FPGA Implementation of Adaptive Filters based on GSFAP using Log Arithmetic

Milan Tichy and Jan Schier
Institute of Information Theory and Automation
Academy of Sciences of the Czech Republic
Prague, Czech Republic
Email: {tichy,schier}@utia.cas.cz

David Gregg
University of Dublin
Trinity College
Dublin, Ireland
Email: David.Gregg@cs.tcd.ie

Abstract— **Adaptive filters are used in many applications of digital signal processing. Digital communications and digital video broadcasting are just two examples. The GSFAP algorithm, discussed in the paper, is characterized by convergence superior to the popular NLMS, with only slightly higher complexity. The paper deals with floating-point-like implementation of algorithm using FPGA hardware. We present an optimized core for the GSFAP, built using logarithmic arithmetic which provides very low cost multiplication and division. The design is crafted to make efficient use of the pipelined logarithmic addition units. The resulting GSFAP core can be clocked at more than 80 MHz on the one million gate Xilinx XC2V1000-4 device. It can be used to implement filters of orders 20 to 1000 with a sampling rate exceeding 50 kHz. For comparison, we implemented a similar NLMS core and found that although it is slightly smaller than the GSFAP core and it allows a higher signal sampling rate (around 70 kHz) for the corresponding filter orders, GSFAP has adaptation properties that are much superior to NLMS, and that our core can provide very sophisticated adaptive filtering capabilities for resource-constrained embedded systems.**

I. Introduction

Adaptive filters are widely used in digital signal processing (DSP) for countless applications in telecommunications, digital broadcasting, etc.

While a wide variety of filtering algorithms have been proposed in the literature, the most important categories are perhaps those based on least mean squares (LMS) [1] (with the power-normalized sibling NLMS [2]) and recursive least squares (RLS) [3], [2]. Algorithms based on LMS are fast and simple to implement, but they suffer from slow convergence. The RLS-based algorithms converge faster, but they are usually considered too computationally expensive, particularly in applications like echo cancellation where filters with up to several hundred taps are required. More recently, a category of algorithms based on *affine projection* (AP) [4], sometimes referred to as the *generalized NLMS*, have been developed, which provide some compromise between the slow convergence of LMS and the computational complexity of RLS.

In order to reduce the computational complexity of the original affine projection algorithm (APA) [4], a fast version (FAP) [5] was developed. Although the FAP algorithm converges quickly and has low computational requirements, it is actually rather unpromising because it is numerically unstable, especially for non-stationary signals. This is due to the use of fast RLS (FRLS) [6], [7] in the algorithm. A number of variants have been proposed which solve the numerical stability problems while maintaining the advantages of FAP. The "modified" FAP [8], [9] uses the matrix inversion lemma employed in the classical RLS algorithm, thus avoiding the problems with fast RLS but at the cost of greater computational requirements. Conjugate gradient (CG) FAP [10] builds on the results of the modified FAP, and uses the conjugate gradient method [11] to deal with the matrix inversion. The FAP-based algorithm that we believe to be the most suitable for hardware implementation is the Gauss-Seidel (GS) FAP [12] which replaces the CG method with the Gauss-Seidel method [13]. This algorithm has all the advantages of modified FAP and CGFAP, but has lower computational complexity, allowing an efficient implementation with fewer hardware resources.

The complexity of LMS based algorithms is $\mathcal{O}(L)$, typically $2L+1$ multiply-accumulate (MAC) operations per iteration, where L is the filter order. The complexity of NLMS is similar, $2L+3$ MAC operations and 1 division per iteration. On the contrary, the complexity of the RLS-based algorithms is $\mathcal{O}(L^2)$. The memory requirements of RLS are also much higher than those of (N)LMS. Fast versions of the RLS algorithm with complexity $\mathcal{O}(L)$ exist, which partly solve the complexity issues. The RLS lattice algorithm requires $18L$ operations and the Fast Transversal Filter (FTF) requires $8L$ operations, or $9L$ in the stabilized form. This however comes at the expense of problems with numerical stability. The complexity of FAP is $2L+\mathcal{O}(N)$ where N is referred to as the projection order. All other FAP variants mentioned above have complexity $2L+\mathcal{O}(N^2)$ where GSFAP is the most efficient, its complexity being $2L+N^2+4N-1$ MAC operations and N divisions per sample period. The projection order is almost always very much smaller than the filter order, i.e. $N \ll L$, so the time complexity is usually dominated by L rather than N^2.

