A Parallel Hardware Architecture for Information-Theoretic Adaptive Filtering

Stefan Craciun*+, Alan D. George*, Herman Lam*, Jose C. Principe+

* NSF Center for High-Performance Reconfigurable Computing (CHREC)
+ Computational Neuro-Engineering Laboratory (CNEL)
Department of Electrical and Computer Engineering, University of Florida
Gainesville, Florida, 32611-6200

(email: craciuns@ufl.edu, george@chrec.org, hlam@chrec.org, principe@cnel.ufl.edu)

Abstract — **Information-theoretic cost functions such as minimization of the error entropy (MEE) can extract more structure from the error signal, yielding better results in many realistic problems. However, adaptive filters (AFs) using MEE methods are more computationally intensive when compared to conventional, mean-squared error (MSE) methods employed in the well-known, least mean squares (LMS) algorithm. This paper presents a novel, parallel hardware architecture for MEE adaptive filtering. The design has been implemented and evaluated in real-time on one of the servers of the Novo-G machine in the NSF CHREC Center at the University of Florida, believed to be the most powerful reconfigurable supercomputer in academia. By pipelining the design and parallelizing independent computations within the algorithm, our proposed hardware architecture successfully achieves a speedup of 5800 on one FPGA, 23200 on one quad-FPGA board, and 46400 on two quad-FPGA boards, as compared to the same algorithm running in software (optimized C program) on a single CPU core. Just as important, our results show that this reconfigurable design does not lose precision while converging to the optimum solution in the same number of steps as the software version. As a result, our approach makes it possible for AFs using the MEE cost function to adapt in real-time for signals that require a sampling rate in excess of 400 kHz and thus can target a much wider range of applications.**

I Introduction

Adaptive filters are very important in the area of signal processing and have a very large number of applications in digital signal processing (DSP), such as system identification, noise cancellation, and signal prediction to name just a few. If the statistics of the input signal are unknown (as is often the case), an adaptive filter can be used to estimate the required signal statistics by means of an iterative learning (adaptation) process. Adaptive filters are often referred to as "intelligent" or "smart" systems precisely because they are capable of dynamically estimating the statistics of the incoming signal and, furthermore, adjusting their internal parameters (impulse response/weights) to meet a specific performance criterion. This performance criterion is referred to as the cost function. The cost function defines the rules of optimal adaptation, and is used by the AF to compute the new filter weights. The new computed weights are then fed back to the adaptive filter and replace the old weights until an optimal and stable solution is reached. All learning algorithms search the solutions space for global minima. During each iteration, weights are adjusted in small increments by moving them in the direction opposite to the gradient. Ideally, the weights of the system will adjust and reach the global minima in a finite number of iterations [1].

The most popular criterion used in adaptation has been the mean-squared error (MSE) [2] because it presents a tractable mathematical solution and leads to very simple and computationally efficient algorithms such as least mean squares (LMS). Although the MSE cost function has proven to be successful, it is only an optimum solution when applied to Gaussian signals with linear filters [3]. A new cost function, minimization of error entropy (MEE) proposed in reference [4] and inspired by concepts of information theory, adapts by minimizing the average information content (entropy) of the error rather than just the power of the error [5]. The minimization of entropy extracts as much uncertainty as possible from the error signal, leading to better weights when compared to the MSE criterion. This alternative cost function has demonstrated superior performance when compared to the MSE cost function [5], especially for nonlinear signal processing. However, Renyi's quadratic entropy needs transcendental evaluations of sample pairs for its estimation, resulting in a substantial increase in computational complexity, which to date has made this algorithm impractical for real-time implementation (online learning).

Reconfigurable computing (RC) provides an attractive solution by creating a unique hardware architecture that can be

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422.

978-1-4244-9517-7/10/$26.00 ©

adapted to the computational requirements of a particular cost function. The efficient mapping of complex algorithms onto a hardware architecture based upon FPGA technology can take advantage of the unique data flow and inherent data dependencies that exist within the algorithm. Using RC techniques such as parallel data processing (wide parallelism) and pipelining (deep parallelism), we can create a sustainable architecture that makes the MEE cost function more appealing for real-time applications and for cases when the input signal requires high sampling rates.

The organization for the remainder of the paper is as follows. In Section 2, the computational requirements and an overview of the MEE algorithm are given. In Section 3, we describe the proposed architectural model for an AF based upon a MEE cost function and the design of the subcomponents of the architecture. This new design has been implemented and evaluated on up to two boards (i.e. eight FPGAs) in one server of Novo-G. In Section 4, we present results and analysis of real-time experiments of the AF conducted on Novo-G. In doing so, we measure the speedup of the AF as compared to software, and at the same time demonstrate its precision and scalability. Section 5 provides a summary and conclusions.

II Algorithm Overview

In the signal-processing domain it has been shown that designing and synthesizing high-speed architectures for feed-forward structures (for example the finite impulse response or FIR filter) is considerably easier than designing hardware architectures for signal-processing algorithms that require feedback [6]. While pipelining can significantly increase the speed at which the filter converges, in adaptive systems it has limited impact due to feedback [7]. This feedback loop that updates the filter coefficients prevents the next input from being processed until the filter coefficients have been updated. A solution to this problem is to delay the weight updates by a given number of cycles and allow a new input to be processed using the weights updated *n* cycles back, which is referred to as the *delay technique* in signal processing. Even though the weights are updated every cycle, there is a delay of *n* cycles between the current input and the time the weights will be updated. Such is the case of the DLMS algorithm [8], which for small delays and an appropriate step size efficiently uses the pipelining technique to increase the throughput of the regular FIR filter [9]. Another very similar and useful method is to use the *interleaving technique* [10]. This method allows for the introduction of delays at different stages of the transfer function, making it possible to pipeline the overall structure.

We observe that there is a fundamental difference between MEE cost functions and those of conventional LMS filters. For MEE cost functions, the new weights are computed based upon the current and past input values. The adapting weights are computed using a batch of input samples that generate a sequence of errors. The number of errors used to compute the new weights is known as the window size. Within one feedback loop, there are significantly more computations before each weight is updated. This observation clearly suggests that pipelining and parallelizing an MEE algorithm can have a much larger impact on hardware speedup than using the same techniques on MSE cost functions, since there is a considerably longer sequence of computations within the feedback loop.

The general structure for adaptive filters can be divided into three major blocks, as shown in Figure 1.

