sugeno–Kang (TSK)-type GPU-FNN consists of four layers, as shown in Fig. 1. Each rule in the GPU-FNN has the following form:

$$\text{Rule } k\text{: IF } x_1 \text{ is } A_{k1} \text{ And}, \ldots, \text{And } x_n \text{ is } A_{kn} \quad \text{THEN } y_l \text{ is } w_{kl}, \quad k = 1, \ldots, r \quad (1)$$

where A_{kj} is a fuzzy set, w_{kl} is a real number, and r is the total number of rules. The function of each layer is described as follows.

In Layer 1, each node corresponds to one input variable. The node first scales the range of each input variable, if necessary, and transmits scaled input values to the next layer.

In Layer 2, each node corresponds to one fuzzy set and calculates a membership value. In this layer, the fuzzy set A_{kj} is employed with the following Gaussian membership function:

$$M_{kj}(x_j) = \exp\left\{-\left(\frac{(x_j - m_{kj})^2}{\sigma_{kj}^2}\right)\right\} \quad (2)$$

where m_{kj} and σ_{kj} denote the center and width of the fuzzy set, respectively. The number of fuzzy sets in each input variable is equal to the number of fuzzy rules.

In Layer 3, each node represents a fuzzy logic rule and performs antecedent matching of this rule by the use of the following AND operation:

$$\mu_k(\vec{x}) = \prod_{j=1}^{n} M_{kj}(x_j) \quad (3)$$

where $\vec{x} = [x_1, \ldots, x_n]$. The number of nodes in this layer is equal to the number of rules r.

In Layer 4, the number of output nodes is equal to the number of output variables. Each node performs as a defuzzifier by the use of a weighted average operation. The consequent parameter w_{kl} functions as link weight. The defuzzified output can be written as follows:

$$y_l = \Phi \cdot \sum_{k=1}^{r} w_{kl}\mu_k, \quad \Phi = 1 \Big/ \sum_{i=1}^{r} \mu_i, \quad l = 1, \ldots, L \quad (4)$$

where L is the number of output variables.

B. Structure Learning

Initially, there are no rules in the GPU-FNN. All rules are constructed by the online structure learning that is used in the SONFIN [2]. The firing strength $\mu_k(\vec{x})$ in (3) is used as the criterion to judge if a new fuzzy rule will be generated. For the first incoming datum $\vec{x}(0)$, a new fuzzy rule is generated with the center and width of the Gaussian membership function assigned as follows:

$$m_{1j} = x_j(0), \quad j = 1, \ldots, n, \quad \text{and} \quad \sigma_{1j} = \sigma_{\text{init}} \quad (5)$$

where σ_{init}(=0.35 in this paper) is a prespecified value that determines the initial width of the first cluster. For the succeeding incoming data (t), find

$$K = \arg \max_{1 \le k \le r(t)} \mu_k(\vec{x}(t)) \quad (6)$$

where $r(t)$ is the number of existing rules at time t. If $\mu_k \le \mu_{th}$, a new rule is generated, where $\mu_{th} \in (0, 1)$ is a prespecified

threshold that decays with training iteration number. Once a new rule is generated, the next step is to assign the center and width of the corresponding membership function. Here, these values are assigned by

$$m_{(r(t)+1)j} = x_j(t) \tag{7}$$

$$\sigma_{(r(t)+1)j} = \beta \cdot \| \vec{x} - \vec{m}_K \|.^2 \tag{8}$$

where coefficient β (=0.5 in this paper) determines the overlapping between two rules in the input space. This paper sets β to be half the distance between membership function centers of the new rule and the Kth rule so that there is a suitable overlapping between the two neighboring rules.

For parameter learning, a gradient descent algorithm tunes all consequent and antecedent parameters. Details of the learning algorithm are described as follows. Let y_l^d be the lth desired output for an input datum. The objective function to be minimized is defined by

$$E = \frac{1}{2}\sum_{l=1}^{L}(y_l - y_l^d)^2. \tag{9}$$

The antecedent and consequent parameters are updated by

$$w_{kl}(t+1) = w_{kl}(t) - \eta\frac{\partial E}{\partial w_{kl}}, \frac{\partial E}{\partial w_{kl}} = \left(y_l - y_l^d\right)\cdot \Phi \cdot \mu_k \tag{10}$$

$$\begin{aligned} m_{kj}(t+1) &= m_{kj}(t) - \eta\frac{\partial E}{\partial m_{kj}}, \frac{\partial E}{\partial m_{kj}} \\ &= \sum_{l=1}^{L}[(y_l - y_l^d)\cdot(w_{kl} - y_l)]\cdot\Phi\cdot\mu_k \\ &\quad \cdot \frac{2(x_j - m_{kj})}{(\sigma_{kj})^2} \end{aligned} \tag{11}$$

$$\begin{aligned} \sigma_{kj}(t+1) &= \sigma_{kj}(t) - \eta\frac{\partial E}{\partial \sigma_{kj}}, \frac{\partial E}{\partial \sigma_{kj}} \\ &= \sum_{l=1}^{L}[(y_l - y_l^d)\cdot(w_{kl} - y_l)]\cdot\Phi\cdot\mu_k \\ &\quad \cdot \frac{2(x_j - m_{kj})^2}{(\sigma_{kj})^3} \end{aligned} \tag{12}$$

where η is a learning constant that controls the converging speed of the gradient decent algorithm.

III. Basic Concepts of Graphic Processing Unit and Compute Unified Device Architecture

Recently, the GPU has been transformed into the general-purpose GPU. The programmable GPU has evolved into a high parallel, multithreaded, multicore processor with huge computational power and memory bandwidth. This paper uses a GPU-accelerated solver for FNN implementation. The GPU used is the NVIDIA Tesla C1060 computing processor, which is an extended computing card, connected to a PC by means of a PCI-Express interface. The NVIDIA Tesla C1060 has a total of 4 GB of dedicated memory with a 102 GB/s peak memory bandwidth. This architecture can satisfy the requirements of massive data-parallel operations. Fig. 2 shows the hardware of the Tesla C1060. Thirty streaming multiprocessors (SMs) make up the Tesla C1060. Each SM is a single-instruction-multiple-data (SIMD) processor and is composed of eight streaming processors (SPs). Each SM has 16 KB of the on-chip shared memory. The shared memory is a type of high-speed memory, and data can be read (or written) by all eight SPs in an SM. The Tesla C1060 has read-only texture memory (cache) and constant spaces. Textures can be used to avoid uncoalesced loads from global memory, thereby improving performance.

