A parallel method for tuning Fuzzy TSK Systems with CUDA

Bruno B. Ferreira
Programa de Pós-Graduação em Informática
Universidade Federal do Rio de Janeiro

Adriano J. O. Cruz
Programa de Pós-Graduação em Informática
Universidade Federal do Rio de Janeiro

Abstract

This paper studies an option for offloading some types of AI processing to the Graphics Processing Unit (GPU), by proposing the parallelization of the Batch Least Squares (BLS) method for tuning consequent parameters and the gradient method for tuning input fuzzy sets in a Takagi-Sugeno-Kang Fuzzy Inference System using the Compute Unified Device Architecture (CUDA). A method is proposed to generate the required intermediary matrices using heavy data parallelism. The learning consists of several iterations of BLS for the output values and gradient tuning for the input fuzzy sets. The explanation of the methods is followed by a performance comparison with a typical CPU-only approach and evaluation of the feasibility of using this method in real-time inside of a game.

Keywords:: Artificial Intelligence, Parallel Processing, Programming languages and techniques, CUDA, Fuzzy Logic

Author's Contact:

tinnus@gmail.com
adriano@nce.ufrj.br

1 Introduction

1.1 Fuzzy Logic and Fuzzy Inference Systems

Fuzzy Inference Systems (FIS) are inference systems based on Fuzzy Logic[Zadeh 1965]. They work like a "black box" with any number of input and output variables that are, in general, numeric (and real-valued). These variables are divided into sets defined by functions (usually trapezoid, triangle or gaussian) existing within the domain of the variable and that return a membership value between 0 and 1 for that fuzzy set. The inference process is the way by which the values of the output variables are calculated depending on the values set for the input variables, very much like other methods such as Artificial Neural Networks.

This work is based on the TSK[Takagy and Sugeno 1985; Sugeno and Kang 1988] FIS model, where the consequent of a rule for an output variable is a N-order polynomial function of the input values. Because of the mathematically flexible nature of this model, it is often a good choice for automatic learning of intelligent systems based on examples. The parameters that one would have to learn to define a TSK system then correspond to the definition of the input variables' sets (or membership functions), the output polynomial coefficients and the combination of fuzzy sets that form each rule of the system. Gaussian functions are usually used for automatic learning, due to their continuity and differentiantion properties, which are required for some parameter tuning methods such as the gradient method[Ross 2010].

1.2 GPGPU and CUDA

Graphics Processing Units (GPUs) are inherently data-parallel processors. Their architectures, originally created and evolved for drawing graphics on the screen, revolve around repeating the same, often simple, operation for a big amount of different data.

However, the erlier GPUs had very specific and low-programmable hardware, which made the utilization of them for general purpose computation a daunting task, with the overhead often neutralizing the gains in parallelization. Even then, as computing power and programmability of GPUs increased, some groups managed to develop methods to use them for general-purpose parallel computation. This new area of research was called General Purpose GPU (GPGPU).

Eventually, CUDA was introduced by nVidia, simplifying greatly the development of GPGPU programs. CUDA[1] is an extension of the C/C++ language that enables developers to create general-purpose code that runs on the GPU using a familiar language and development environment. Afterwards, other similar frameworks were also created for this purpose, such as OpenCL[2] and DirectCompute[3].

2 Related Work and Objectives

Considering methods in the realm of Fuzzy Logic, [Anderson and Coupland 2008] have adapted the classical linguistics-based Mamdani Fuzzy model for running on GPUs with CUDA, with promising results for large amounts of data.

A complete implementation of a parallel method for training of TSK systems from data samples using CUDA was done by [Juang et al. 2011]. In that paper, a structure called the GPU-FNN is proposed, based on the structure and learning methods of the SONFIN[Juang and Lin 1998]. The results obtained were very encouraging for problems with high dimensional inputs, since the proposed parallelization method explores concurrent processing in the domain of inputs dimensions and rules, but not between the samples themselves. Hence, it is not a very good choice for problems with a large number of samples and/or a small number of input dimensions (such as some that can be encountered in video games, e.g. the prediction of player behavior based on historical data).

The objective of this paper is, then, to provide an alternative method for automatic training of Fuzzy TSK systems that explores parallelism in the samples domain and achieves expressive speedups even for small input dimensions and rule numbers. Of special interest is the application of such methods for offloading parts of the AI computation of a game to the GPU, allowing the utilization of AI techniques that are traditionally regarded as too slow to be used in real time.