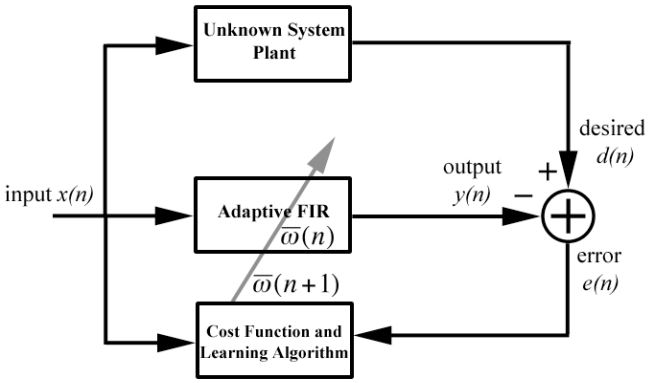

Figure 1. Adaptive filter structure

The first block is an adaptive FIR filter capable of changing its impulse response (filter weights). The output of this filter *y(n)* is simply the dot-product between the input vector and the weight vector:

$$y = \sum_{i=1}^{n} x_i \cdot \omega_i \qquad (1)$$

The output *y(n)* is then compared to a desired signal *d(n)*. In supervised learning, the desired signal represents the goal of the filter. As the output of the filter approaches this desired signal, the weights also approach the optimal solution. In system identification, the desired signal is the output of an unknown system (plant) for which the AF is trying to find the transfer function. The error *e(n)* is the difference between the filter output and the desired value $e(n) = d(n) - y(n)$.

The second building block consists of the cost function and learning algorithm. Cost functions define the rules of adaptation (optimal criterion) and the learning algorithm computes the new filter weights, which are then fed back to the first block. For the MEE cost function the iterative weight updates are of the form:

$$\omega(n+1) = \omega(n) + \mu \nabla V \qquad (2)$$

where $\omega(n+1)$ represent the new weights, $\omega(n)$ the current weights, μ is the step size, and ∇V is the gradient of the information potential (IP). Erdogmus et al. [4] has defined a nonparametric estimator of the IP as:

$$V = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \kappa(e_i - e_j) \qquad (3)$$

where κ represents a kernel function from density estimation. We can obtain the gradient of the IP by taking the derivative with respect to the weights:

$$\nabla V = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \kappa(e_i - e_j) * (e_i - e_j) * (x_i - x_j) \qquad (4)$$

In Eq. 4, we have used the Gaussian kernel for κ defined as:

$$G(e_i - e_j) = \frac{1}{\sqrt{2\pi}\sigma} \exp^{\frac{\|e_i - e_j\|^2}{2\sigma^2}} \qquad (5)$$

The challenge here is to design an architecture that accelerates this iterative process by exploiting the maximum amount of inherent parallelism (wide and deep) while taking advantage of the data dependencies within each iteration. Two important parameters define the computational complexity of the overall filter adaptation. The first is the *filter size*, which is equal to the number of weights. As the order of the filter is increased, more weights have to be computed, fed back, and updated. For gradient ascent, the complexity of the algorithm is linear or $O(L)$ with respect to the filter order *(L)*. A more influential parameter governing the complexity of the MEE cost function is the *window size* (*n* in Eqs. 3 and 4). As seen in Eq. 4, the gradient of the IP is a double summation over *i* and *j*, which means that the window size over which the IP gradient is estimated is the critical factor influencing the computational complexity of the algorithm. The computational complexity is $O(N^2)$ with respect to the window size. Figure 2 shows the quadratic increase in computation time (for software) when the window size is increased.

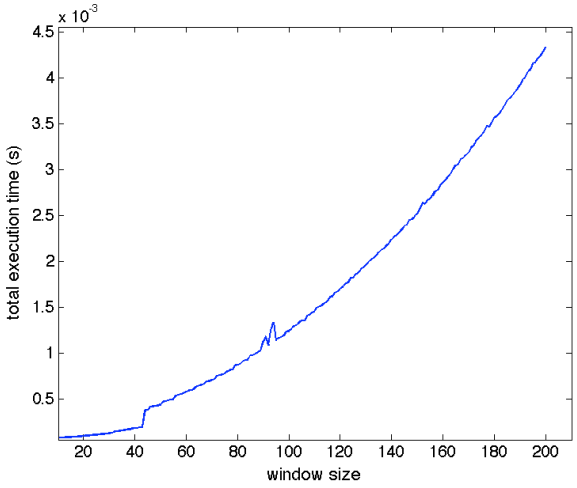

Figure 2. Software execution time of one weight update as a function of window size

It should also be noted that the window size has a direct effect on the (filter) performance of the algorithm. A larger window size will provide a better IP estimate, directly translating into a smoother and faster convergence of the weights to the optimal solution. However, the quadratic increase in complexity creates a steep tradeoff between filter performance and computation time. This paper proposes an architecture that transforms the time-performance tradeoff into a linear dependency.

Besides its computational complexity, the MEE cost function presents numerous challenges. These challenges include the need to accurately evaluate the exponential function as part of the Gaussian kernel (Eq. 5) and the need to avoid losing precision over multiple iterations. Past AF implementations have strictly used either entirely floating-point or fixed-point designs [11,12]. Fixed-point AFs present a much lower latency [11] and require less FPGA logic resources. The effect of small latency on feedback systems is a very popular topic in AF and provides numerous advantages. However, a floating-point implementation has the major advantage of maintaining better precision with no overflow for big swings in the input signal [13]. This behavior translates directly to better filter performance, because the filter requires less number of iterations to find the optimum solution, and at the same time the solution is more accurate. The design we propose is a hybrid of the two, utilizing precise floating-point functions where precision is needed and simple fixed-point blocks where precision is less important. Due to its complex cost function and relatively novel approach, prior to this work AFs using the MEE cost function have not been realizable for high-speed, real-time processing. This paper explores an efficient parallelization of the MEE cost function as a part of an AF that can significantly accelerate the adaptation process, providing a reconfigurable design that can be used in real-time for numerous applications.

III Parallel Algorithm and Architecture

In this section, the main building blocks of the AF (Adaptive FIR and cost function shown in Figure 1) are decomposed into smaller and more detailed components, illustrating the novel manner in which we have mapped the MEE algorithm onto a hardware structure. Batch learning represents the centerpiece of this algorithm and is the key idea that inspired and enabled this reconfigurable design. For batch learning, all the input data is available *a priori*. Ideally, in a fully pipelined design, the frequency at which the input vectors enter the system is equal to the clock rate. Figure 3 shows the window size over which we will evaluate the IP gradient ∇V (Eq. 4), which is then used to update the weights (Eq. 2). If we have all the samples over the entire window size available, we can divide the window into a sequence of input vectors and arrange them chronologically *x(0)* through *x(n-1)*, in the same manner in which they would enter the pipeline. The size of each input vector is equal to the filter order.