In this paper we develop an optimized core for FPGA implementation of the GSFAP [12] algorithm. To reduce the resource requirements of the floating-point computations, we represent numbers using the logarithmic number system (LNS) [14], [15]. To evaluate the design, a configurable 1000-

1-4244-0383-9/06/$20.00 ©2006 IEEE

0) Initialization:

$$\mathbb{R}_{-1} = \delta\,\mathbb{I}, \quad \mathbf{p}_{-1} = \frac{1}{\delta}\,\mathbf{b}, \quad \mathbf{b} = \begin{bmatrix}1 & 0 & \dots & 0\end{bmatrix}^{\mathrm{T}}$$

$$\boldsymbol{\epsilon}_{-1} = \mathbf{0}, \quad \hat{\mathbf{w}}_{-1} = \mathbf{0}, \quad \mathbf{u}_{k<0} = \mathbf{0}, \quad \boldsymbol{\xi}_{k<0} = \mathbf{0}$$

1) Update $\mathbb{R}_k$:

a) $u_k \dashrightarrow \mathbf{u}_k,\ \boldsymbol{\xi}_k$

b) $\mathbb{R}_k = \mathbb{R}_{k-1} + \boldsymbol{\xi}_k\boldsymbol{\xi}_k^{\mathrm{T}} - \boldsymbol{\xi}_{k-L}\boldsymbol{\xi}_{k-L}^{\mathrm{T}}$

2) Update $\mathbf{p}_k$ (Gauss-Seidel iteration):

for $(i = 0;\ i < N;\ i = i + 1)$

$$p_{i,k} = \left[b_i - \sum_{i>j}^{N} R_{ij,k}p_{j,k} - \sum_{i<j}^{N} R_{ij,k}p_{j,k-1} \right] / \; R_{ii,k}$$

end;

3) Compute e_k:

a) $y_k = \mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1} + \mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$

b) $e_k = d_k - y_k$

4) Update $\boldsymbol{\epsilon}_k$ and $\hat{\mathbf{w}}_k$:

a) $\boldsymbol{\epsilon}_k = e_k\mathbf{p}_k + \begin{bmatrix}0 \\ \bar{\boldsymbol{\epsilon}}_{k-1}\end{bmatrix}$

b) $\hat{\mathbf{w}}_k = \hat{\mathbf{w}}_{k-1} + \mu\mathbf{u}_{k-N+1}\epsilon_{N-1,k}$

Fig. 1. The Gauss-Seidel Fast Affine Projection (GSFAP) algorithm.

tap adaptive filter was implemented. We compare the GSFAP core with similar NLMS core and show that although the NLMS core is slightly smaller than the GSFAP core and it allows a higher signal sampling rate (around 70 kHz comparing to 50 kHz for GSFAP) for the corresponding filter orders, GSFAP has adaptation properties that are much superior to NLMS, and that our core can provide very sophisticated adaptive filtering capabilities for resource-constrained embedded systems.

II. The GSFAP Algorithm

A brief review of the GSFAP algorithm is now given. The GSFAP algorithm is summarized in Fig. 1.

The *filter order*, i.e. the number of filter coefficients, is denoted as L. Individual coefficients (weights) constitute the *coefficient vector* $\mathbf{w}_k$. Instead of updating the actual (true) filter coefficients $\mathbf{w}_k$, the variants of the FAP algorithms discussed in this text use more efficient alternate coefficient update. The *alternate coefficient vector* is referred to as $\hat{\mathbf{w}}_k$ and has the same number of elements. The number N is referred to as the *projection order*.

The key variables are initialized in step 0. Steps 1 through 4 represent one iteration of the GSFAP algorithm.

The update of an $N \times N$ matrix $\mathbb{R}$ at time k is performed in step 1. The matrix $\mathbb{R}_k$ is the *autocorrelation matrix* of the excitation signal and it is symmetric and positive definite. The vector $\boldsymbol{\xi}_k$ consists of the first N elements of the vector $\mathbf{u}_k$ which is called *excitation* (input) *signal vector*. The excitation signal vector consists of the delayed sequence of L input samples. The vector $\boldsymbol{\xi}_{k-L}$ has the same structure as $\boldsymbol{\xi}_k$ but it represents the state at time $k - L$.

In step 2, the vector $\mathbf{p}_k$ is calculated. This vector is in fact the first column of the inverse of matrix $\mathbb{R}_k$. This problem is equivalent to solving a set of linear equations $\mathbb{R}_k\mathbf{p}_k = \mathbf{b}$, where $\mathbf{b}$ is a vector of length N, in which all elements are zero, except for the first element which has the value one. As shown in [12], one iteration of the Gauss-Seidel method provides a good estimate of the actual vector $\mathbf{p}_k$ if $\mathbf{p}_{k-1}$ is used as the initial value for each GS iteration. The symbol $R_{ij,k}$ denotes the ij-th element of the matrix $\mathbb{R}_k$ and the symbol $p_{i,k}$ denotes the i-th element of the vector $\mathbf{p}_k$.