3 Technical Foundations

3.1 TSK Fuzzy Systems

As mentioned briefly in section 1.1 and explained in detail in [Ross 2010], a TSK FIS is composed by input and output variables, and a set of inference rules, where:

- Antecedents are the AND of tuples of the form *variable IS set*;
- Consequents are of the form *variable IS* $P(x_1, x_2, ..., x_n)$, where $P(x)$ represents a n-order polynomial of the input variables.

The calculation of the output of a typical TSK FIS is done as follows:

- Each rule is evaluated and given an *activation degree* based on the combination of membership values returned by each (variable, set) pair defined in the antecedent of the rule. This combination is analogous to an AND operation and is usually done by multiplying the membership values together. Considering gaussian membership functions for the input variables,

[1] http://www.nvidia.com/cuda
[2] http://www.khronos.org/opencl/
[3] http://en.wikipedia.org/wiki/DirectCompute

and according to [Ross 2010], the full expression for the activation degree of rule k is then

$$\mu_k(x) = \prod_{j=1}^{M} exp\left(-0.5 * \left(\frac{x_j - c_j^k}{w_j^k}\right)^2\right), \quad (1)$$

where M is the number of input variables and c_j^k and w_j^k are the center and width of the gaussian membership function defined for variable x_j and the fuzzy set defined for that variable in the antecedent of rule k.

- The final output for each variable is calculated as a weighted average of the evaluation of the polynomial outputs of each rule, where the weight given to each output value is the activation degree of that rule. This way, if $P_k(x)$ represents the output of rule k, the final output for one variable of the fuzzy system is

$$f(x) = \frac{\sum_{k=1}^{K} \mu_k(x) * P_k(x)}{\sum_{k=1}^{K} \mu_k(x)} \quad (2)$$

We can then conclude that, ultimately, the output $f(x)$ of a Fuzzy TSK system as described here for an input vector x depends on the values of c_i^k, w_i^k and the coefficients of the polynomials $P_K(x)$. The next section discusses methods to find those values by means of unsupervised training based on a set of samples with known outputs.

3.2 Tuning TSK Fuzzy Systems

Several methods have been proposed to tune the parameters of Fuzzy TSK systems[Ross 2010; Passino and Yurkovich 1998]. Two specific methods that are applicable to some or all of the parameters that have to be learned are discussed in the following sections. It is important to consider that the metric typically used as target for optmization (and the one used in these methods) is the Mean Squared Error (MSE), which is calculated as such:

$$\frac{\sum_{i=1}^{N} \sum_{l=1}^{L} (y_l^d(i) - y_l(i))^2}{N}, \quad (3)$$

where N is the number of samples used in training, L is the number of output variables, $y_l^d(i)$ is the desired value for the $i-th$ sample and the $l-th$ output variable, and $y_l(i)$ is the correspondent output given by the trained system.

3.2.1 The Batch Least Squares Method for tuning of TSK outputs

One of the classic ways to tune consequent parameters is by transforming the problem into a linear overdetermined system of equations and solving it using the classical Least Squares approach[Ross 2010].

Let us consider that all output polynomials are of order 0, which means that all of them correspond to just a constant value, and are represented by p_k. If we consider ϕ to be a vector containing the constant (in relation to p_k) values in equation 2 and θ to be the vector of p_k values, then equation 2 becomes $f(x) = \phi \cdot \theta$. This is for one sample datum only. Then, considering Φ as the matrix formed by joining the ϕ vectors for all samples, Y as the vector of training output values for all samples, and that our objective is to optimize for the bet fit of $f(x)$ to Y, we have the final overdetermined (considering $N > K$) linear system of equations to be optimized:

$$\Phi\theta = Y \quad (4)$$

It is worth noting that this rationale can be extended for output polynomials with degrees greater than 0. In this case, θ (and therefore ϕ) would be enlarged to encompass the extra coefficients. The extra values in ϕ would then be equal to the 0-order ones, multiplied by the according products of input variables as present in the polynomials (i.e. x_i, x_i^2, $x_i x_j$ etc).

3.2.2 The Gradient Method for tuning of TSK parameters

The gradient method can be used to tune all parameters in a FIS, and more information on it can be found in [Jang 1993; Ross 2010; Juang et al. 2011]. In our case, we want to tune the values of c_j^k, w_j^k and p_k for minimizing the MSE as defined in equation (3). We use the equations defined in [Juang et al. 2011] for one step of tuning for a single sample.

If we then want to do a single tuning step considering the MSE of all the samples, it suffices (since the gradient is a linear operation) to calculate the average of the error gradients for all samples and use that average in the tuning equations just once. This strategy is especially useful for concurrent environments, as will become clear in section 4.