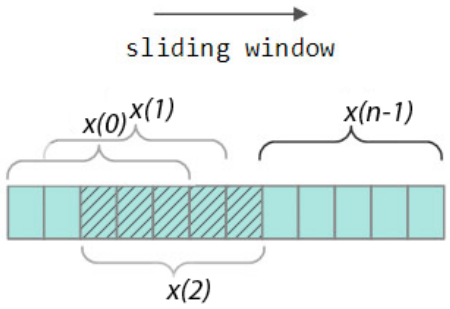

Figure 3. Forming the sequence of input vectors from a given window size

The IP gradient is evaluated over a window of errors. The latency of the entire design becomes less detrimental to speedup as the window size increases. We will show later in this section that speedup is linearly proportional to the window

size. Evaluating the IP gradient over a window size increases in precision as the window grows, but we should be very careful in choosing a particular window size because the complexity of this evaluation is $O(N^2)$ as shown in Eqs. 3 and 4. Therefore, hardware resources required to implement the IP gradient also increase quadratically as a function of window size, resulting in a fewer number of AFs that can fit onto each FPGA for very large window sizes, and consequently limiting speedup.

A. FIR Filter

The hardware design and implementation of a FIR filter has been rigorously studied in the past [14, 15, 16, 17]. In our work, we use a simple pipelined implementation that allows a new input vector to be clocked in every clock cycle. Figure 4 shows a detailed structure of the FIR architecture. Just as the sliding window in Figure 2 suggests, a new input vector is presented to the FIR at every clock cycle, when a new sample enters the input delay line. Concurrently, all past samples shift one register to the right and the last value *x(1)* (the oldest sample in the delay line) is simply forgotten. The new input vector $\overline{X} = [x(1) \quad x(2) \quad \ldots \quad x(n)]$ is multiplied by the current weight vector $\overline{W} = [\omega_1 \quad \omega_2 \quad \ldots \quad \omega_n]$, and then each term of the dot-product is added according to Eq. 1. In our design, multiplication is performed with a floating-point mega-function from Altera's core library and the additions that follow are computed in fixed-point. The weight registers store floating-point values since, for the same number of bits, floats have a much wider dynamic range, which is necessary because the weights are changed by a wide range of values [18]. A 32-bit fixed-point range would be insufficient to represent this wide range of values. Finally, the output of this block is the error $e(n)$, where *n* represents the time index.

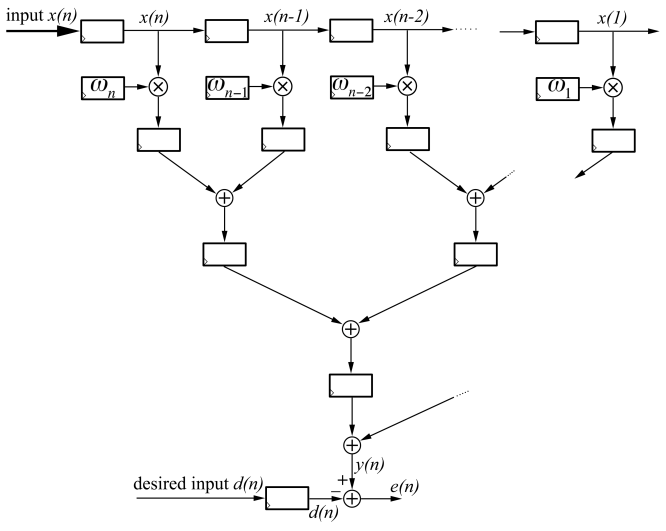

Figure 4. Adaptive FIR functional block

B. Mapping MEE Cost Function to Hardware

The MEE cost function can be further divided into four main subcomponents as suggested by Eq. 4. Figure 5 shows these four subcomponents, their interconnections, and the direction of the dataflow among them. All four blocks are pipelined to create the MEE cost function. We will next look at each block in more detail to describe the fine-grained parallelism that lies at the root of the MEE algorithm.

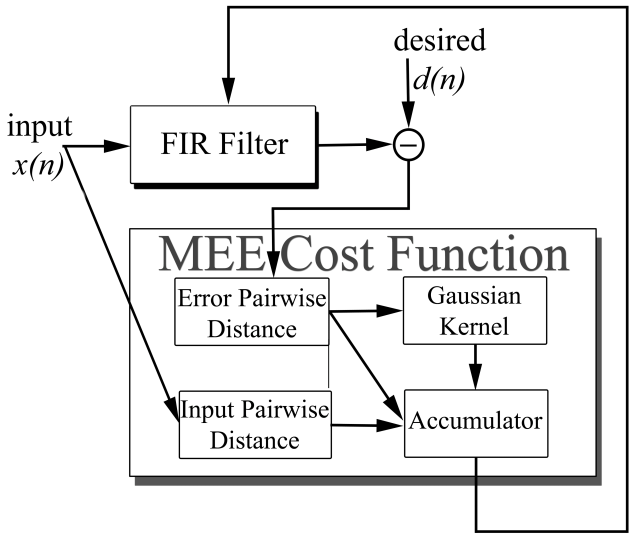

Figure 5. MEE cost function subcomponents

1) Error Pairwise Distance Block

The MEE cost function (Eq. 4) uses the error calculation provided by the FIR block to evaluate the IP gradient. This equation is composed of the three terms that are multiplied and then summed over *i* and *j*. The first term inside the double summation is the pairwise distance between all errors:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}(e_i - e_j)$$

This task can seem daunting because for a window size of *N*, $N^2/2$ pairwise error distances have to be computed. However, the FIR block is designed such that after an initial latency, the errors become available in a sequential manner, after each clock cycle, which means that we can start computing pairwise subtractions as soon as the first two errors become available. The design illustrated in Figure 6 shows the errors stored in a registered delay line, and all the pairwise distances computed between the current error and all the past ones. This parallelized design of pairwise distance calculations achieves considerable speedup when compared to software because it employs *N* clock cycles to compute $N^2/2$ subtractions. For every new error, the number of pairwise distances increases by one until it reaches a maximum of *n*.