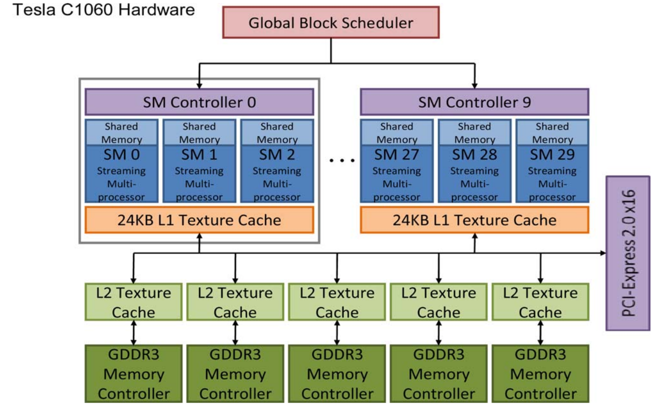


Fig. 2. Block diagram of the NVIDIA Tesla C1060 computing processor hardware.

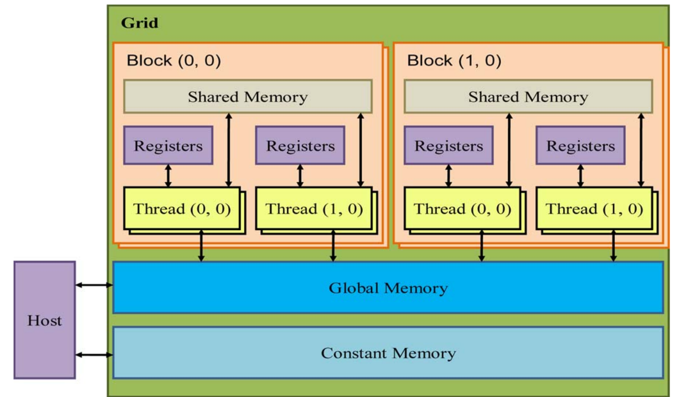


Fig. 3. CUDA programming model.

Fig. 3 shows the CUDA programming model. The model is a set of massive threads that run in parallel. A thread block is a number of SIMD threads that work on an SM at a given time, can exchange information through the shared memory, and can be synchronized. The operations are systematized as a grid of thread blocks. For operation parallelism, the programming model allows a developer to partition a program into several subproblems, each of which is executed independently on a block. Each subprogram can be further divided into finer pieces that perform the same function for execution on different threads within the block. For dataset parallelism, datasets can be divided into smaller chunks that are stored in the shared memory, and

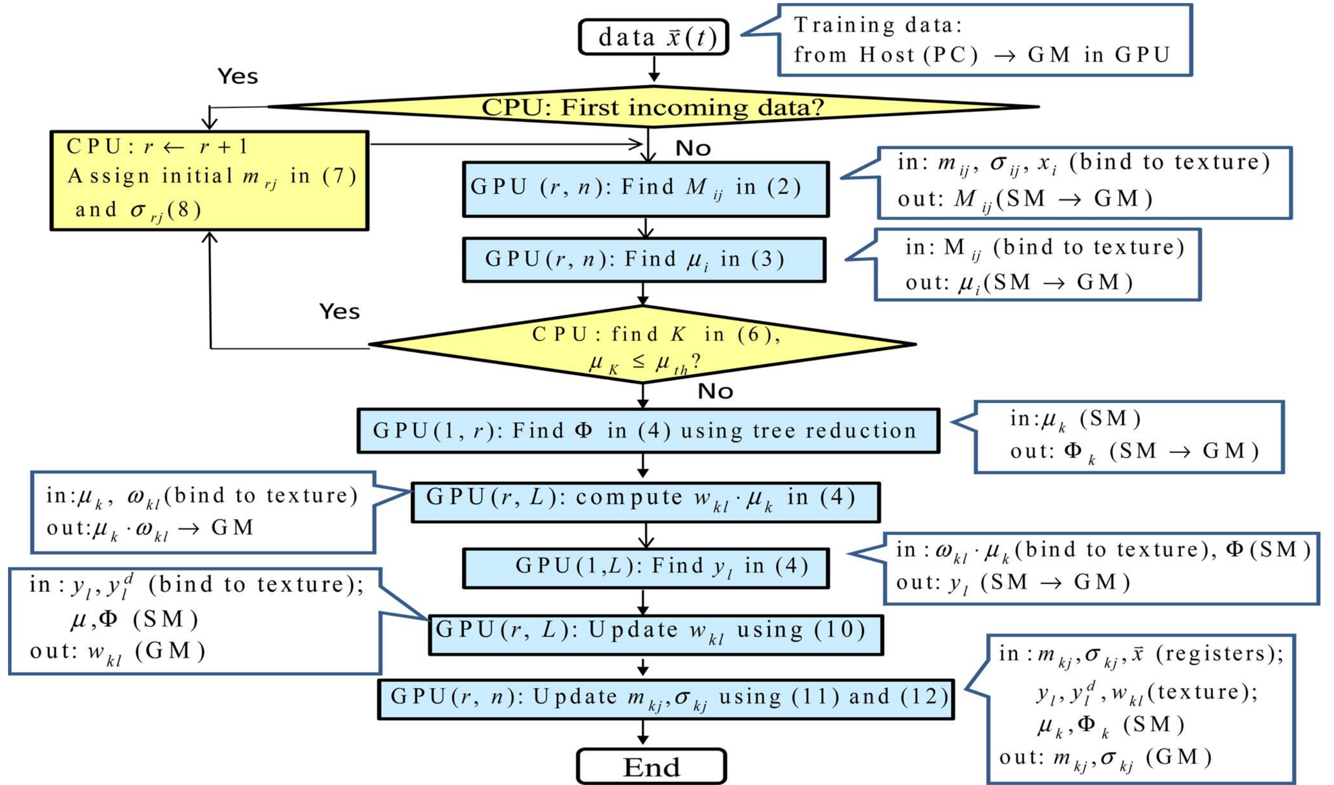


Fig. 4. Flow chart of GPU-FNN implementation for each incoming data (t), where GPU(a,b) means that there are a blocks and b threads per block to implement a function, and SM and GM represent the shared memory and global memory, respectively.

each chunk is visible to all threads of the same block. This local data arrangement approach reduces the need to access off-chip global memory, which reduces data access time.

In GPU-FNN, blocks of threads are partitioned based on fuzzy rules so the number of blocks is equal to the number of fuzzy rules. This partition method makes good use of the parallel and independent properties of fuzzy rules. The partition method also considers the ease of the block number scalability when the rule number changes in the structure-learning process of the GPU-FNN. For memory management in the GPU-FNN, datasets are also divided into chunks according to the fuzzy rules. By this means, datasets used in the same fuzzy rule are fast read (with coalescing data access) from global memory and are allocated to the same shared memory. As introduced in Section IV-A, these rule-relevant datasets are stored in the concatenated shared memory to make possible the use of tree-reduction techniques for membership value multiplication and firing strength summation. The GPU-FNN enables the function of texture memory fetching. The proposed memory management approach reduces data access time.