4 Method Description

The proposed and developed method uses the BLS method for tuning consequents, the gradient method for tuning antecedents and considers a TSK Fuzzy System with 0-order polynomials (but could be easily extended for greater orders as described in section 3.2.1). The starting configuration of the gaussian membership functions is defined by dividing the input space uniformly in an arbitrary number of sets of equal width. This number of sets should be defined empirically depending on the problem. Other, more refined, methods could be used for defining the starting configuration of the gaussian parameters (such as ones based on clustering), as long as they can work on all samples at once and are based only on the training data (which makes the method described in [Juang et al. 2011] unsuitable). The general method is basically the same as the original ANFIS as described in [Jang 1993], considering that the samples are all presented at once and the global gradient is used for tuning of antecedents.

4.1 CPU Implementation

A reference CPU implementation was done with completely sequential code, without exploring any CPU-level parallelism with libraries such as OpenMP and MPI. The calculation of Φ and the gradient method tuning were created by hand, and for solving the Least Squares problem the LAPACK library was used. Since the tests were done in a Windows environment, the LAPACK libraries used for compilation were pre-built Windows binaries provided by the Innovative Computing Laboratory from University of Tennessee[4].

4.2 GPU Implementation

All of the algorithms, except the initial definition of gaussian parameters (which has negligible time) were implemented concurrently in CUDA. Like the CPU implementation, the code for calculating Φ and adjusting the antecedents was written by hand. The CULA[5] library was used for solving the Least Squares problem in the GPU. The hand-crafted code was separated into 3 kernels, the working of which will be detailed in the following sections. The standard parallel programming technique of tree reduction is used several times for finding the sum of all elements in an array. These reductions are all done inside each block, using shared memory as storage, which makes the operations easy to implement and very fast. The implementation of tree reduction is largely the same as in [Juang et al. 2011].

4.2.1 Starting up

Before starting the main iteration loop and running the first kernel, the starting data (sample inputs and outputs and initial gaussian parameters) is copied from host (CPU) memory to device (GPU) memory. Then a main loop starts and what is described in the following sections is done for an arbitrary number of iterations desired for the convergence of the system.

[4] http://icl.cs.utk.edu/lapack-for-windows/

[5] http://www.culatools.com/

4.2.2 Calculating Phi

The partition used for this kernel was of one block for each sample and one thread for each rule. Due to the tree reduction code needing a power-of-two-sized array to work, the number of rules is rounded up to the next power of two and the activation of the extra dummy rules is set to 0. After running this kernel we have the matrix Φ full and ready in device memory. The working of this kernel is then as follows:

- *each thread*: Calculate the activation degree (μ_k) for the current rule and sample (as per equation (1)) and store it in shared memory;
- *all threads in a block*: Calculate the sum of activations of this sample for all rules by tree reduction on shared memory;
- *each thread*: Normalize the rule's activation degree by dividing it by the sum of all activations in the block and store it in Φ.

4.2.3 Finding θ with Least Squares

This consists of a simple call to the CULA library, passing the calculated Φ to find the corresponding θ. Before the library call, a copy of Y is done, because the result of the estimation is written in the same array passed as the right-hand side of the linear system.

4.2.4 Calculating the error gradients

The calculation and application of the error gradients was divided in two kernels. The first one calculates the error gradients for each sample. The second is responsible for averaging them and applying them to the parameters. This division was made because the block-thread partition for each part of the computation is different.

The partition of the first kernel is the same as the one for calculating Φ: one block per sample and one thread per rule. The same idea of dummy rule-values was used to pad the tree reduction present. This kernel is largely responsible for calculating the error gradients. The working of this kernel is then as follows:

- *each thread*: Calculate one element of $\phi \cdot \theta$ for all outputs and store it in shared memory;
- *all threads in a block*: Calculate the sum of all values found above for each output by tree reduction on shared memory. This is equal to the output of the fuzzy system for each sample and output variable;
- *one thread per block*: Find the error for the current sample by subtracting the expected output (in Y) from the calculated output;
- *each thread*: Calculate the error gradients of c_j^k and w_j^k for this sample and rule, and for each input variable. Store them in global memory.

4.2.5 Applying the error gradients

The bulk of the work in the last kernel is calculating the average of each error gradient across all samples by a tree reduction operation inside each block. Thus, the partition for this kernel is one block per rule, and one thread for a set of samples. Since we cannot start as many threads inside a block as there might be samples, the number of samples is divided by a fixed number of groups (32 in this implementation) and a thread is created for each group. This thread first sums, sequentially, the error gradients for all samples in its group. A reduction is done afterwards to get the global sum for the rule corresponding to that block. The first thread of the block then calculates the final averages and applies them to tune c_j^k and w_j^k. Some implementation care must be taken to treat the last group since it might have a size that's different from the others. The working of this kernel is then as follows:

- *each thread*: Calculate the sum of the error gradients for all samples inside the corresponding group and store it in shared memory;
- *all threads in a block*: Find the sum of gradients from all groups by tree reduction on shared memory. This is equal to the sum of gradients for all samples;
- *one thread per block*: Calculate the final averages by dividing the gradient sums by the number of samples;
- *one thread per block*: Apply the final gradient averages to tune the values of c_j^k and w_j^k.