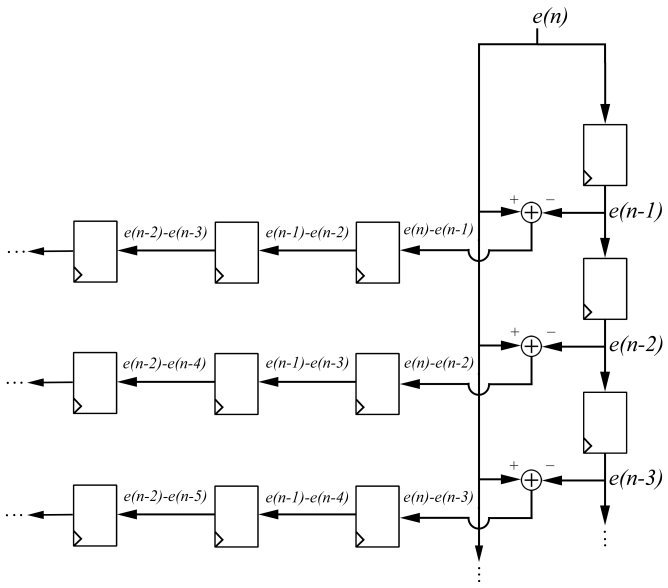

Figure 6. Pipelined pairwise distance computation

When the last error *e(n)* (for a given window size = *n*) is clocked into the delay line, the pairwise distance block will compute the final *N* pairwise distances between error $e(n)$ and all previous errors $[e(n-1), e(n-2), e(n-3), ..., e(1)]$. All error distances are stored in a 2D array of pipelined registers. By taking advantage of the algorithm dataflow, this design will use only *N* subtractors to compute $N^2/2$ pairwise distances in *N* clock cycles. The following 2D matrix is constructed when all *N* errors have passed through this block:

$$\begin{bmatrix} e_2-e_1 & e_3-e_2 & \dots & e_{n-1}-e_{n-2} & e_n-e_{n-1} \\ . & e_3-e_1 & \dots & e_{n-1}-e_{n-3} & e_n-e_{n-2} \\ . & . & \dots & e_{n-1}-e_{n-4} & e_n-e_{n-3} \\ \dots & \dots & \dots & \dots & \dots \\ . & . & \dots & . & e_n-e_1 \end{bmatrix}$$

It should be noted that this is an upper triangular matrix where one column is computed for each clock cycle starting from the left. The computation of distance matrices is very important in a number of fields including bioinformatics, physics, and chemistry. In general, for calculating pairwise distances in applications such as structural or sequential alignments, systolic arrays are used to accelerate calculations [19]. However, in this case, not all errors for which we wish to calculate the pairwise distances are available at the same time. They become available one by one, every clock cycle. In such cases, using a systolic array would prove to be a waste of resources. The pipelined design shown in Figure 6 is much more effective, parallelizing as many computations as are available (all subtractions between the current error and all past errors) within each clock cycle.

2) Gaussian Kernel Block

The second building block is used to compute the Gaussian kernel computation: $\sum_{i=1}^{n}\sum_{j=1}^{n} G(e_i - e_j)$

Eq. 5 shows the complete form of this kernel. As soon as the pairwise error distances become available, they are inputted to the Gaussian kernel block as shown in Figure 7. This block is directly connected to the pairwise distance block, such that all pairwise distances computed within the current cycle are fed directly into a Gaussian kernel block.

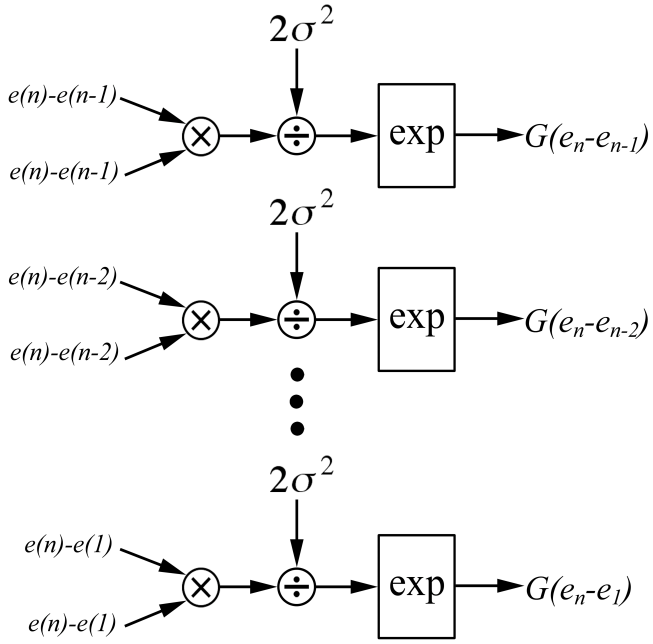

Figure 7. Gaussian kernel block

For a given window size of *N*, the cost function requires $N^2/2$ Gaussian kernel computations. However, by efficiently mapping the data flow of this algorithm to hardware, we compute $N^2/2$ Gaussian kernels in only *N* cycles plus the latency of one kernel. The exponential block used is Figure 7 comes from Altera's floating-point, mega-function library. This floating-point exponential has the capability to be pipelined, as do all algebraic functions used in this design, and as a result it minimizes the impact of latency on the total execution time. The results of the Gaussian kernel block are stored in a 2D register array. The following 2D matrix is constructed when all $N^2/2$ Gaussian kernels have been computed:

$$\begin{bmatrix} G(e_2-e_1) & G(e_3-e_2) & \dots & G(e_{n-1}-e_{n-2}) & G(e_n-e_{n-1}) \\ . & G(e_3-e_1) & \dots & G(e_{n-1}-e_{n-3}) & G(e_n-e_{n-2}) \\ . & . & \dots & G(e_{n-1}-e_{n-4}) & G(e_n-e_{n-3}) \\ \dots & \dots & \dots & \dots & \dots \\ . & . & \dots & . & G(e_n-e_1) \end{bmatrix}$$

Each column in this matrix is computed one clock cycle at a time starting from the left-most kernels.

The last term remaining in the MEE cost function (Eq. 4) is the input pairwise distances $\sum_{i=1}^{n}\sum_{j=1}^{n}(x_i - x_j)$. The hardware required for computing these terms is identical to that of the error distances block. All of the three building blocks are pipelined as shown in Figure 9.

The tree 2D arrays contain all the terms needed to compute the gradient of the IP in Eq. 4. The double summation becomes an iterative accumulation of these terms as they become available.

3) Accumulator Block

Figure 8 shows the general pipelined architecture of the accumulator. The three matrices are multiplied and the results are added to compute the IP. One accumulator block is dedicated to each row of the matrix. In total, there are *N* accumulators to parallelize the double summation in Eq. 4; each is responsible for computing a quantity ∇V_i of the IP.