An efficient method to calculate the filter output y_k and the estimation error e_k using the alternate coefficient vector $\hat{\mathbf{w}}_k$ rather than the original weight vector $\mathbf{w}_k$ is shown in step 3. Vector $\bar{\boldsymbol{\epsilon}}$ is called the *normalized estimation error* vector and is of dimension N. The symbol $\bar{\boldsymbol{\epsilon}}_{k-1}$ denotes a vector consisting of the $N - 1$ uppermost elements of vector $\boldsymbol{\epsilon}_{k-1}$ and the symbol $\tilde{\mathbf{R}}_{0,k}$ represents a vector that consists of the $N - 1$ lowermost elements of the first (left) column of the matrix $\mathbb{R}_k$.

In step 4, the normalized estimation error vector $\boldsymbol{\epsilon}_k$ and consequently the alternate coefficient vector $\hat{\mathbf{w}}_k$ are updated. The "old" excitation signal vector $\mathbf{u}_{k-N+1}$ and the lowermost element of the newly updated vector $\boldsymbol{\epsilon}_k$ denoted as $\epsilon_{N-1,k}$ are used to update the alternate coefficient vector $\hat{\mathbf{w}}_k$.

The parameter μ is the relaxation factor which represents the algorithm step-size parameter. The algorithm is stable for $0 < \mu < 2$. The parameter δ is the regularization parameter that prevents the correlation matrix $\mathbb{R}$ from becoming singular. It is usually set to $10^{-7} < \delta < 1$ depending on the input signal.

III. Logarithmic Arithmetic

In order to maintain accuracy of the algorithm in the FPGA implementation, we decided to implement the computations using a floating-point-like logarithmic arithmetic. The parameters of the library are briefly presented in this section.

The *logarithmic number system* (LNS) was chosen in order to reduce resource requirements and to achieve short latency as compared to other floating-point solutions. Logarithmic multiplication and division require only very simple logic. Although addition and subtraction are more complex in LNS, recent advances have made them feasible in small FPGA devices. We use the High Speed Logarithmic Arithmetic (HSLA) cores, described in [15].

Table I shows the resource requirements of our LNS units in comparison to Underwood's [16] highly-optimized IEEE single-precision floating point units. The major disadvantage of LNS arithmetic is the number of Block RAMs used by the ADD/SUB unit for storing the look-up tables. These units are always instantiated in pairs. While the resource requirements for a pair of LNS ADD/SUB pipes is significantly higher than for a pair of the floating point cores, LNS multiplier units need a small fraction of the size of the floating point multipliers. The most common operations in many matrix algorithms are multiplication and addition. When we sum the

TABLE I

COMPARISON OF 32-BIT LNS AND FLOATING-POINT UNITS

Unit	FFs	LUTs	Slices	MULTs	BRAMs
Floating point					
Adder	696	611	496	0	0
2-pipe adder	1392	1222	992	0	0
Multiplier	821	722	598	4	0
Divider	2476	2220	1929	0	0
LNS 32-bit					
2-pipe adder	1702	2135	1648	8	28
Multiplier	35	139	83	0	0
Divider	35	145	82	0	0

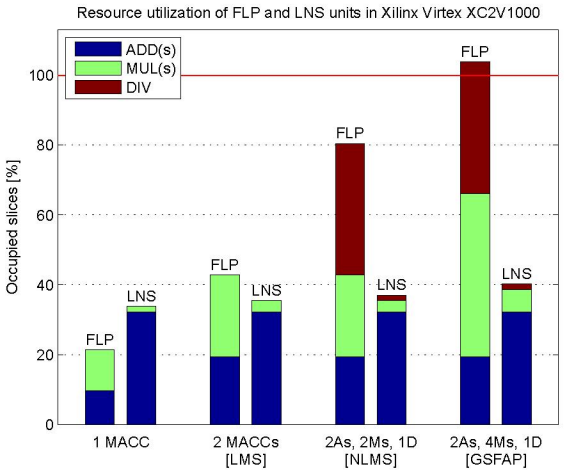

Fig. 2. Comparison of the LNS and Underwood's arithmetics resource utilization for single MACC unit and the LMS, NLMS and GSFAP algorithms.