After running this kernel, one iteration of tuning the fuzzy system is complete. Since now the antecedents have changed, in the next iteration Φ will be different, and the answer to the Least Squares problem will also (hopefully) be closer to the optimal value. This should continue until one is satisfied with the result of the training. At the end of each iteration, the state of the fuzzy system and all the temporary matrices are available in device memory and may be copied to host memory for checking and using.

5 Results

The developed method was tested by training a 0-order Fuzzy TSK system with two inputs and 35 rules. The CPU and GPU versions of the algorithm were executed in 3 machines, with 16^2, 32^2, 64^2, 128^2 and 256^2 samples. Each training session lasted for 100 iterations.

The source of the training data is a Mamdani FIS used to park a simulated car while driving in reverse with constant speed as described in [Nguyen and Widrow 1990; Kosko 1992]. Since this system had 5 sets for one input variable and 7 sets for the other, we defined that the same two inputs in the TSK system would also start with 5 and 7 gaussian membership functions. It is expected that the gradient method adjusts them to the best shape to represent the original system.

Samples of the system were then taken by dividing the space of the input variables in 16x16, 32x32, 64x64, 128x128 and 256x256 points and evaluating it at those points. This generated 5 sets of data, containing 16^2, 32^2, 64^2, 128^2 and 256^2 samples.

5.1 Hardware and software specification

The code for the tests was compiled on Windows 7 with the Visual Studio compiler and CUDA Toolkit version 4.2. The compiler was set to generate 32-bit code. The tests were ran on 3 different machines (to be referred to as Machine 1, Machine 2 and Machine 3). All machines ran Windows 7 Professional 64-bit. Table 1 contains the CPU, GPU and RAM specification of each machine.

Machine	CPU	RAM	GPU
Machine 1	i7 2600 @3.40GHz	8GB	GeForce GTX 550 Ti
Machine 2	i7 960 @3.20GHz	16GB	Tesla C2070
Machine 3	i7 960 @3.20GHz	16GB	GeForce GTX 590

Table 1: *Test machine configurations*

5.2 Testing methodology and resulting times

The time spent on each operation (Calculating Φ, least squares and gradient method) on each iteration was recorded. The GPU operations were timed by using CUDA events. The CPU operations were recorded using the QueryPerformanceCounter Windows API call. The time taken to do the initial copying of samples (as well as copying back the final learned parameters) and gaussian parameters was not counted because it is negligible compared to 100 iterations of the method. The time spent for duplicating Y at each iteration was also very small, and is not listed in the tables, although it was accounted for in the total time and speedup calculations.

Table 2 shows the breakdown of total times by each of the 3 main parts of the algorithm.

Machine		Calc. Phi	Least Squares	Gradient	Total
M. 1	CPU	15.930	42.154	8.132	66.374
	GPU	1.165	8.801	3.579	13.550
Speedup		13.67x	4.79x	2.27x	4.90x
M. 2	CPU	19.349	55.317	8.407	83.315
	GPU	0.486	11.973	1.493	13.959
Speedup		39.77x	4.62x	5.63x	5.97x
M. 3	CPU	19.238	55.054	8.374	82.907
	GPU	0.410	11.782	1.143	13.341
Speedup		46.91x	4.67x	7.32x	6.21x

Table 2: *Breakdown of execution times (in seconds) for each step in all 3 test machines, for* 256^2 *samples*

5.3 Interpretation of results

In Table 2, it clearly shows that the better general speedup for Machine 2 and Machine 3 comes from the slower CPUs (when compared with Machine 1), rather than the different GPUs, since the GPU times are all more or less equivalent. However, from Table 2 we can see that the speedups were very different between the GPUs depending on what part of the computation was considered. The Phi calculation and gradient adapting performed visibly better on the Tesla C2070 and the GeForce GTX 590, whereas the opposite happened with the Least Squares operation.

It's also interesting to note that the proposed strategy managed to achieve a mesurable speedup even with a small fuzzy system (only 2 inputs and 35 rules), by exploring parallelism among the sample data set. This is in contrast to the GPU-FNN described in [Juang et al. 2011], where a system with only 2 inputs performed worse on the GPU than on the CPU due to parallelism being explored primarily on the input dimensions. The GPU-FNN, on the other hand, achieved bigger speedups with high-dimensional inputs.