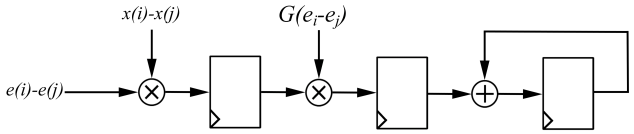

Figure 8. Pipelined Accumulator

By accumulating the terms of Eq. 4 every time a new column of the 2D arrays becomes available, we further accelerate the algorithm. The double summation containing $N^2/2$ additions is completed in only *N* clock cycles plus the latency of the pipelined accumulator. By parallelizing the computations of pairwise distances, and pipelining them with the Gaussian kernel block and the final accumulator block, we improve the asymptotic time complexity of the algorithm from $O(N^2)$ in the serial form to $O(N)$ in the parallel form.

Figure 9 shows the entire parallelization of the MEE cost function. The inputs are *n* samples from the batch $\overline{X} = [x_1 \quad x_2 \quad \ldots \quad x_n]$ (*n* is the window size) and *n* errors $[e(n) \quad e(n-1) \quad \ldots \quad e(1)]$ (from the FIR Filter). The outputs are the new weights where $\nabla V = \nabla V_1 + \nabla V_2 + \ldots + \nabla V_n$ and the weights are updated using Eq. 2.

Going back to Figure 2, the software version of this algorithm was quadratically dependent upon the window size, making it impossible to obtain fast and smooth convergence of the weights to their optimal solution without incurring a large time penalty. By transforming the relationship between computation time and window size to a linear relationship, we can now better satisfy the tradeoff between filter performance and total computation time, as explored in the next section.

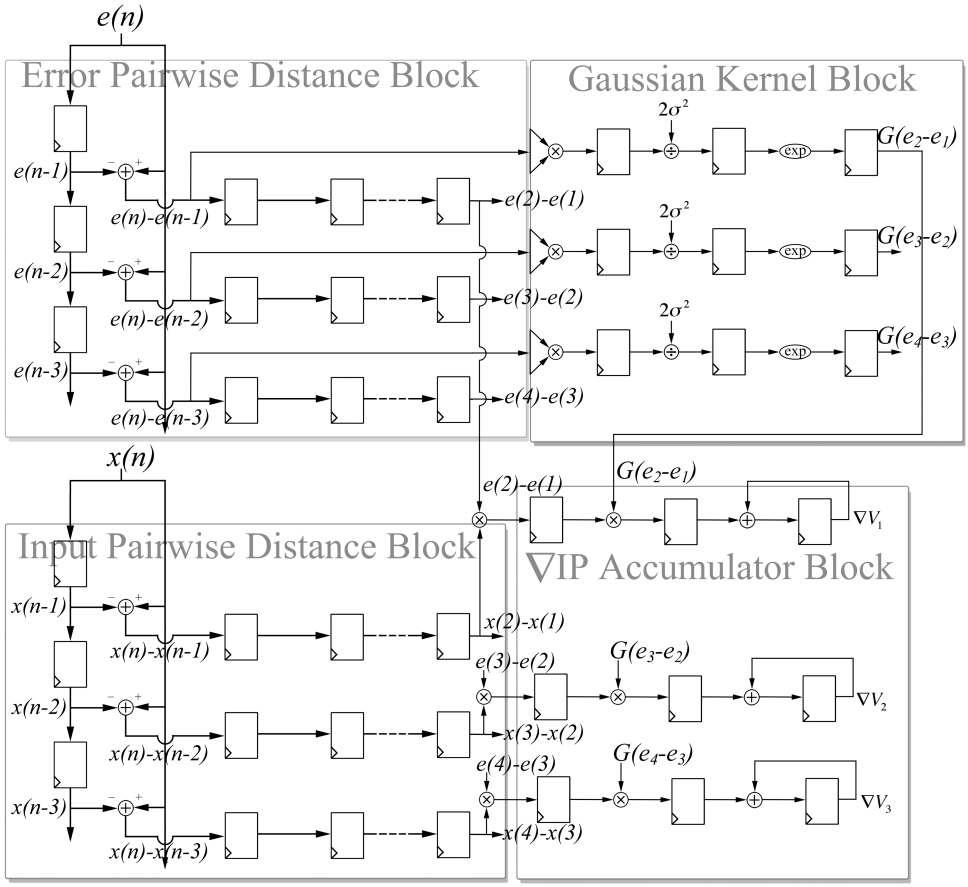

Figure 9. Overall architecture of MEE cost function

IV RESULTS AND ANALYSIS

The platform used to test the performance and scalability of this parallel architecture is the Novo-G reconfigurable supercomputer. Housed in the NSF Center for High-Performance Reconfigurable Computing (CHREC) at the University of Florida, Novo-G currently consists of 48 GiDEL PROCStar-III quad-FPGA boards. Each of these 192 FPGAs is an Altera Stratix-III E260 device featuring 768 18×18 multipliers and 256K logic elements, with 4.25GB of dedicated memory in three parallel banks directly attached. For more detailed information on Novo-G, see [20].

Software baselines used for comparison in this section are coded in C, compiled using GCC with optimization –O4, and executed on an AMD 2.4 GHz Opteron with 4GB of DDR400 RAM. The AF design was tested in real-time to compare its performance versus software. In doing so, we address two major challenges:

1. How does the hardware design compare to the software implementation in terms of precision over multiple iterations. Does the design converge to the optimal weights in the same number of steps?
2. How much speedup does the hardware architecture achieve over the software implementation?

A) Precision Results and Analysis

One of the most popular applications of AFs is system identification. Given an unknown system, the AF can approximate its transfer function in a finite number of iterations. Figure 1 back in Section 2 shows the usual setup of this experiment. A filter with set coefficients will provide the desired signal. A sequence of 2000 samples consisting of white Gaussian noise is inputted to both the fixed filter (i.e. unknown system) and to the AF. The weights are adapted for 2000 iterations using the MEE criterion. By tracking the weight changes over time, we can determine how fast and how accurately they converge to the optimum values, while also comparing these results with those from the software implementation. When the algorithm uses higher computational precision, a fewer number of iterations is needed to reach the optimal solution.