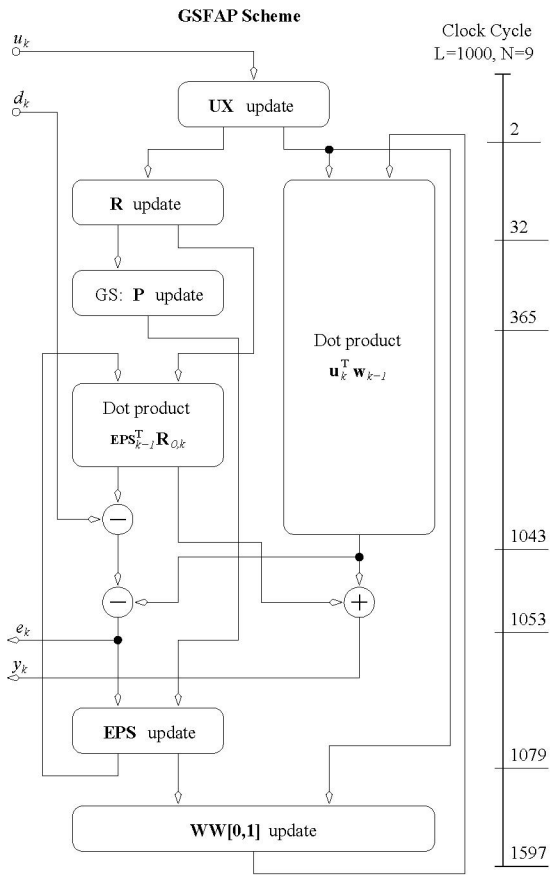

Fig. 3. Block diagram of top-level architecture of the GSFAP core.

resources required by two multiply-add pipes, we see that the LNS units require fewer resources (except for Block RAMs). Since many DSP algorithms require division or square root the advantage of the LNS architecture is evident. Another important issue is the clock speed and latencies. Underwood's adder can be clocked at up to 165 MHz on the XC2V6000-5 FPGA, with a latency of up to 13 cycles, the multiplier at 125 MHz with a latency of 16 cycles, whereas the divider at 105 MHz, but with a latency of 37 cycles. In contrast, the latencies of the LNS adder, multiplier and divider are 8, 1, and 1 cycles respectively, at a clock rate over 80 MHz on XC2V6000-4 FPGA. The XC2V6000-4 is perhaps 10% slower than the XC2V6000-5, so we could reasonably expect the LNS units to reach 90 MHz on the latter. From this we can conclude that if latency is important, the LNS cores give considerable advantage.

In Figure 2, a comparison between the resource utilization of the LNS and Underwood's arithmetics for a single MACC unit and for the implementation of the LMS, NLMS and GSFAP algorithms is given, using the Xilinx Virtex XC2V1000 device. As we will see in Section IV, for efficient implementation of GSFAP 2 add, 4 multiply and 1 division units have to be used. The figure clearly demonstrates the advantages of using LNS arithmetic rather than the classical floating point. It should be mentioned that the LNS ADD/SUB unit disadvantage of using a considerable amount of Block RAMs is not an issue in our case since algorithms' internal structures (input and weight vectors, correlation matrix, etc.) can easily fit into remaining Block RAMs.

IV. ARCHITECTURE

In this section we present architecture of our GSFAP core. Both the mapping of the algorithm onto the LNS arithmetic units and of the data structures to the Block RAMs is described. The algorithm employs one LNS addition/subtraction (ADD/SUB **A** and **B** — two separate, parallel pipelines) unit, four LNS multipliers (MUL **A**, **B**, **C**, and **D**) and one LNS divider (DIV **A**). Non-scalar data structures, vectors and a matrix, are stored in Virtex-2 Block RAMs.

The top-level architecture of the GSFAP algorithm is depicted in Fig. 3. The blocks in the diagram correspond to individual steps in the algorithm presented in Fig. 1. The blocks and operations depicted on the left-hand side of the diagram employ the first ADD/SUB pipeline (pipeline **A**) while the operations on the right-hand side employ the second pipeline of the ADD/SUB unit (pipeline **B**). The block denoted as "**UX** update" does not use any ADD/SUB units; the block denoted as "**WW[0,1]** update" utilizes both pipelines. The time line on the far right side of the diagram indicates the clock

cycles during which different parts of the design are active in the GSFAP based filter with the parameters $L = 1000$ (filter order) and $N = 9$ (projection order).

The sequence of $L+N$ input signal samples is stored in the Block RAM referred to as **UX** (vector). This data sequence constitutes the excitation signal vector $\mathbf{u}_k$ and the vector $\boldsymbol{\xi}_{k-L}$. The vectors $\boldsymbol{\xi}_k$ and $\mathbf{u}_{k-N+1}$ overlap the first N elements and the last L elements of **UX**, respectively. The **UX** vector storage is implemented as a circular buffer. We keep the state of the buffer in the register `UX_state` that is of the same width as the number of the **UX** address lines. To update the **UX** vector, we just need to decrement the register `UX_state` and to write the new data sample u_k to **UX**[`UX_state`].

As mentioned in Section II, $\mathbb{R}$ is a symmetric matrix. It is stored in the Block RAM **R**. We store the whole matrix, not just the upper triangle, even though this almost doubles the memory requirements for this matrix. The reasons are:

1) The dimensions of $\mathbb{R}$ are the same as the projection order, which is reasonably small. Experiments show that using projection order higher than 20 does not improve algorithm performance, i.e. convergence and tracking properties, significantly. A matrix with dimensions of up to $N = 22$ will fit in a single Virtex-2 Block RAM.
2) The update of the matrix $\mathbb{R}$ can be implemented very efficiently using "re-indexing" (see below).
3) Control logic necessary to calculate addresses of individual elements of the matrix is simpler and faster.