IV. Graphic-Processing-Unit-Implemented Fuzzy Neural Network Using the Compute Unified Device Architecture

The GPU-FNN network output functions in (2)–(4) and the parameter learning functions in (10)–(12) are parallel. Therefore, these functions are implemented by the use of GPUs, as introduced in the following sections. The structure-learning functions in (5)–(8) are simple and do not possess much parallelism. Therefore, these functions are implemented in the CPU. Fig. 4 shows the flowchart of the GPU-FNN implementation, where the number of blocks and the number of threads per block for each network function, and the input–output memory access of each thread is also indicated in the flowchart. Fig. 5 shows the timing of each GPU-FNN function for an input data $\vec{x}(t)$. For clarity, it is assumed that no rules are generated (which holds for most input data) after structure learning in this figure. If a new rule is generated, then additional GPU time of $T_1 + T_2$ is necessary for the computation of membership functions and firing strength of the new rule. The implementation idea in Fig. 5 is that the same mathematical function with the only difference in rule, input, or output indexes is implemented in parallel by the use of blocks of threads. These functions mainly include the membership function in (2), rule strength in (3), output computation in (4), consequent parameter update in (10), and antecedent parameter update in (11) and (12). For the GPU-FNN with high-dimensional inputs, the number of functions that can be implemented in parallel is large. For example, there are 500 membership functions that can be implemented in parallel for a GPU-FNN with 10 rules and 50 inputs. The maximum number of parallel processing thresholds in a block is relatively too small to implement these functions. Therefore, this paper implements GPU-FNN by the usage of blocks of thresholds. The number of

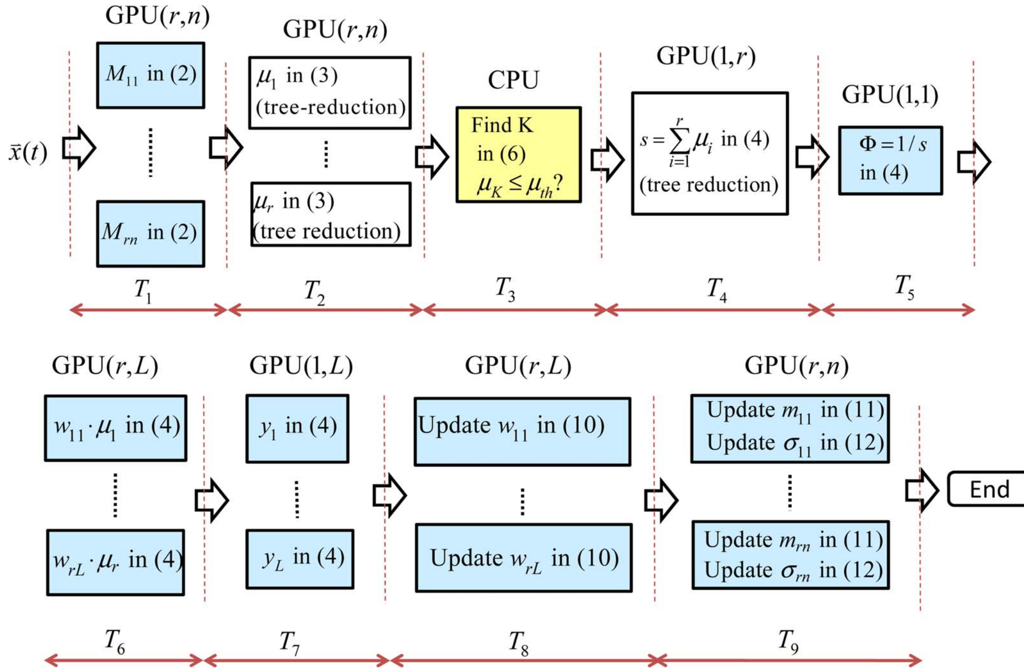


Fig. 5. Timings of the functions implemented by the use of the GPU-FNN, where the white block and shaded block in the GPU represent a "block" and "thread," respectively.

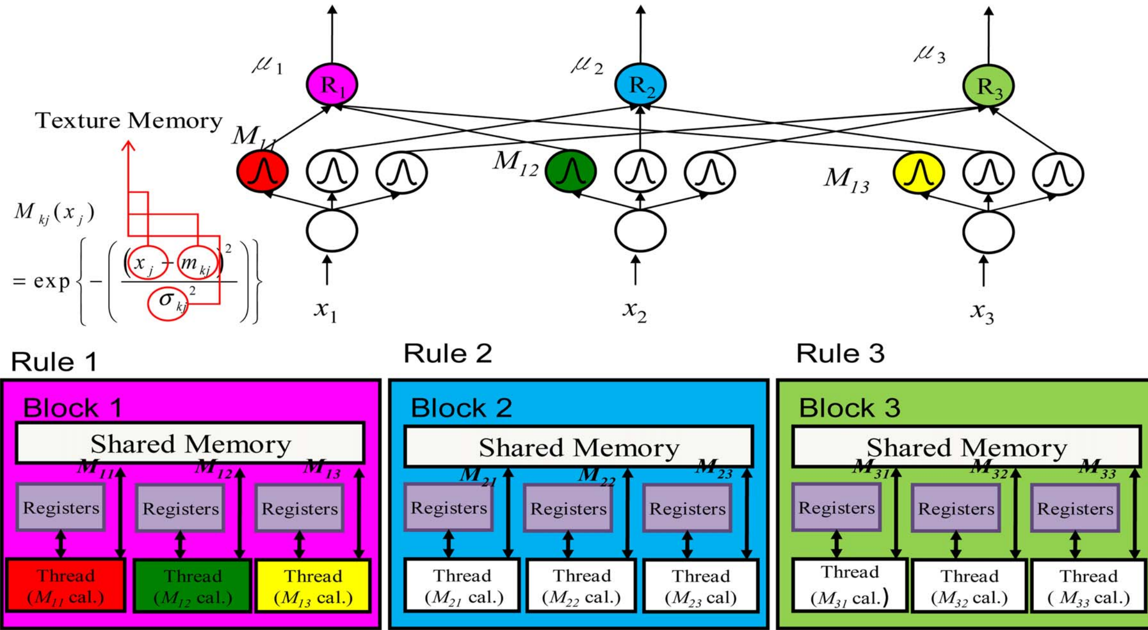


Fig. 6. Block-thread partition to calculate the membership value M_{kj} when the rule number is $r = 3$, and the input variable number is $n = 3$.

blocks is equal to the number of rules as explained in Section III. The following details the implementation approach.