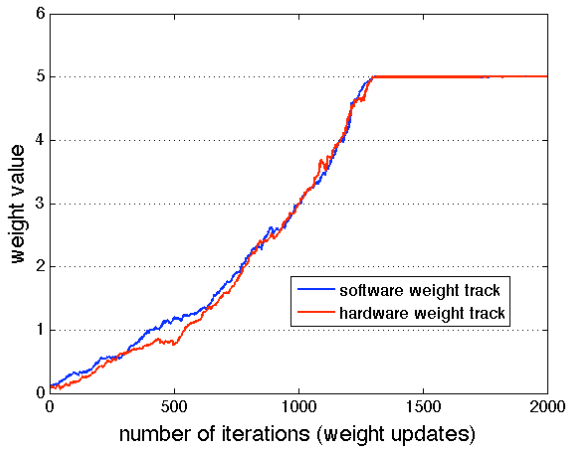

Figure 10. Convergence of adaptive filter weight to same value as weight of observed plant

Figure 10 tracks the value of one weight over 2000 iterations for both hardware and software. The results are very similar with both weights converging to the optimal value (5). More importantly, the convergence takes the same number of iterations, demonstrating that our hybrid (float/fixed) 32-bit architecture maintains identical precision to software throughout the entire adaptation process.

In all the experiments the order of the FIR is set to 10 while the window size varies (IP gradient is evaluated over the window size).

B. Performance Analysis of a Single AF

When evaluating speedup, the parameter that plays a crucial role is the size of the window over which the IP gradient is evaluated (Eq. 4). This parameter represents the number of input samples used to compute a weight update (Eq. 2). The window size influences execution time by increasing the number of computations within the weight feedback loop (Eq. 2). Figure 11 plots the execution time in hardware of one weight update versus varying window sizes.

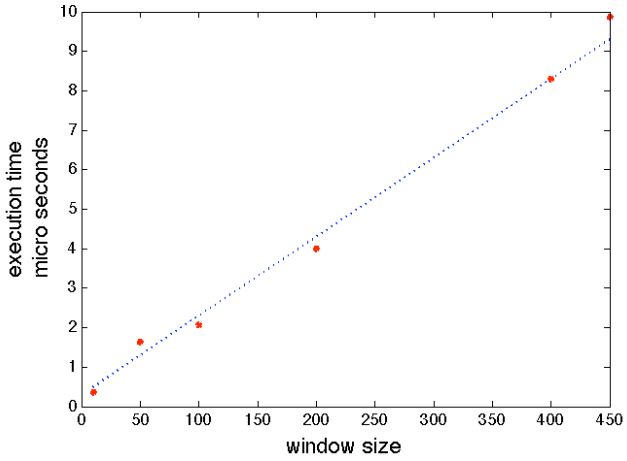

Figure 11. Execution time vs. window size

The dependence between window size and execution time is linear. This result is extremely important because, as shown back in Figure 2, execution time in software is quadratically dependent upon window size and requires significantly more time. We have provided a reconfigurable design that takes advantage of the inherent parallelism within the MEE algorithm, transforming the asymptotic time complexity from $O(N^2)$ to $O(N)$. This outcome has major significance for the field of AFs because the MEE cost function, which has been proven to achieve better results than conventional MSE algorithms, can now be implemented in real-time for signals that require sampling frequencies on the range of 400 kHz. However, as shown later, increasing the window size provides an increase in speedup only up to a point, beyond which the speedup will decline. The reason is the exponential dependence between the window size and the FPGA resources required (logic cells). Table 1 reports the percentage of hardware resources consumed by one AF as the filter's window size is increased.

TABLE I. FRACTION OF LOGIC CELLS UTILIZED ON FPGA (STRATIX-III E260) FOR VARIOUS WINDOW SIZES

AF window size	10	50	100	200	400	450
Fraction of total logic cells used	<1%	<1%	5%	23%	49%	99%

The consequence of this exponential growth in resource requirements for increasing window sizes is that fewer AFs can fit onto one FPGA. The most critical resource utilized by our design for this particular FPGA is logic cells (99%) followed by DSP block elements (65%). Table 2 shows the number of AFs that can fit onto one Stratix-III E260 FPGA as the AF window size is increased. The number of AF drops off drastically for window sizes greater than 100.

TABLE 2. MAXIMUM NUMBER OF AFS PER FPGA (STRATIX-III E260) FOR VARIOUS WINDOW SIZES

AF window size	50	100	200	400	450
Max. number of AFs	243	20	6	2	1

Thus, it is misleading to evaluate speedup by simply assuming that for an increase in window size, the number of computations inside the feedback loop will also increase, leading to considerable speedup (since we have proven that hardware time complexity is $O(N)$ while software is $O(N^2)$). For infinite resources this assertion would be true, but of course in reality the resources required to keep up with a growing window size also increase exponentially.

Figure 12 shows how speedup varies with an increase in window size, where peak speedup is achieved around a window size of 100. For this particular window size (100), one iteration (a weight update) in software is executed in 0.668 ms. As a comparison, the hardware implementation runs one iteration (a weight update using only one AF) in 2.30 μs achieving a speedup of **290**. What is more revealing is the difference in frequency at which the hardware design and the software implementation can sample a given signal. The maximum sampling frequency for software is 1.497 kHz, which means that any speech filtering application alongside many important applications are impossible to sample at or above the Nyquist frequency. While the MEE cost function is busy evaluating the next weight it cannot process a new sample any faster that every 0.668 ms. By contrast, the hardware AF design accelerates each iteration and is capable of sampling signals at 435 kHz while employing the MEE algorithm. This achievement broadens the spectrum of signals to which the AF can adapt, making it possible to exploit the superior MEE algorithm for popular applications such as acoustic echo cancellation [22] and denoising speech [23], as well as many neural applications [24] that require higher-order statistics to adapt the filter coefficients.

In summary, the correlation between speedup and window size is a tradeoff of two factors. The window size sets the number of parallelizable computations inside the feedback loop and also the quantity of hardware resources required to map the algorithm in hardware. The number of parallelizable computations inside the feedback loop contributes to the achievable speedup, since our proposed architecture takes advantage of the inherent parallelism. Also the number of filters that can adapt in parallel is an important contributing factor towards speedup. Ideally, the largest speedup is obtained for a window size that is large enough to require many parallelizable computations within the feedback loop but not so large that it requires an excessive amount of FPGA resources that would prevent many filters from adapting in parallel on one FPGA.