To update the matrix $\mathbb{R}$, we need to calculate

$$\mathbb{R}_k = \mathbb{R}_{k-1} + \boldsymbol{\xi}_k\boldsymbol{\xi}_k^{\mathrm{T}} - \boldsymbol{\xi}_{k-L}\boldsymbol{\xi}_{k-L}^{\mathrm{T}} \tag{1}$$

This operation involves N^2 multiply and accumulate operations. The Block RAM **R** is accessed $2N^2$ times (reading from and writing to **R**). However, it can be shown that we can get the same result at much lower cost by using the following procedure. Let us define the *correlation vector*

$$\mathbf{r}_k = \begin{bmatrix} r_{0,k} & r_{1,k} & \dots & r_{N-1,k} \end{bmatrix}^{\mathrm{T}} \tag{2}$$

which can be updated in each iteration as follows

$$\mathbf{r}_k = \mathbf{r}_{k-1} + \xi_{0,k}\boldsymbol{\xi}_k - \xi_{0,k-L}\boldsymbol{\xi}_{k-L} \tag{3}$$

where $\xi_{0,k}$ and $\xi_{0,k-L}$ are the first (uppermost) elements of the vectors $\boldsymbol{\xi}_k$ and $\boldsymbol{\xi}_{k-L}$, respectively, and the vector $\mathbf{r}_{k-1}$ is the leftmost column of the matrix $\mathbb{R}_{k-1}$. Then the matrix $\mathbb{R}_k$ can be constructed so that each element of the original matrix $\mathbb{R}_{k-1}$ is shifted diagonally by 1 and the vector $\mathbf{r}_k$ is used as the first row/column of the matrix $\mathbb{R}_k$. The procedure requires $2N$ multiply and accumulate operations and the Block RAM has to be accessed N times for reading and $2N-1$ times for updating. This operation is schematically depicted in Fig. 4.

Our hardware implementation of $\mathbb{R}$ and its update is similar to using a circular buffer. We call this process "re-indexing". The actual state of the buffer is kept in the register `R_state` which is decremented by the value $N+1$ in each iteration. The "**R** update" is implemented in the following way. The values of vector $\boldsymbol{\xi}_k$ are read from **UX** and the multiplication $\xi_{0,k}\boldsymbol{\xi}_k$

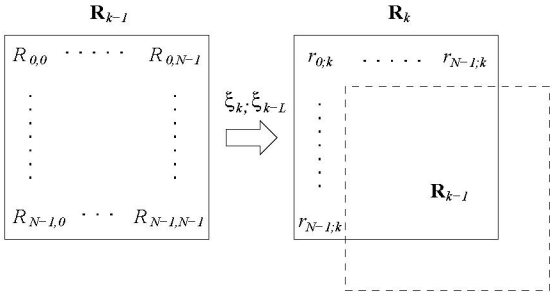

Fig. 4. Updating matrix $\mathbb{R}_k$ by shifting the original matrix diagonally.

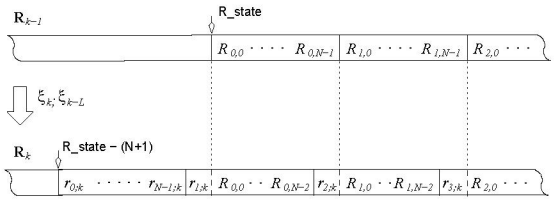

Fig. 5. Implementing diagonal shifting of matrix $\mathbb{R}_k$ by "re-indexing".

using MUL **B** unit is performed while another "process" reads values from **R** to get the values of vector $\mathbf{r}_{k-1}$. The results of the multiplication are then added to the values read from **R**. After the **R** has been read the value of `R_state` is updated (decremented by $N+1$). When the adder pipe is freed the values of $\boldsymbol{\xi}_{k-L}$ are read from **UX** and the multiplication $\xi_{0,k-L}\boldsymbol{\xi}_{k-L}$ is performed. Results of this multiplication are subtracted from results of the "adding" process. Results of the subtraction, forming the vector $\mathbf{r}_k$, are written to both ports of Block RAM **R** to positions representing the first row/column of the matrix $\mathbb{R}_k$. All operations are pipelined, so no storage for the intermediate results or vector $\mathbf{r}_k$ is necessary. For a matrix of order 9 this operation takes only 30 clock cycles. This is dramatically faster than the simple approach, which involves N^2 multiply and accumulate operations. The mechanism of "re-indexing" of matrix $\mathbb{R}$ in Block RAM **R** is depicted in Fig. 5.