A. Rule Firing Strength Calculation

Fig. 6 shows the partition of the membership function calculation in (2) into blocks of threads. The number of blocks (grid dimension) is equal to the number of fuzzy rules r. Block k is responsible for the calculation of the n membership values M_{kj}, $j = 1, \ldots, n$, for rule k. The number of threads (block dimension) in each block is equal to the number of input variables n. Each thread does the same function of membership value computation, with the only difference being input data values, and puts the computed result into the shared memory. In each block, a tree-reduction technique is used to implement the product of all of the membership values [see (3)] stored in the shared memory. Fig. 7 shows the tree-reduction technique for multiplication implementation. The concatenated data $s[0], \ldots, s[2^q - 1]$

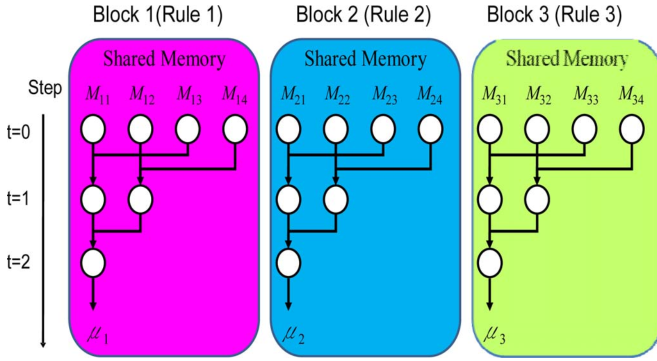


Fig. 7. Tree-reduction multiplication technique to calculate the membership product to obtain the rule firing strength.

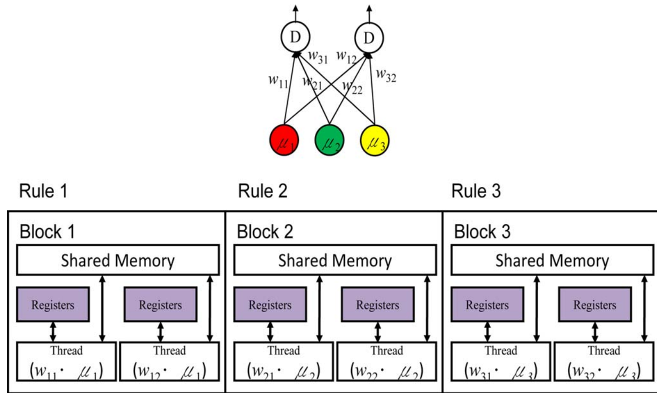


Fig. 8. Block-thread partition to compute the product $w_{kl} \cdot \mu_k$, when the rule number is $r = 3$, and the output variable number is $L = 2$.

($q = \lfloor \log_2 n \rfloor$) in the shared memory are divided into two parts, $s[0], \ldots, s[2^{q-1} - 1]$ and $s[2^{q-1}], \ldots, s[2^q - 1]$, where $s[i] = 1$ for $n \leq i \leq 2^q - 1$. Fig. 7 shows the case when $q = 2$. In the first stage, a total of 2^{q-1} multiplications, $s[0] \cdot s[2^{q-1}]$, $s[1] \cdot s[2^{q-1} + 1], \ldots,$ and $s[2^{q-1} - 1] \cdot s[2^q - 1]$ are executed in parallel by threads. The pairwise parallel multiplication approach continues for q times after which all data are multiplied and the final result (μ_i) is stored in$s[0]$. Without the use of the tree-reduction operation, the $n - 1$ multiplications for the computation of μ_i in each block takes $n - 1$ multiplication times while the tree reduction takes only $q$multiplication times. Take the case when $n = 100$ as an example. It takes 10 and 99 multiplication times for implementations with and without the use of the tree-reduction technique, respectively. The reduction in time is much significant for an FNN with a higher number of input dimensions as considered in this paper.

B. Network Output Calculation

One block with multiple threads by the use of the tree-reduction technique in Section IV-A (with multiplication replaced by summation) is applied to find the firing strength summation $\sum_{k=1}^{r} \mu_k$ in (4). The reciprocal of the summation value Φ is calculated and stored in the shared memory.

Fig. 8 shows the block-thread partition for computing $w_{kl} \cdot \mu_k$ in (4). The number of blocks is equal to the number of rules r, and the number of threads per block is equal to the number of output variables L. Each thread in the same block k computes the same function of multiplication $w_{kl} \cdot \mu_k$, with μ_k read from the shared memory. Then, the GPU-FNN uses one block with L threads to compute y_l. In this block, thread l, first, computes the summation of all $w_{kl} \cdot \mu_k$ (i.e., $\sum_{k=1}^{r} w_{kl}\mu_k$) and, then, multiplies the summation value with Φ to get the network output y_l.

C. Consequent Parameter Learning

Fig. 9 shows the block–thread partition for consequent parameter learning in (10). The number of blocks is equal to r and the number of threads in each block is equal to L. Each thread computes the same function (10) with the only difference being input data values. The lth thread in the kth block computes the update of w_{kl}. Data Φ and μ_k are stored in the shared memory of each block because each thread of the same block uses the same Φ and μ_k values, which reduces the time to access data from global memory.

D. Antecedent Parameter Learning

Fig. 10 shows the block–thread partition for antecedent parameter learning in (11) and (12). The number of blocks is equal to r and the number of threads in each block is equal to n. Each thread computes the same parameter update functions in (11) and (12) with the only difference being input data values. Like consequent parameter tuning, Φ and μ_k are read from the shared memory in each block. In addition, the repeated used data x_j, m_{kj}, and σ_{kj} are prestored in registers to reduce access time.

V. Experiments

The GPU-FNN was implemented on a PC with a dual 2.66 GHz Intel CPU Nehalem X5550 with 24 GB RAM and one NVIDIA Tesla C1060 computing card with NVIDIA driver version 8.15.11.9038 and CUDA 2.3, running Linux CentOS 5.4[1]. The GPU-FNN was written in C++ by the use of NVIDIA's CUDA. Implementation of the same FNN by the use of the CPU was conducted on the same computer for learning time comparison. The CPU and GPU codes were implemented without the use of OpenMP, message passing interface (MPI), or any other multitask library. Four classification problems and one regression problem were solved by the GPU-FNN. Table I shows statistics of datasets (which include attribute format, the numbers of attributes, classes, training, and test data) in Examples 1–5. It should be emphasized that in these examples the major concern is the speed of the GPU-FNN instead of the classification performance. Therefore, this paper does not do any cross validation in the classification problems.

