A HARDWARE-EFFICIENT IMPLEMENTATION OF THE FAST AFFINE PROJECTION ALGORITHM

Haw-Jing Lo and David V. Anderson

Center for Signal and Image Processing
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, GA 30332

ABSTRACT

This paper presents a high-throughput, low-latency, hardware-efficient fixed-point implementation of the fast affine projection (FAP) algorithm. The proposed architecture utilizes reusable distributed arithmetic (RDA) in combination with optimizations in the update process to enable the coefficients to be updated in a fixed number of cycles independent of filter length. Fixed-point simulations show that the effect of replacing some of the multiplications with arithmetic shifts is minor, with RDA-FAP maintaining a faster convergence rate than NLMS. The proposed design is also compared against a multiplier-based design in terms of number of computations and number of clock cycles needed for a single FAP update cycle.

Index Terms— distributed arithmetic, adaptive filters, fast affine projection

1. INTRODUCTION

Adaptive filtering is widely utilized in applications such as echo cancellation, noise cancellation, channel equalization, and system identification. Typical adaptive filters use the LMS (least mean squares) and NLMS (normalized LMS) algorithms. These algorithms are commonly used due to their simplistic nature of implementation. However, for colored signals such as speech, the convergence speed of LMS and NLMS is slow. The adaptive affine projection algorithm was proposed as a generalization to the NLMS algorithm [1], and offers a faster convergence rate for colored signals but at the cost of higher computational complexity. The fast version (FAP) offers recursive least squares type convergence with slightly more computation than NLMS [2], but suffers from numerical instabilities due to the way the matrix inversion is computed [3].

Recently several methods for computing the fast affine projection have been introduced suitable for fixed-point implementations, differing primarily in how the matrix inversion problem is solved. For example, the conjugate gradient FAP (CGFAP) [4] and Gauss-Seidel FAP (GSFAP) [5] approach the matrix inversion problem by reducing the problem into solving a linear system of equations using the conjugate gradient method and Gauss-Seidel method respectively. Oh, *et al.* assume the matrix is Toeplitz and solve for the inverse using the Levinson-Durbin recursion [6]. The majority of the proposed FAP algorithms have been implemented using digital signal processors (DSPs) because of the divisions involved in the coefficient update, multiple computations, and the non-trivial datapath of the algorithm. The sequential execution nature coupled with the large number of computations restrict the maximum throughput attainable by DSPs.

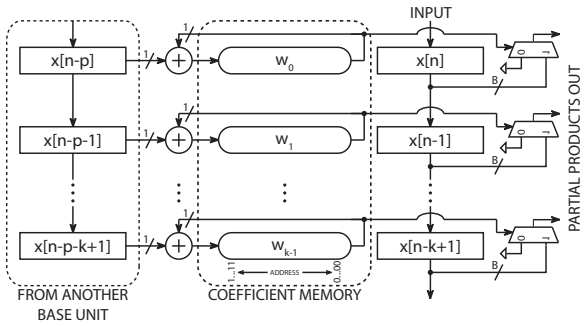


Fig. 1. Block diagram of the RDA-FAP base unit. B represents the number of bits.

This paper presents a hardware-efficient, high-throughput, low-latency, fixed-point implementation of the FAP algorithm based on the Gauss-Seidel approach. The proposed architecture achieves high throughput and low complexity in three ways: 1) by using reusable distributed arithmetic [7], 2) using single iteration LUT-based division, and 3) parallelization of the coefficient update computations. This proposed architecture is termed RDA-FAP. The computations per output and performance of the RDA-FAP architecture is compared against that of a multiplier-based FAP (M-FAP) architecture. The computations per output include the number of additions, multiplications, and divisions, while the performance involves the number of clock cycles needed to complete one adaptive cycle. The RDA-FAP architecture is also synthesized using a Xilinx Virtex-4 LX100.