One of the key modules of the algorithm is the Gauss-Seidel (GS) solver used to calculate the vector $\mathbf{p}_k$ (step 2 of the algorithm in Fig. 1). Each element $p_{i,k}$ of the vector $\mathbf{p}_k$ depends on previously computed elements, $p_{0\dots i-1,k}$ and $p_{i+1\dots N-1,k-i}$. Therefore the individual elements cannot be updated simultaneously. Instead we use a pipelined architecture, in which the computations using the available values of $\mathbf{p}_k$ are performed first, and the value that is dependent on the previous iteration is performed last. Fig. 6(a) depicts the hardware architecture that calculates one step, that is one element of vector $\mathbf{p}_k$. This operation is performed N times to update the whole vector $\mathbf{p}$. The computation of next element can start after the previous value has been written to the Block RAM referred to as **PEPS**. We use this Block RAM to store both vector $\mathbf{p}$ and ϵ. In order to minimize the cycle count, we use two MUL (**A** and **B**) units and both ports of Block RAMs **R** and **PEPS**. The multiplication results are summed

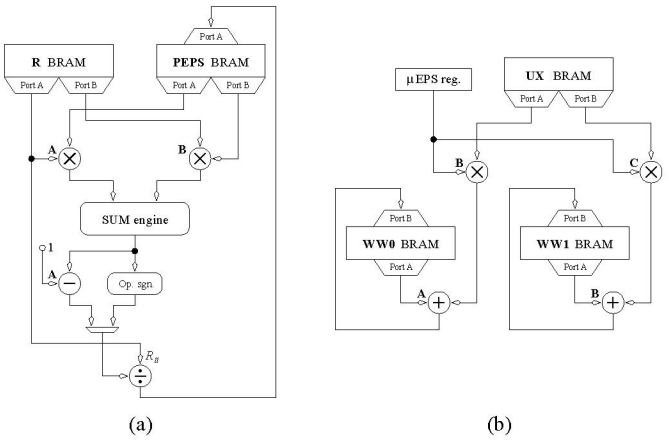

Fig. 6. Architecture schematic of one step of the update of vector $\mathbf{p}$ stored in the Block RAM **PEPS** (a) and of the update of the alternate coefficient vector $\hat{\mathbf{w}}$ stored in the Block RAMs **WW0** and **WW1** (b).

and the result forms dot product of two $N-1$ length vectors. Recall that vector $\mathbf{b}$ has the value 1 in its first element, and all other elements have the value 0. Because of this, we need to subtract the value from 1 on the first step, but in all other steps we are subtracting from zero, which we can implement by simply negating the sign bit. Considering the ADD/SUB unit latency and that the negation of the sign bit costs virtually nothing, we save $9*(N-1)$ clock cycles for complete "**P** update". The result is then divided by a corresponding diagonal element of matrix $\mathbb{R}$ and the resulting value is finally written to the **PEPS** Block RAM. It is important to recall that the LNS multiplication and division are fast and cheap, so the resulting hardware is highly efficient.

The next step of the algorithm is to compute the filter output y_k and the estimation error e_k as shown in the step 3 in Fig. 1. The dot product $\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$ is calculated, just after the vector $\mathbf{p}$ has been updated, using the port **A** of the ADD/SUB unit and the MUL **A** and **B** units. The resulting value is then multiplied by scalar μ in order to get the value $\mu\bar{\boldsymbol{\epsilon}}_{k-1}^{\mathrm{T}}\tilde{\mathbf{R}}_{0,k}$. As depicted in Fig. 3, the "long" dot product (of two vectors of length given by the filter order L) $\mathbf{u}_k^{\mathrm{T}}\hat{\mathbf{w}}_{k-1}$ is calculated in parallel with previously described blocks. It employs the MUL **D** and the ADD/SUB **B** units.

The last two steps of GSFAP are the "**EPS** update" and the "**WW[0,1]** update". They both have similar structure so only the update of the vector $\hat{\mathbf{w}}$ is described. The hardware structure of this stage is shown in Fig. 6(b). In order to minimize the latency of the GSFAP core we split the vector $\hat{\mathbf{w}}$ into two vectors which are stored in separate Block RAMs **WW0** and **WW1**. This allows us to use two independent pipelines to update them as depicted in the figure. This stage utilizes both pipelines of the ADD/SUB unit and the MUL **B** and **C** units. At the end of this step one GSFAP iteration is complete and the core is ready to acquire new data samples.

V. Results and Conclusions

We have created separate GSFAP cores for LNS 32- and 19-bit precision. The parameters of the cores are fully configurable. It is possible to change both filter order L (for values $20 \leq L \leq 1000$) and filter parameters μ and δ. However, modifying the value of N requires minor architectural changes. To demonstrate performance, we fixed the parameters $L = 1000$ and $N = 9$. In this configuration, a full iteration of the GSFAP algorithm takes 1597 cycles and performs 4227 logarithmic operations (2.64 ops/cycle). For comparison we have also created similar cores that implement the NLMS algorithm, which also come in 32- and 19-bit variations and are fully configurable. NLMS is more regular and less computationally intensive than GSFAP. With the corresponding filter parameters, it can perform a full iteration in 1088 clock cycles, and performs 4008 logarithmic operations (3.68 ops/cycle).