Example 1 (DNA Dataset 1). The DNA dataset is from the Statlog collection [26]. This dataset consists of 3186 samples in which 2000 samples are used for training and the left 1186 samples are used for testing. Each sample consists of 180 attributes, each of which takes the binary value 1 or 0. The dataset has three classes in total. This experiment set the learning rate η to 0.02. The same learning rate was also used in the following

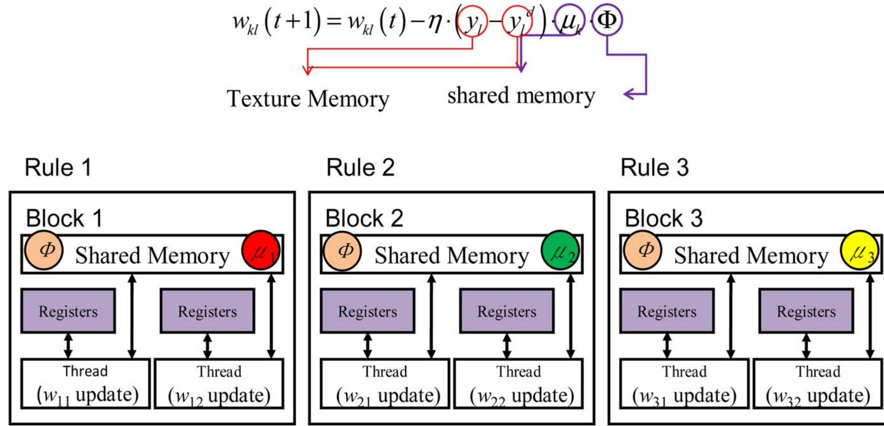


Fig. 9. Block-thread partition to update the consequent parameter w_{kl} when the rule number is $r = 3$, and the output variable number is $L = 2$.

TABLE I
STATISTICS OF DATASETS IN EXAMPLES 1–5 USED FOR GPU-FNN EVALUATION

Classifier	Example 1	Example 2	Example 3	Example 4	Example 5
Classification/Regression	3 classes	2 classes	10 classes	2 classes	regression
# Attributes	180	60	256	86	2
Attribute format	binary	real	real	real	real
#Training data	2,000	1,000	7,291	5,822	50,000
#Test data	1,186	2,175	2,007	4,000	200

TABLE II
GPU-FNN CLASSIFICATION PERFORMANCE FOR DIFFERENT NUMBERS OF RULES AND THE TRAINING TIME AND TEST ERROR RATE COMPARISON BETWEEN GPU AND CPU IN EXAMPLE 1

Rule numbers		3	7	15	137
Threshold μ_{th}		0.01	0.014	0.0155	0.0234397
GPU Training error rate		26%	0.15%	0.15%	0.05%
Testing error rate	CPU	31%	5.14%	4.13%	5.31%
	GPU	31%	5.14%	4.13%	5.31%
Training time (sec)	CPU	4,809.0	9,409.6	16,042.5	128,667.1
	GPU	2,771.9	2,677.4	2,741.0	3,489.1
Time Speed up		1.73 X	3.51 X	5.85 X	36.88 X

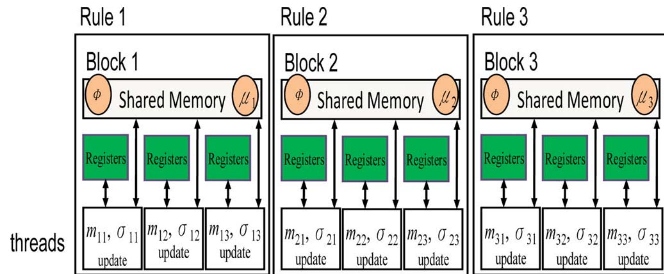


Fig. 10. Block-thread partition to update the antecedent parameters m_{kj} and σ_{kj} when rule number $r = 3$ and input variable number $n = 3$.

three classification examples. The number of iterations was set to 5000. Table II shows the training time, and training and testing results of the GPU-FNN for different thresholds μ_{th}. Table II shows that a larger value of μ_{th} generates a larger number of rules and smaller values of training error rates in general. When the number of rules is large, overtraining occurs and the test error rate increases. Table II also shows the training time and test error rates of CPU implementation for these different numbers of rules. The test error rates of the GPU and CPU implementations are the same. The results show that the GPU implementation significantly reduces training time compared with CPU implementation, especially when the rule number is large. A speedup of 36 times is achieved when the rule number is 137. The classification performance of the GPU-FNN was compared with reported results of different classifiers [24]. Table III shows the classifiers used for comparison, which include FNN [25], radial-basis-function-kernel-based support vector machine (SVM) [27], reduced SVM (RSVM) [28], and support-vector-based FNN (SVFNN) [24]. The results indicate that the GPU-FNN achieves a lower test error rate than the classifiers used for comparison with a much smaller model size.

Example 2 (DNA Dataset 2). The DNA dataset is from the University of California-Irvine (UCI) repository

TABLE III
COMPARISONS OF DIFFERENT CLASSIFIERS FOR THE DNA DATASET IN EXAMPLE 1

Classifier	FNN [25]	RBF-kernel-based SVM [27]	RSVM[28]	SVFNN[24]	GPU-FNN
Structure	Not available	1152×3 support vectors	372×3 support vectors	334×3 rules	15 rules
Test error rate	16.64%	4.21%	7.7%	5.8%	4.13%

TABLE IV
GPU-FNN CLASSIFICATION PERFORMANCE FOR DIFFERENT NUMBERS OF RULES AND THE TRAINING TIME AND TEST ERROR RATE COMPARISON BETWEEN GPU AND CPU IN EXAMPLE 2

Rule number		5	34	139	427	715
Threshold μ_{th}		0.01	0.02	0.03	0.05	0.07
GPU Training error rate		15.50%	1.70%	2.20%	2.30%	2.40%
Testing error rate	CPU	17.885%	3.770%	3.954%	3.678%	8.690%
	GPU	17.885%	3.770%	3.954%	3.678%	8.690%
Training time (sec)	CPU	90,596.9	355,833.6	1,288,006.1	3,824,729.0	6,386,055.0
	GPU	134,516.5	143,047.5	162,192.7	225,526.9	363,435.4
Time Speed up		0.674 X	2.488 X	7.941 X	16.960 X	17.571 X

TABLE V
GPU-FNN CLASSIFICATION PERFORMANCE FOR DIFFERENT NUMBERS OF RULES AND THE TRAINING TIME AND TEST ERROR RATE COMPARISON BETWEEN GPU AND CPU IN EXAMPLE 3

Rule number		4	7	50	186	408
Threshold μ_{th}		0.01	0.03	0.05	0.07	0.09
GPU Training error rate		61.871%	35.071%	2.935%	2.387%	2.208%
Testing error rate	CPU	62.432%	37.917%	7.225%	6.577%	6.278%
	GPU	62.432%	37.917%	7.225%	6.577%	6.278%
Training time (sec)	CPU	3,940,575.3	5,762,113.0	31,055,524.0	111,716,120.0	248,085,472.0
	GPU	1,068,477.1	1,071,143.5	1,186,485.8	2,290,084.8	3,159,869.5
Time Speed up		3.69 X	5.38 X	26.17 X	48.78 X	78.51 X

(http://archive.ics.uci.edu/ml/). The task is to classify two types of splice junctions in DNA sequences: exon/intron (EI) or intron/exon (IE) sites. This dataset consists of 1000 training samples and 2175 test samples. Each sample consists of 60 attributes. The number of iterations was set to 500. Table IV shows the training time, training, and testing results of the GPU-FNN for different thresholds $\mu_{\rm th}$.When the number of rules is too small (such as five rules), the test error rate is high. As in Example 1, when the number of rules is large, overtraining occurs and the test error rate increases. Table IV also shows the training time and testing result of the CPU. As in Example 1, the test error rates of the GPU and CPU implementations are the same. Table IV shows that the GPU implementation, significantly, reduces training time compared with CPU implementation except when the rule number is very small. When the rule number is 5, the GPU implementation shows more training time than CPU implementation. Analysis of the break on GPU speedup is given in Section VI.