The rest of this paper is organized as follows. The fast affine projection algorithm is presented in Section 2. The architecture and operation of the proposed RDA-FAP adaptive filter is detailed in Section 3, along with simulation results validating the optimizations performed. Section 4 discusses the results of computations per output and performance between the RDA-FAP and M-FAP architectures, and concluding remarks are presented in Section 5.

2. FAST AFFINE PROJECTION

The fast affine projection algorithm is described as follows. For $n < 0$,

$$\mathbf{w}(n) = \mathbf{0}, \boldsymbol{\eta}(n) = \mathbf{0}, \mathbf{R}(n) = \delta\mathbf{I}, \boldsymbol{\alpha}(n) = \mathbf{0}, \mathbf{r}(n) = [\delta, \mathbf{0}]^T \quad (1)$$

and for $n \geq 0$,

$$\mathbf{r}(n) = \mathbf{r}(n-1) + x(n)\boldsymbol{\alpha}(n) - x(n-L)\boldsymbol{\alpha}(n-L) \quad (2)$$

Table 1. Number of Multiply-Accumulate Operations and Clock Cycles

	# of MAC operations		# of clock cycles	
Operation	**RDA-FAP**	**M-FAP**	**RDA-FAP**	**M-FAP**
Filtering	L	L	$B+\log_2(L)$	$L/N+\log_2(N)$
Compute $\bar{\mathbf{r}}^T(n)\underline{\boldsymbol{\eta}}(n-1)$	$p-1$	$p-1$	$p-1$	1 or $p-1$
Solving $\mathbf{Rp}=\mathbf{b}$	p^2-p	p^2-p+20	p^2-p	p^2-p+20
Compute $e(n)$	2	2	2	2
Compute $\boldsymbol{\epsilon}(n)$ and $\boldsymbol{\eta}(n)$	p	p	1	1
Update $\mathbf{w}(n)$	L/B	L	B	L/N
Total*	$L+\frac{L}{B}+p^2+p+2$	$2L+p^2+p+21$	$2B+\log_2(L)+3$	$\frac{2L}{N}+\log_2(N)+3$

* The total number of clock cycles assumes that filtering takes the longest amount of time.

$$\mathbf{R}(n)=\begin{bmatrix} r_0(n) & r_1(n) & \dots & r_{p-1}(n) \\ r_1(n) & r_0(n-1) & \dots & r_{p-2}(n-1) \\ \vdots & & \ddots & \\ r_{p-1}(n) & r_{p-2}(n-1) & \dots & r_0(n-p+1) \end{bmatrix} \tag{3}$$

$$e(n)=d(n)-\mathbf{x}^T(n)\mathbf{w}(n-1)-\mu\bar{\mathbf{r}}^T(n)\underline{\boldsymbol{\eta}}(n-1) \tag{4}$$

$$\mathbf{e}(n)=\begin{bmatrix} e(n) \\ (1-\mu)\underline{\mathbf{e}}(n-1) \end{bmatrix} \tag{5}$$

$$\boldsymbol{\epsilon}(n)=\mathbf{P}(n)\mathbf{e}(n) \tag{6}$$

$$\boldsymbol{\eta}(n)=\begin{bmatrix} 0 \\ \underline{\boldsymbol{\eta}}(n-1) \end{bmatrix}+\boldsymbol{\epsilon}(n) \tag{7}$$

$$\mathbf{w}(n)=\mathbf{w}(n-1)+\mu\eta_{p-1}(n)\mathbf{x}(n-p+1) \tag{8}$$

where $\mathbf{w}(n)$ represents the coefficients of a L-tap filter, p is the projection order, $\mathbf{x}(n)=[x(n),x(n-1),\dots,x(n-L+1)]^T$, $\mathbf{X}(n)=[\mathbf{x}(n),\mathbf{x}(n-1),\dots,\mathbf{x}(n-p+1)]^T$, $\mathbf{P}(n)=\mathbf{R}^{-1}(n)$ is the inverse of the $p\times p$ autocorrelation matrix $\mathbf{R}(n)$, $\boldsymbol{\eta}(n)$ is a $p\times 1$ vector, η_{p-1} is the bottommost element of $\boldsymbol{\eta}(n)$, $\underline{\mathbf{e}}(n)$ and $\underline{\boldsymbol{\eta}}(n)$ are the $p-1$ upper elements of $\mathbf{e}(n)$ and $\boldsymbol{\eta}(n)$ respectively, $\bar{\mathbf{r}}(n)$ is the lower $p-1$ elements of $\mathbf{r}(n)$, and $\boldsymbol{\alpha}(n)=[x(n),x(n-1),\dots,x(n-p+1)]^T$.