Our cores were developed on a Xilinx XC2V6000-4 (6-million gate, speed grade 4) FPGA and on a much smaller Xilinx XC2V1000-4 (1-million gate, speed grade 4) device. Table II shows parameters of developed cores. All designs can be clocked at 80 MHz. At this clock speed, the GSFAP and NLMS designs perform over 210M and 294M log-operations per second, respectively (the M log-operations are equivalent to MFLOPS). The GSFAP 32-bit core occupies only a small fraction (14%) of the 6-million gate XC2V6000-4 device. On the 1-million gate XC2V1000-4, it uses very large percentage of available resources—in particular, it consumes 99% slices. For the maximum filter length ($L = 1000$), the filter is able to perform noise/echo cancellation on signals at a sampling rate of more than 50 kHz. The corresponding NLMS core is some 15% smaller and can operate on signals at a sampling rate of around 73 kHz. The FPGAs used to test the implementations are speed grade 4 devices. More expensive speed grade 6 devices would allow even faster clock speeds, in the order of 100 MHz, while the Virtex-4 devices would be even faster.

To demonstrate a practical application of adaptive filters, we developed an *echo cancellation* example. In this case the adaptive filter is used to suppress echo generated by the unknown system, typically a room or a car cabin. The real room impulse responses and speech signals have been used in our experiments. The results of using the LNS 32-bit implementation of NLMS and GSFAP algorithms for this task are shown in Figure 7. We used the input and echo signals sampled at the rate 16 kHz, the adaptive filters of length $L = 500$ and the GSFAP projection order $N = 9$. The step-size parameter was chosen $\mu = 1$ for both NLMS and GSFAP. The left picture of the figure shows the echo signal to be suppressed and the convergence rates of NLMS and GSFAP. The picture on the right-hand side represents residual echo for both algorithms. It should be noted that the variance (var) of residual echo—which can also be used as a measure of quality of the adaptive algorithm—"left" by the NLMS adaptive filter is 5.79×10^{-4} while for GSFAP it is 6.32×10^{-5}. Although the NLMS core is slightly smaller than the GSFAP and can process signals at higher sampling rates, experiments show that GSFAP has adaptation properties that are much superior to NLMS, and that our core can provide very sophisticated adaptive filtering capabilities for resource-constrained embedded systems.

TABLE II

RESOURCE UTILIZATION OF THE 32- AND 19-BIT LNS GSFAP AND NLMS CORE

	GSFAP								NLMS							
	LNS 32-bit				LNS 19-bit				LNS 32-bit				LNS 19-bit			
	xc2v1000-4		xc2v6000-4		xc2v1000-4		xc2v6000-4		xc2v1000-4		xc2v6000-4		xc2v1000-4		xc2v6000-4	
Slice Flip Flops	4,835	47%	4,496	6%	3,414	33%	3,234	4%	4,408	43%	4,069	6%	3,026	29%	2,846	4%
4 input LUTs	6,049	59%	6,058	8%	4,245	41%	4,294	6%	4,834	47%	4,831	7%	3,301	32%	3,369	4%
Occupied Slices	5,118	99%	4,833	14%	3,538	69%	3,353	9%	4,473	87%	4,160	12%	2,973	58%	2,820	8%
Tbufs	1,280	50%	1,280	7%	192	7%	192	1%	1,280	50%	1,280	7%	192	7%	192	1%
Block RAMs	34	85%	34	23%	12	30%	12	8%	32	80%	32	22%	10	25%	10	6%
MULT18X18s	8	20%	8	5%	8	20%	8	5%	8	20%	8	5%	8	20%	8	5%
Clock rate	80.006 MHz		80.051 MHz		80.502 MHz		80.160 MHz		80.051 MHz		80.058 MHz		80.652 MHz		80.483 MHz	