Example 3 (Handwritten ZiPcode Recognition). The task is to classify ten handwritten digits automatically scanned from the envelopes by the U.S. Postal Service. The dataset is from (http://www-stat.stanord.edu/ElemStatLearn) [29]. The dataset consists of 7291 samples for training and 2007 samples for testing. Each sample consists of 256 attributes, which represent the 256 pixel values in each 16×16 gray image. The number of training iterations was set to 500. Table V shows the performance of the GPU-FNN for different numbers of thresholds and rules. The performance is poor when the number of rules is smaller than 10. Table V also shows the training time and test result of CPU implementation. As in Examples 1 and 2, the test error rates of the GPU and CPU implementations are the same. Table V shows that the GPU implementation significantly reduces training time compared with CPU implementation.

Example 4 (Insurance Company Dataset). The dataset is from the UCI repository. The dataset consists of 5822 samples (5822 customer records) for training and 4000 samples for testing,

TABLE VI
GPU-FNN CLASSIFICATION PERFORMANCE FOR DIFFERENT NUMBERS OF RULES AND THE TRAINING TIME AND TEST ERROR RATE COMPARISON BETWEEN GPU AND CPU IN EXAMPLE 4

Rule number		3	79	194	326	527	808
Threshold μ_{th}		0.00064	0.02	0.03	0.03264	0.05	0.08
GPU Training error rate		12.384	6.23	0.052	0.017	0.000	0.000
Testing error rate	CPU	12.225%	10.525%	9.375%	9.375%	9.100%	9.350%
	GPU	12.225%	9.825%	9.475%	9.175%	9.275%	9.350%
Training time (sec)	CPU	7496.0	65,179.8	143,948.3	235,430.5	373,412.0	582,072.1
	GPU	7614.4	8,576.5	9,968.1	11,945.6	18,173.1	21,847.0
Time Speed up		0.984 X	7.60 X	14.44 X	19.71 X	20.55 X	26.64 X

TABLE VII
GPU-FNN IDENTIFICATION PERFORMANCE FOR DIFFERENT NUMBERS OF RULES AND THE TRAINING TIME AND TEST RMSE COMPARISON BETWEEN GPU AND CPU IN EXAMPLE 5

Rule number		5	9	26	31
Threshold μ_{th}		0.05	0.1	0.19	0.25
Testing RMSE	CPU	0.01835451	0.00791722	0.00569621	0.01072730
	GPU	0.01835450	0.00791719	0.00569627	0.01072733
Training time (sec)	CPU	0.143.	0.168	0.617	0.777
	GPU	13.504	13.621	13.676	13.794
Time Speed up		0.0106X	0.0123X	0.0451X	0.0563X

with each sample that consists of 86 attributes. The dataset has two classes in total. Each attribute is normalized in [0, 1]. The number of iterations was set to 5000. Table VI shows the performance of the GPU-FNN for different numbers of thresholds and rules. The GPU-FNN achieves a training error rate of 0% when the rule number is 527, and the test error rate is 9.275%. Table VI also shows the training time and test result of CPU implementation. As in Example 2, Table VI shows that GPU implementation significantly reduces the training time compared with CPU implementation except for a very small number of rules. When the rule number is 808, the CPU implementation takes more than 6 d for training. GPU implementation takes approximate 6 h. Different from the comparison results in Examples 1 to 3, Table VI shows that there is a small difference between the test error rates of GPU and CPU implementations for a large number of rules. In contrast with Example 1, the input attributes in the example are real numbers instead of binary numbers. In contrast with Examples 2 and 3, the number of training iterations in this example is ten times longer. A larger number of training iterations causes a larger difference in the learned FNNs parameter values between GPU and CPU implementations.

Example 5 (Plant Identification). In Examples 1 to 4, the number of GPU-FNN inputs are larger than 50. This example studies the GPU-FNN performance when the input dimension is small. This example uses the GPU-FNN to identify a nonlinear system. The plant that is to be identified is guided by the difference equation

$$y_d(t+1) = \frac{y_d(t)}{1+y_d^2(t)} + u^3(t). \quad (13)$$

In accordance with [2], [7], the training patterns are generated with $u(t) = \sin(2\pi t/100)$. For the purpose of comparisons, 50 000 training data and 200 testing data are produced. The GPU-FNN inputs are $y_d(t)$ and $u(t)$, and the desired output is $y_d(t+1)$. Performance is evaluated by the use of the root-mean-squared error (RMSE). The learning coefficient η is set to 0.07. Table VII shows the performance of the GPU-FNN for different numbers of thresholds and rules. Because the input dimension is small, Table VIII shows that a small number of rules achieve good performance. Table VIII also shows the training time and test result of CPU implementation. The test RMSE difference between GPU and CPU implementations is smaller than 10^{-7}

TABLE VIII
COMPARISONS OF DIFFERENT FNNS FOR THE IDENTIFICATION PROBLEM IN EXAMPLE 5

Classifier	ETS [5]	Simpl_ETS [6]	SAFIS [7]	GPU-FNN
Rule number	19	18	8	9
Testing RMSE	0.0082	0.0122	0.0116	0.0079

TABLE IX
TRAINING TIME COMPARISONS OF DIFFERENT GPU IMPLEMENTATION APPROACHES AND CPU FOR THE DATASET (ATTRIBUTES = 60) IN EXAMPLE 2

Rule number		5	34	139	427
Training time (sec)	CPU	90,596.9	355,833.6	1,288,006.1	3,824,729.0
	GPU(new)	173,345.2	262,465.9	285,186.0	414,738.3
	GPU	134,516.5	143,047.5	162,192.7	225,526.9
Time Speedup GPU vs GPU(new)		1.289 X	1.835 X	1.758 X	1.839 X

for different numbers of rules. Unlike the results in Examples 1–4, this example shows that the training time of the GPU implementation is smaller than that of CPU implementation for different numbers of rules. Because there are only two network inputs in this example, training time of CPU implementation is less than 1 s even when the rule number is 31. For this kind of simple learning problem, it is unnecessary to use the GPU-FNN for implementation. Benefits and disadvantages of the usage of the GPU-FNN in contrast to the CPU-FNN are discussed in the next section.