2.1. Gauss-Seidel Iteration

The GSFAP algorithm uses the Gauss-Seidel (GS) iterative method for solving linear systems. The GS method solving the system $\mathbf{Ax}=\mathbf{b}$ is described below,

$$x_i^{(k+1)}=\frac{1}{a_{ii}}\left(b_i-\sum_{j<i}a_{ij}x_j^{(k+1)}-\sum_{j>i}a_{ij}x_j^{(k)}\right) \tag{9}$$

where k is the iteration count, and $i=1,2,\dots,N$, where $\mathbf{x}$ and $\mathbf{b}$ are $N\times 1$ vectors, and $\mathbf{A}$ is a $N\times N$ matrix. An important property of the GS iteration is that the elements $x_k^{(k+1)}$ uses the elements of $x_k^{(k+1)}$ that have already been computed in the current iteration. This means that no additional storage is needed, and all the computation is done in place. The GS iterative method is run until $\mathbf{x}$ converges to the correct solution. By setting $\mu=1$ in Eqn. 5, the vector $\mathbf{e}(n)$ is reduced to the scalar $e(n)$, reducing Eqn. 6 to,

$$\boldsymbol{\epsilon}(n)=\mathbf{p}(n)e(n) \tag{10}$$

where $\mathbf{p}(n)$ is the leftmost column of $\mathbf{P}(n)$, and

$$\mathbf{R}(n)\mathbf{p}(n)=[1,\mathbf{0}]^T \tag{11}$$

which is then solved using the Gauss-Seidel iterative method. It has been shown in [5] that a single iteration is enough for the FAP adaptive algorithm to converge. Given the initial conditions (Eqn. 1), the assumption that $\mathbf{R}(n)$ is Toeplitz, and using the Gauss-Seidel method to solve Eqn. 11, for a $p=3$ case, $\mathbf{p}(n)$ is evaluated as

$$p_0(n)=\frac{1}{r_0(n)} \tag{12}$$

$$p_1(n)=\frac{1}{r_0(n)}[-r_1(n)p_0(n)] \tag{13}$$

$$p_2(n)=\frac{1}{r_0(n)}[-r_2(n)p_0(n)-r_1(n)p_1(n)] \tag{14}$$

with a minimal number of storage elements needed since only the current values of $\mathbf{r}(n)$ are stored instead of both the current and delayed values. The previous collection of equations is what is used in the implementation of the RDA-FAP adaptive filter.