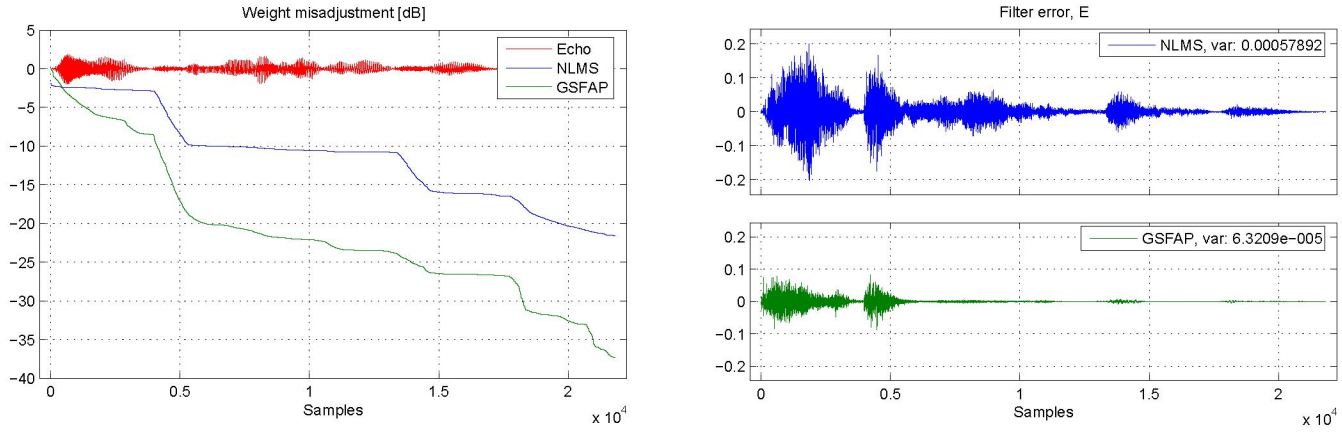

Fig. 7. Comparison of NLMS ans GSFAP within the echo cancellation application using speech signals.

ACKNOWLEDGMENT

This work was supported and funded by the European Commission under the Sixth Framework Programme within the Marie Curie Intra-European Fellowship scheme, Project No. MEIF-CT-2003-502085. The paper reflects only the authors' view and the European Commission is not liable for any use that may be made of the information contained herein.

REFERENCES

[1] B. Widrow and S. D. Stearns, *Adaptive Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall, 1985.

[2] S. Haykin, *Adaptive Filter Theory*, 4th ed. Upper Saddle River, New Jersey: Prentice Hall, 2002.

[3] N. Kalouptsidis and S. Theodoridis, *Adaptative System Identification and Signal Processing Algorithms*. Englewood Cliffs, New Jersey: Prentice Hall, 1993.

[4] K. Ozeki and T. Umeda, "An adaptive filtering algorithm using an orthogonal projection to an affine subspace and its properties," *Electronics and Communications in Japan*, vol. 67-A, no. 5, pp. 126–132, Feb. 1984.

[5] S. L. Gay and S. Tavathia, "The fast affine projection algorithm," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, ICASSP '95*, vol. 5. IEEE, 9–12 May 1995, pp. 3023–3026.

[6] J. Cioffi and T. Kailath, "Fast, fixed-order, least-squares algorithms for adaptive filtering," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, ICASSP '83*, vol. 8. IEEE, Apr. 1983, pp. 679–682.

[7] D. T. M. Slock and T. Kailath, "Numerically stable fast transversal filters for recursive least squares adaptive filtering," *IEEE Transactions on Signal Processing*, vol. 39, no. 1, pp. 92–114, Jan. 1991.

[8] Q. G. Liu, B. Champagne, and K. C. .Ho, "On the use of a modified fast affine projection algorithm in subbands for acoustic echo cancelation," in *Proceedings of the Seventh Digital Signal Processing Workshop*. IEEE, 1–4 Sept. 1996, pp. 354–357.

[9] Y. Kaneda, M. Tanaka, and J. Kojima, "An adaptive algorithm with fast convergence for multi-point sound control," in *Proceedings of the Active '95*, Newport Beach, California, 6–8 July 1995, pp. 993–1004.

[10] H. Ding, "A stable fast affine projection adaptation algorithm suitable for low-cost processors," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, ICASSP '00*, vol. 1. IEEE, 5–9 June 2000, pp. 360–363.

[11] D. G. Luenberger, *Linear and Nonlinear Programming*, 2nd ed. Reading, Massachusetts: Addison-Wesley, 1984.

[12] F. Albu, A. Fagan, J. Kadlec, and N. Coleman, "The Gauss-Seidel fast affine projection algorithm," in *Proceedings of the Workshop on Signal Processing Systems, SIPS '02*. IEEE, 16–18 Oct. 2002, pp. 109–114.

[13] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. Baltimore, Maryland: The Johns Hopkins University Press, 1996.

[14] J. N. Coleman, E. I. Chester, C. I. Softley, and J. Kadlec, "Arithmetic on the European Logarithmic Microprocessor," *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 702–715, July 2000.

[15] R. Matousek, M. Tichy, Z. Pohl, J. Kadlec, C. Softley, and N. Coleman, "Logarithmic number system and floating-point arithmetics on FPGA," in *12th International Conference on Field Programmable Logic and Applications*, vol. 2438. Springer-Verlag, Sept. 2002, pp. 627–636.

[16] K. Underwood, "FPGAs vs.CPUs: Trends in peak floating-point performance," in *ACM SIGDA 12th Iternational Symposium on Field Programming Gate Arrays*. Monterey: ACM, Feb. 2004, pp. 171–179.