For comparison purposes, Table VIII shows the reported test errors of different FNNs that were applied to the same problem [7], which include an evolving Takagi–Sugeno fuzzy model (ETS) [5], a simplified version of the ETS (Simpl_ETS) [6], and a sequential adaptive fuzzy inference system (SAFIS) [7]. These FNNs use first-order TSK-type rules, where the number of consequent parameters in each rule is larger than the GPU-FNN. The results show that the test error of the GPU-FNN is smaller than those of the FNNs used for comparison. The number of rules in the GPU-FNN is approximately half of those in the ETS and simpl_ETS. The GPU-FNN uses one more rule than the SANFIS. However, the total number of network parameters in the SANFIS is 56 and is larger than the number of 45 in the GPU-FNN. This example shows that the GPU-FNN, as well as other FNNs with structure/parameter learning, is proposed to decrease the space of the fuzzy system and achieve good performance at the same time. However, as shown in Examples 1 to 4, there are still several problems (especially, for problems with high-dimensional attributes) that need a large set of rules to achieve good performance. Implementation by the usage of the GPU-FNN for these problems helps reduce training time to obtain a well-performed model.

VI. DISCUSSIONS

A. Analysis

This section analyzes the benefits and shortcomings of the GPU-FNN in contrast to the CPU-FNN. In the GPU-FNN, there are nr, nr, and nL threads to compute the membership functions, update antecedent parameters, and update consequent parameters, respectively. The speedup of the GPU-FNN heavily depends on the network input and output dimensions, and the number of rules. In Examples 1 and 3, the input and output dimensions of the GPU-FNN are larger than 100 and 2, respectively. The results in Tables II and V show that the GPU-FNN shows a speedup in training time, even when the number of rules is smaller than 5. In Examples 2 and 4, there are only two network outputs and the network input dimension is in the range of [50, 100], which is smaller than those in Examples 1 and 3. The results in Tables IV and VI show that the GPU-FNN achieves a speedup when the number of rules is large (such as larger than 30). When the number of rules is too small (such as smaller than 10), the GPU-FNN takes longer training time than CPU-FNN. However, Tables II and V show that the training and test performances have an obvious degradation for a small set of rules. As a result, FNNs with a large set of rules are inevitable in the two examples. The results in Examples 1 to 4 show that the GPU-FNN achieves a significant speedup for the problems with the number of attributes being larger than approximately 50. For the problems whose number of attributes is smaller than 50, an FNN with a small number of rules may achieve good performance. For these problems, the GPU implementation may take longer training time than CPU implementation (as Example 5 shows) though the former uses multiple thresholds. The major reasons are explained as follows. First, the GPU implementation needs additional data transfer between different memories, such as between host (PC) and global memory, and between global memory and shared memory. Second, the processing speed of the CPU is faster than the multicore processor in the GPU. As a result, a sufficient number of parallel processing threads in the GPU-FNN implementation is necessary to show its benefits. Therefore, the proposed GPU-FNN is most suitable to train FNNs with high-dimensional inputs (in particular, for those higher than 50).

B. Comparisons

This section compares the GPU-FNN implementation performance with another new GPU design approach. As stated in Section VI-A and shown in Fig. 6, computation of the firing strength μ_i in (3) is implemented by the use of the blocks of threads followed by the tree-reduction technique. Each thread in Fig. 6 does little work and only computes a membership value M_{kj}. This section tries a different design approach. In the new design approach, a block of threads that executes the membership functions for a single rule is implemented. That is, there are r threads in a block and the kth thread computes the n membership values M_{kj}, $j = 1, \ldots, n$, followed by their direct product to get μ_k. Each thread in this new design does much work than the GPU-FNN. The total number of parallel processing tasks in this new design is r in contrast with nr in the original GPU-FNN implementation. Tables IX and X show the training times of this new design [denoted as GPU (new)] for the datasets in Examples 2 (attributes = 60) and 5 (attributes = 2), respectively. Table IX shows that the new GPU implementation

TABLE X
TRAINING TIME COMPARISONS OF DIFFERENT GPU IMPLEMENTATION APPROACHES AND CPU FOR THE DATASET (ATTRIBUTES = 2) IN EXAMPLE 5

Rule number		5	9	26	31
Training time (sec)	CPU	0.143.	0.168	0.617	0.777
	GPU(new)	13.794	13.912	14.179	14.307
	GPU	13.504	13.621	13.676	13.794
Time Speed up GPU vs GPU(new)		1.021X	1.021X	1.037X	1.037X

approach takes long training time than the GPU-FNN implementation whether a large or a small rule set is implemented. In contrast with CPU, the GPU (new) implementation also helps reduce training time for a large set of rules. In Example 5, the FNN input dimension is only 2. Table X shows that training time of the GPU-FNN is only slightly shorter than that of the GPU (new) because the number of threads n in each block of the GPU-FNN is only 2. In conclusion, the GPU-FNN is much more efficient than the GPU (new) design approach. The speedup in training time increases with the number of FNN inputs n.

VII. CONCLUSION

This paper proposes the implementation of an FNN on a GPU to reduce FNN training time. In the GPU-FNN, blocks of threads are partitioned according to the parallel and independent properties of fuzzy rules. This partition makes good use of the shared memory and parallel processing properties of threads. Experimental results show that the GPU implementation significantly reduces training time compared with CPU implementation for FNNs with high-dimensional inputs. This GPU-FNN makes it more practical to apply an FNN to solve different problems, especially those with high-dimensional attributes. Experimental results of training and test error rates of the GPU-FNN demonstrate that it is worthwhile to implement an FNN on a GPU. More applications of the GPU-FNN will be studied in the future.