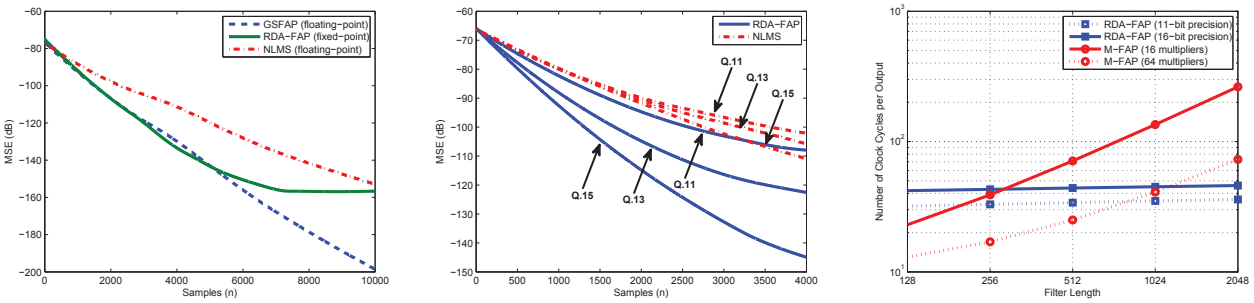
3. FAST AFFINE PROJECTION USING RDA

FIR filters based on reusable distributed arithmetic have been shown to be a hardware-efficient architecture for high order digital filters, requiring fewer resources than the multiplier-based counterpart, and at the same time freeing up those hardware resources to be used for other functions. This hardware efficiency is carried over to the FAP algorithm, reducing the latency and increasing the throughput associated with filtering (Eqn. 4). Further throughput increase can be obtained by computing as much of the GSFAP algorithm in parallel as possible.

3.1. Reusable Distributed Arithmetic

Distributed arithmetic (DA) is a bit serial method of computing the inner product of two vectors with a fixed number of cycles [7]. Consider the inner product of two L dimensional vectors $\vec{a}$ and $\vec{x}$, where $\vec{a}$ is a constant vector, $\vec{x}$ is the input sample vector, and y is the result.

$$y=\sum_{k=0}^{L-1}a_kx_k \tag{15}$$



(a) RDA-FAP, NLMS, and GSFAP. (b) The effects of reducing bit precision. (c) Clock cycles comparison.

Fig. 2. (a) and (b) Convergence curves comparing the performance of RDA-FAP in different scenarios. For both plots $L = 64$, $p = 4$, $\mu = 1$, and $\varphi = 0.95$. The results are obtained by averaging 20 runs. (c) Total number of clock cycles needed per output for different cases of RDA-FAP and M-FAP.

Using B-bit 2's complement binary representation scaled such that $|x_k| \leq 1$ produces

$$x_k = -b_{k(B-1)} + \sum_{n=0}^{B-2} b_{kn} 2^{-(B-1)+n} \tag{16}$$

where b_{kn} are the bits (0 or 1) of x_k, $b_{k(B-1)}$ is the most significant bit, and b_{k0} is the least significant bit. Substituting Eqn. 16 into Eqn. 15 yields

$$y = -\sum_{k=0}^{L-1} a_k b_{k(B-1)} + \sum_{n=0}^{B-2} \left[\sum_{k=0}^{L-1} a_k b_{kn} \right] 2^{-(B-1)+n} \tag{17}$$

Equation 17 represents the distributed arithmetic computation. The values of b_{kn} are either 0 or 1, resulting in the bracketed term in Eqn. 17 having only 2^L possible values. Since $\vec{a}$ is a constant vector, the bracketed term can be precomputed and stored in memory using either a lookup table (LUT) or ROM. The lookup table is then addressed using the individual bits of the input samples, x_k with the final result y computed after B cycles, regardless of the lengths of the vectors $\vec{a}$ and $\vec{x}$. Typical DA methods require a memory with memory size exponentially dependent on the filter length whereas reusable DA has a linear dependence of memory size on length. The linear dependence requires extra additions, introducing extra $\log_2(L)$ clock cycles. A large portion of the RDA architecture can be reused, saving hardware resources while maintaining low-latency and high-throughput for large filter sizes. The use of RDA also enables the update of all the coefficients (Eqn. 8) to be performed in a fixed number of cycles that is independent of filter length. The RDA-FAP base unit block diagram is shown in Fig. 1.

The operation of the RDA-FAP base unit is as follows. During the filtering phase, the coefficient bits are read out from the coefficient memory, controlling the partial product generation. Once filtering is complete and $\eta_{p-1}(n)$ is ready, the coefficients are updated simultaneously due to the way they are stored in memory (Fig. 1).

3.2. Coefficient Update Mechanism

The computations for coefficient update can be performed in parallel with filtering up to a certain point, where the value of $e(n)$ is needed. Considering the ability of RDA to complete the filtering operation in a relatively small and fixed number of cycles, the calculations leading up to $e(n)$ (Eqns. 2, 11) should be completed by the time $e(n)$ needs to be computed. RDA-FAP implements the coefficient update equations using a single multiply-accumulate (MAC) unit in conjunction with p adders. The order of operations is described in the following paragraph.