Finally, the raw speedup values may be a little misleading since no CPU-level parallelism was explored in the tests. However, in typical video game scenarios, both CPU time and GPU time might be at a premium depending on the particular situation. So, even if a parallel CPU implementation could reduce the raw speedups and put both CPU and GPU at the same performance ballpark, it is still useful to be able to allocate both resources efficiently and choose between them depending on the specific needs of the application.

6 Conclusions and future work

This paper proposed a way of training a Fuzzy Inference System of the TSK type on the GPU. A method was described to offload the training of consequents with Least Squares and antedecents with the Gradient Method to the GPU. This method was tested with a small system (2 inputs and 35 rules) and varying amounts of training sample data. Tests ran on 3 medium-to-high-end machines showed that the training time on the GPU surpassed the CPU by a factor of around 5-6x.

We can then conclude that the proposed method works as a way to offload the training of Fuzzy Inference Systems to the GPU, and can even reduce the time and energy requirements of running such a training. Of special note is that expressive speedups were achieved even with a small number of inputs and rules, as long as there is a considerably large number of samples. This paves the way for using Fuzzy Systems efficiently for learning complex patterns in a broader set of situations.

6.1 Ramifications for game AI

One of the strongest motivations of this work was the idea of offloading parts of the AI computation of a game to the GPU, in order to enable more types of AI in games, especially those involving real-time learning of player behavior. It is expected that these results encourage game developers to give a second look at this approach, which traditionally was frowned upon due to the heavy computation time that was usually required.

6.2 Future Work

It remains to be studied how the proposed method performs with data sets of higher dimensionality and number of rules, and especially how it compares to the GPU-FNN[Juang et al. 2011], both in terms of final MSE and performance on high-dimensional systems.

There are also some optimizations that remain to be experimented. One possible optimization in the method proposed is to do the least squares estimation only at each K^{th} step, since it is the costliest operation, and also one with a small speedup. Some lower-level optimizations include studying the use of constant and texture memory and modifying the second gradient kernel to do a two-level reduction in order to exploit more parallelism.

Also, the method should be implemented in real-time inside a game to evaluate how much of a speed impact it has, and if the technique can be used to enhance the player's sense of challenge imposed by the AI. An especially interesting prospect is implementing a game where a virtual opponent uses this system to learn the player behavior over time, and adapt to it by choosing his actions based on predictions of future player behavior.

Acknowledgements

We would like to thank Thiago Gomes and Lucas Ribeiro for helping running the tests and lending two of the machines used to run them.

References

ANDERSON, D., AND COUPLAND, S. 2008. Parallelisation of fuzzy inference on a graphics processor unit using the compute unified device architecture. In *Proceedings of the 2008 UK Workshop on Computational Intelligence*.

JANG, J.-S. 1993. Anfis: adaptive-network-based fuzzy inference system. *Systems, Man and Cybernetics, IEEE Transactions on 23*, 3 (may/jun), 665 –685.

JUANG, C.-F., AND LIN, C.-T. 1998. An online self-constructing neural fuzzy inference network and its applications. *Fuzzy Systems, IEEE Transactions on 6*, 1 (feb), 12 –32.

JUANG, C.-F., CHEN, T.-C., AND CHENG, W.-Y. 2011. Speedup of implementing fuzzy neural networks with high-dimensional inputs through parallel processing on graphic processing units. *Fuzzy Systems, IEEE Transactions on 19*, 4 (aug.), 717 –728.

KOSKO, B. 1992. Comparison of fuzzy and neural truck backer-upper control system. In *Proceedings of 1992 IJCNN International Joint Conference*, 339–361.

NGUYEN, D., AND WIDROW, B. 1990. The truck backer-upper: An example of self-learning in neural network. *IEEE Control System Magazine*, 10, 18–23.

PASSINO, K. M., AND YURKOVICH, S. 1998. *Fuzzy control*. Addison-Wesley, Menlo Park, Calif.

ROSS, T. J. 2010. *Fuzzy Logic with Engineering Applications*. John Wiley & Sons, Ltd.

SUGENO, M., AND KANG, G. T. 1988. Structure identification of fuzzy model. 15–33.

TAKAGY, T., AND SUGENO, M. 1985. Fuzzy identification of systems and its applications to modeling and control. *IEEE Transactions on System, Man and Cybernetics 15*, 1, 116–132.

ZADEH, L. A. 1965. Fuzzy sets. *Information and Control 8*, 338–353.