REFERENCES

[1] J. S. Jang, "ANFIS: Adaptive-network-based fuzzy inference system," *IEEE Trans. Syst., Man, Cybern.*, vol. 23, no. 3, pp. 665–685, May 1993.
[2] C. F. Juang and C. T. Lin, "An on-line self-constructing neural fuzzy inference network and its applications," *IEEE Trans. Fuzzy Syst.*, vol. 6, no. 1, pp. 12–32, Feb. 1998.
[3] D. Kukolj and E. Levi, "Identification of complex systems based on neural and Takagi–Sugeno fuzzy model," *IEEE Trans. Syst., Man, Cybern., B, Cybern.*, vol. 34, no. 1, pp. 272–282, Feb. 2004.
[4] N. K. Kasabov and Q. Song, "DENFIS: Dynamic evolving neural-fuzzy inference system and its application for time-series prediction," *IEEE Trans. Fuzzy Syst.*, vol. 10, no. 2, pp. 144–154, Apr. 2002.
[5] P.P. Angelov and D. P. Filev, "An approach to online identification of Takagi–Sugeno fuzzy models," *IEEE Trans. Syst., Man Cybern., B, Cybern.*, vol. 34, no. 1, pp. 484–498, Feb. 2004.
[6] P. P. Angelov and D. P. Filev, "Simpl_eTS: A simplified method for learning evolving Takagi–Sugeno fuzzy models," in *Proc. Int. Conf. Fuzzy Syst.*, 2005, pp. 1068–1072.
[7] H. J. Rong, N. Sundararajan, G. B. Huang, and P. Saratchandran, "Sequential adaptive fuzzy inference system (SAFIS) for nonlinear system identification and prediction," *Fuzzy Sets Syst.*, vol. 157, no. 9, pp. 1260–1275, 2006.
[8] P. Angelov and X. Zhou, "Evolving fuzzy systems from data streams in real-time," in *Proc. Symp. Evolving Fuzzy Syst.*, 2006, pp. 29–35.
[9] C. F. Juang and Y. W. Tsao, "A self-evolving interval type-2 fuzzy neural network with on-line structure and parameter learning," *IEEE Trans. Fuzzy Syst.*, vol. 16, no. 6, pp. 1411–1424, Dec. 2008.
[10] J. D. Rubio, "SOFMLS: Online self-organizing fuzzy modified least-squares network," *IEEE Trans. Fuzzy Syst.*, vol. 17, no. 6, pp. 1296–1309, Dec. 2009.
[11] J. A. M. HernandezmF.G. Castaneda and J. A. M. Cadenas, "An evolving fuzzy neural network based on the mapping of similarities," *IEEE Trans. Fuzzy Syst.*, vol. 17, no. 6, pp. 1379–1396, Dec. 2009.
[12] J. J. Rubio and J. Pacheco, "A stable online clustering fuzzy neural network for nonlinear systems identification," *Neural Comput. Appl.*, vol. 18, no. 6, pp. 633–641, 2009.
[13] J. A. Iglesias, P. Angelov, A. Ledezma, and A. Sanchis, "Evolving classification of agents' behavior: A general approach," *Evolving Syst.*, vol. 1, no. 3, pp. 161–171, 2010.
[14] J. J. Rubio, D. M. Vázquez, and J. Pacheco, "Backpropagation to train an evolving radial basis function neural network," *Evolving Syst.*, vol. 1, no. 3, pp. 173–180, 2010.
[15] J. J. Rubio Avila, "Stability analysis for an online evolving neuro-fuzzy recurrent neural network," in *Evolving Intelligent Systems: Methodology and Applications*, P. Angelov, D. P. Filev, and N. Kasabov, Eds. New York: Wiley-IEEE Press, 2010, ch. 8, pp. 173–198.
[16] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.
[17] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar./Apr. 2010.
[18] K. S. Kyong and K. Jung, "GPU implementation of neural network," *Pattern Recognit.*, vol. 37, no. 6, pp. 1311–1314, 2004.
[19] D. T. Anderson, R. H. Luke, and J. M. Keller, "Speedup of fuzzy clustering through stream processing on graphics processing units," *IEEE Trans. Fuzzy Syst.*, vol. 16, no. 4, pp. 1101–1106, Aug. 2008.
[20] Y. Tao, H. Lin, and H. Bao, "GPU-based shooting and bouncing ray method for fast RCS prediction," *IEEE Trans. Antennas Propag.*, vol. 58, no. 2, pp. 494–502, Feb. 2010.
[21] M. D. Bisceglie, M. D. Santo, C. Galdi, R. Lanari, and N. Ranaldo, "Synthetic aperture radar processing with GPGPU," *IEEE Signal Process. Mag.*, vol. 27, no. 2, pp. 69–78, Mar. 2010.
[22] T. R. Halfhill, "Parallel processing with CUDA-NVIDA's high-performance computing platform uses massive multithreading," *Microprocessor Rep.*, pp. 1–8, Jan. 2008.
[23] NVIDIA. CUDA. (2011). [Online]. Available at http://www.nvidia.com/object/cuda_home_new.html
[24] C. T. Lin, C. M. Yeh, S. F. Liang, J. F. Chung, and N. Kumar, "Support-vector-based fuzzy neural network for pattern classification," *IEEE Trans. Fuzzy Syst.*, vol. 14, no. 1, pp. 31–41, Feb. 2006.
[25] H. M. Lee, C. M. Chen, J. M. Chen, and Y. L. Jou, "An efficient fuzzy classifier with feature selection based on fuzzy entropy," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 31, no. 3, pp. 426–432, Jun. 2001.
[26] D. Michie, D. J. Spiegelhalter, and C. C. Taylor (1994), Machine learning, neural and statistical classification [Online]. Available: ftp.ncc.up.pt/pub/statlog/
[27] C. W. Hsu and C. J. Lin, "A comparison of methods for multiclass support vector machines," *IEEE Trans. Neural Netw.*, vol. 13, no. 2, pp. 415–525, Mar. 2002.
[28] K. M. Lin and C. J. Lin, "A study on reduced support vector machines," *IEEE Trans. Neural Netw.*, vol. 14, no. 6, pp. 1449–1459, Nov. 2003.
[29] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Berlin, Germany: Springer-Verlag, 2009, ch. 1.

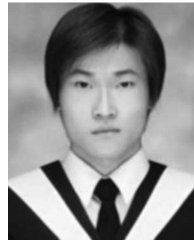
Chia-Feng Juang (M'99–SM'08) received the B.S. and the Ph.D. degrees in control engineering from the National Chiao-Tung University, Hsinchu, Taiwan, in 1993 and 1997, respectively.

Since 2001, he has been with the Department of Electrical Engineering, National Chung-Hsing University (NCHU), Taichung, Taiwan, where he has been a Distinguished Professor since 2009. He has authored or coauthored six book chapters, more than 65 refereed journal papers (including 34 IEEE papers), and more than 70 conference papers. Six of his published journal papers have been recognized as highly cited papers according to the Information Sciences Institute Essential Science Indicators. He is currently a member of the Editorial Board of the *Journal of Information Science and Engineering* and the *International Journal of Computational Intelligence in Control*. He is an Area Editor of the *International Journal of Intelligent Systems Science and Technology*. His current research interests include computational intelligence (CI), field-programmable-gate-array chip design of CI techniques, intelligent control, computer vision, speech signal processing, and evolutionary robots.

Dr. Juang received the Youth Automatic Control Engineering Award from Chinese Automatic Control Society, Taiwan, in 2006, the Outstanding Youth Award from Taiwan System Science and Engineering Society, Taiwan, in 2010, and the Excellent Research Award from NCHU, Taiwan, in 2010.

Teng-Chang Chen received the M.S. degree in electrical engineering from the National Chung-Hsing University, Taichung, Taiwan, in 2009.

He is currently a research assistant with the National Chung-Hsing University. His current research interests include computer vision, neural fuzzy systems, and graphic processing units.



Wei-Yuan Cheng received the B.S. and M.S. degrees from Tamkang University, Taipei, Taiwan, and Tunghai University, Taichung, Taiwan, both in mathematics, in 2004 and 2007, respectively. He is currently working toward the Ph.D. degree in electrical engineering with the National Chung-Hsing University, Taichung..

His current research interests include support vector machines, fuzzy classifiers, computer vision, and image processing and recognition.