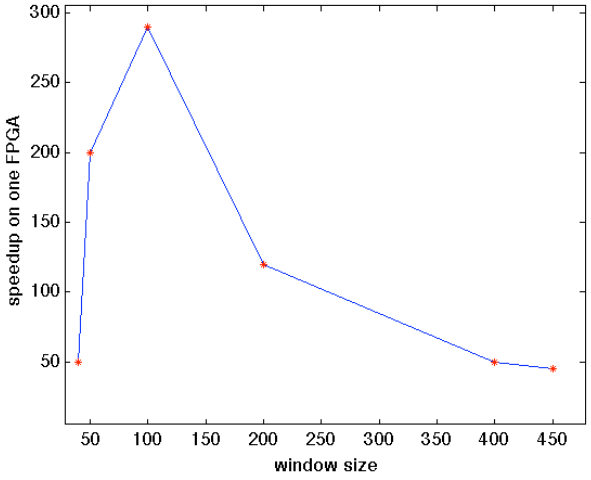

Figure 12. Speedup as a function of window size

The speedup plot in Figure 12 is specific only to this Stratix-III E260 FPGA. For an FPGA with more resources, the speedup peak will shift to the right and higher. Because this design has been written in a reasonably generic form of VHDL, it could be ported to other FPGA devices. For future work it may be interesting to analyze how window size affects speedup for devices with more or less logic cells but such issues are beyond the scope of this paper. Since many applications require the parallel adaptation of multiple independent channels [25], the number of AF functioning in parallel becomes an important factor in deciding whether or not this particular design meets the need of a specific application. For neural applications [26], the number of channels required will often surpass several hundred.

C. *Performance Analysis on Novo-G*

For an AF with a tenth-order FIR and evaluating the IP gradient over 100 samples (window size for batch learning) we were able to fit a total of twenty AFs onto one Stratix-III E260 FPGA. By doing so, we are able to achieve a linear speedup of 290 × 20 = **5800** as compared to the software baseline running on a single CPU core.

By simply replicating this design first on one GiDEL PROCStar-III quad-FPGA board and then on two boards, we

were able to implement a total number of 80 and then 160 AFs capable of adapting independently in parallel. With all eight FPGAs running at the maximum capacity (99% of logic cells utilized), populated by all 160 AFs, the measured speedup rises to **46400**, while the speedup recorded for only one board is exactly half of that (**23200**). The speedup grows linearly because there is very little (i.e. insignificant) overhead from the communication between the host CPU and the FPGAs. The host CPU loads only one sample every clock cycle to the AFs. The only occasion when data is transferred from the FPGAs back to the CPU is when the weights are calculated at the end of the algorithm. Each AF has ten weights and each weight is 32 bits. Compared to the amount of time that elapses for the weights to be calculated, this communication time is negligible.

Figure 13 shows a diagram of the distribution of AFs over one quad-FPGA board. Each AF adapts independently by responding to a batch of 100 samples of white Gaussian noise, over 2000 iterations. Every clock cycle, one sample $x(i)$ of the batch $\overline{X} = \begin{bmatrix} x(1) & x(2) & \dots & x(n) \end{bmatrix}$ is input to each AF. The new weights $\begin{bmatrix} \omega_1 & \omega_2 & \omega_3 & \dots & \omega_9 & \omega_{10} \end{bmatrix}$, computed using Eq. 2, are transferred back to the CPU, which is the equivalent of one iteration.

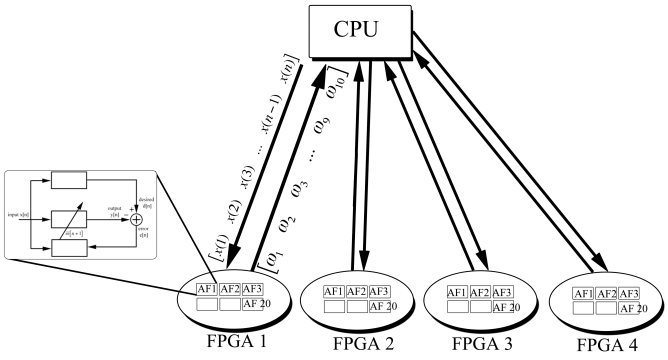

Figure 13. Independent AF blocks adapting in parallel on one quad-FPGA board of Novo-G

For this experiment, only up to two of the 48 quad-FPGA boards in the Novo-G machine were used. Because no off-chip memory was necessary when running the AF design on each FPGA, and because only one sample is input to each AF at every clock cycle, there are no potential bottlenecks expected to arise as more FPGAs are used. Every Novo-G board has its own host CPU core that feeds an input signal to each AF through a simple memory map. There is no node-to-node communication needed for servers in the system, and no transfer of information even between the four FPGAs belonging to the same board; the AFs are embarrassingly parallel with respect to one another. Even though we have demonstrated the efficiency of our hardware design by using a system identification application, we are not constrained to only use Gaussian noise as input to the AF to find the transfer function of an unknown system. There are many applications that require the parallel adaptation of multiple channels [23, 24, 25, 26] to find the optimal weights for each channel.

V Conclusions

This paper proposes a novel parallel architecture that is fine-tuned to the unique needs of the minimum error entropy (MEE) cost function. A hybrid design using both fixed- and floating-point computational blocks maps the MEE cost function onto hardware by taking advantage of the algorithm's dataflow and inherent parallelism. The design minimizes latency without losing precision over an extended number of iterations (2000). The MEE cost function dictates the rules of adaptation for an AF.

For testing the performance of our design, we use a system identification application in which the adaptive weights converge to the optimal solution, and find the transfer function of an unknown plant. The results are compared to a software baseline and prove that the weights of the hardware implementation converge to the same solution obtained by software within the same number of iterations. Furthermore, the asymptotic time complexity of the adaptive algorithm is decreased from $O(N^2)$ in software to $O(N)$ in hardware, providing a linear relationship between performance (convergence time) and execution time.

The most critical factor that influences speedup is the window size (number of input samples) used to compute the weight updates. The window size influences both the number of parallelizable computations within the feedback loop and the hardware resources required to implement the cost function. Larger window sizes will increase the number of parallelizable computations but at the same time require significantly more resources. The result is that less AFs can fit onto one FPGA, a factor that can decrease speedup and make this design undesirable for applications that require many channels to be processed in parallel. We analyze the tradeoff and find the optimum window size (100) for which the maximum speedup is achieved.

Finally, experiments conducted and measurements taken on one board of the Novo-G machine in real-time prove that our architecture obtained significant speedup and is highly scalable (speedup of 290 for one AF, 5800 for one FPGA, 23200 for one quad-FPGA board, and 46400 for two quad-FPGA boards). The most important result is that we can now for the first time target numerous applications that have previously been restricted from using the MEE cost function due to its heavy computational demands.

VI References

[1] Sheng-Fuu Lin; Kumar, P.R.; "Parameter convergence in the stochastic gradient adaptive control law", *Proceedings of the 27th IEEE Conference on Decision and Control*, on 7-9 Dec 1988, Page(s): 1211 - 1212 vol. 2

[2] Zheng-wei; Hu Zhi-Yuan Xie; "Modification of Theoretical Fixed-point LMS Algorithm for Implementation in Hardware", *Second International Symposium on Electronic Commerce and Security,* on 22-24 May 2009, Page(s): 174 – 178 vol. 2

[3] Elliott, R.J.; Krishnamurthy, V.; "Finite dimensional filters for maximum likelihood estimation of continuous-time linear Gaussian systems", *Proceedings of the 36th IEEE Conference on Decision and Control*, on 10-12 Dec 1997, Page(s): 4469 - 4474 vol. 5

[4] Erdogmus, D.; Principe, J.C.; "Generalized Information Potential Criterion for Adaptive System Training", *IEEE Transactions on Neural Networks*, on Sep 2002, Page(s): 1035 – 1044 vol. 13

[5] Erdogmus, D.; Principe, J.C.; "Entropy Minimization Algorithm for Multilayer Perceptrons", *Proceedings from International Joint Conference on Neural Networks* 2001, Page(s): 3003 - 3008 vol. 4

[6] Lok-Kee, Ting; Woods, R.; Cowan, C.F.N.; "Virtex FPGA implementation of a pipelined adaptive LMS predictor for electronic support measures receivers", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, on Jan. 2005, Page(s): 86 – 95 vol. 13

[7] Glentis, G.O.; "Pipelined architectures for the TD-LMS adaptive filter", *IEEE International Conference on Acoustic Speech and Signal Processing*, on 2001, Page(s): 1081 - 1084 vol. 2

[8] Mahfuz, E.; Chunyan Wang; Ahmad, M.O.; "A high-throughput DLMS adaptive algorithm", *IEEE International Symposium on Circuits and Systems*, on 23-26 May 2005, Page(s): 3753 - 3756 Vol. 4

[9] Sizhong Chen; Tong Zhang, "Self-timed dynamically pipelined adaptive signal processing system: a case study of DLMS equalizer for read channel", *IEEE transactions on Circuits and Systems*, on July 2005, Page(s): 1338 – 1347 vol. 52

[10] Fengqi Yu; Willson, A.N., Jr.; "An interleaved/pipelined architecture for adaptive lattice equalizer", *Proceedings of the 43rd IEEE Midwest Symposium on Circuits and Systems*, on 2000, Page(s): 856 - 859 vol. 2

[11] Garcia-Alcantara; V. Rodelllar; V. Gomez-Vilda, P.; "Fixed-point arithmetic trade-offs in adaptive filters for speech recognition", *Electrotechnical Conference, 1998. MELECON, 9th Mediterranean*, on 18-20 May 1998, Page(s): 518 - 521 vol. 1

[12] North, R.C.; Zeidler, J.R.; Ku, W.H.; Albert, T.R.; "A floating-point arithmetic error analysis of direct and indirect coefficient updating techniques for adaptive lattice filters", *Signal Processing, IEEE Transactions on*, on May 1993, Page(s): 1809 – 1823 vol. 41 issue 5

[13] Leon, G.; Jenkins, W.K.; "Adaptive fault tolerant digital filters with single and multiple bit errors in floating-point arithmetic", *The 2000 IEEE International Symposium on Circuits and Systems*, on 2000, Page(s): 630 - 633 vol. 3

[14] Daitx, F.F.; Rosa, V.S.; Costa, E.; Flores, P.; Bampi, S.; "VHDL Generation of Optimized FIR Filters", *2nd International Conference on Signals, Circuits and Systems*, on 7-9 Nov. 2008, Page(s): 1 – 5

[15] Rosa, V.S.; Costa, E.; Bampi, S.; **"**A VHDL Generation Tool for Optimized Parallel FIR Filters", *International Conference on Very Large Scale Integration*, on Oct 2006, Page(s): 134 – 139

[16] Mehendale, M.; Sherlekar, S. D.; Venkatesh, G., "Synthesis of multiplier-less FIR filters with minimum number of additions", *Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, on 1995, Page(s): 668-671

[17] Meher, P.K.; Chandrasekaran, S.; Amira, A.; "FPGA Realization of FIR Filters by Efficient and Flexible Systolization Using Distributed Arithmetic", *IEEE Transactions on Signal Processing,* on July 2008, Page(s): 3009 - 3017 vol. 56, no. 7

[18] Horrocks, D.H.; Bull, D.R.; "Quantization effects in FIR filters: fixed versus floating point", *IEE Colloquium on Electronic Filters*, on 9th June 1989, Page(s): 9/1-9/8

[19] Oliver, T.; Schmidt, B.; Nathan, D.; Clemens, R.; Maskell, D.; "Multiple Sequence Alignment on an FPGA", *Proceedings of the 2005 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, on 22-22 July 2005, Page(s): 326 – 330, vol. 2

[20] Novo-G architecture overview, www.chrec.org/~george/Novo-G.pdf

[21] Mu Cluster overview, http://www.hcs.ufl.edu/lab/mu.php

[22] Chhetri, A.S.; Stokes, J.W.; Florencio, D.A.; "Acoustic Echo Cancelation for High Noise Environments", *2006 IEEE International Conference on Multimedia and Expo*, on 9-12 July 2006, Page(s): 905 – 908

[23] Shuqi Wang; Yin Shi; "An Improved Speech Denoising Algorithm Based on Adaptive Least Mean Square", *International Conference on Industrial and Information Systems*, on 24-25 April 2009, Page(s): 293 – 296

[24] Yunfeng, Wu; Rangayyan, R.M.; Sin-Chun, Ng; "Cancellation of Artifacts in ECG Signals Using a Normalized Adaptive Neural Filter", *29th annual International Conference of the IEEE Engineering in Medicine and Biology Society*, on 22-26 Aug. 2007, Page(s): 2552 – 2555

[25] Mozipo, A.L.T.; Massicotte, D.; Quinton, P.; Risset, T.; "A parallel architecture for adaptive channel equalization based on Kalman filter using MMAlpha", *IEEE Canadian Conference on Electrical and Computer Engineering*, on 12th May 1999, Page(s): 554 -559 vol. 1

[26] Xuedong Chen; Ou Bai; "Towards multi-dimensional robotic control via noninvasive brain-computer interface", *International Conference on Complex Medical Engineering*, on 9 - 11 April 2009, Page(s): 1 - 5