The MAC unit is first used to compute Eqn. 2. Once $\mathbf{r}(n)$ is available, the terms in the square brackets of Eqns. 12-14 are sequentially computed. The reciprocal $1/r_0(n)$ is computed using LUT-based division during the execution of the previous MAC operations. When the square bracket terms are computed, multiply through with the reciprocal to obtain $\mathbf{p}(n)$. At this time $e(n)$ should be ready, however instead of multiplying the vector $\mathbf{p}(n)$ by the scalar $e(n)$, the multiplication operation is converted into an arithmetic shift right operation on $\mathbf{p}(n)$. The error value $e(n)$ is quantized into 8 levels spanning the entire range of $e(n)$, with each level corresponding to a different shift amount. The vector $\boldsymbol{\eta}(n)$ is simultaneously updated with the shifted version of $\mathbf{p}(n)$. In this way no storage is needed for $\boldsymbol{\epsilon}(n)$. With $\boldsymbol{\eta}(n)$ updated, the final coefficient update step can be performed. Similar to the update of $\boldsymbol{\eta}(n)$, instead of computing $\mu\eta_{p-1}(n)\mathbf{x}(n-p+1)$, the correction term $\mu\eta_{p-1}(n)$ is quantized into an arithmetic shift operation on $\mathbf{x}(n-p+1)$. The coefficients can be updated simultaneously now that no intermediate multiplications are required. The update is performed in a bit serial manner, where the matching individual bits of the shifted $x_k(n-p+1)$ and $w_k(n-1)$ are added/subtracted together to update the corresponding bit of $w_k(n)$.

3.3. LUT-based Division

The computation of $\mathbf{p}(n)$ using the Gauss-Seidel method involves a division for every element of $\mathbf{p}(n)$. With $\mathbf{R}(n)$ assumed to be Toeplitz, the p divisions involve dividing by the same number, $r_0(n)$. The reciprocal of $r_0(n)$ is computed, stored, and multiplied to the elements of $\mathbf{p}(n)$. The reciprocal is computed by first normalizing $r_0(n)$ to the range of $1 < \hat{r}_0(n) < 2$. Then K bits from the fractional part of $\hat{r}_0(n)$ are used as the address into a lookup table that holds 2^K corresponding precomputed reciprocal values. The normalization operation is implemented by finding, from MSB to LSB, the first 1 in the bits of $r_0(n)$. The location of the 1 is noted, and K bits to the right of that 1 are used as the address into a lookup table. The quotient is read out from the table, and shifted to the left by the number of shifts needed to move the first 1 to the MSB position. In the RDA-FAP implementation, the normalization and reciprocal lookup operation requires one clock cycle, and the lookup

table contains 16 entries.

3.4. Simulation Results

The fixed-point RDA-FAP was simulated in a system identification configuration against floating-point versions of FAP and NLMS. The input to the system is an AR(1) process represented by $x(n) = w(n) + \varphi x(n-1)$ where $w(n)$ is white gaussian noise and $0 < |\varphi| < 1$. The results show that the quantization of the $e(n)\mathbf{p}(n)$ and $\eta_{p-1}\mathbf{x}(n-p+1)$ operations into arithmetic shifts, along with the approximate reciprocation via LUT-based division, are valid optimizations (Fig. 2(a)). When combined with the use of RDA, the optimizations enable the rapid computation of a single FAP adaptive cycle. The effects of bit precision width are also shown in Fig. 2(b). For decreasing bit precision, the convergence rate of RDA-FAP declines faster than for fixed-point NLMS. However, for the same precision, RDA-FAP still converges faster than NLMS for correlated inputs.

4. RESULTS AND COMPARISONS

The proposed RDA-FAP adaptive filter is compared against a multiplier-based (M-FAP) approach using the same GSFAP algorithm. The two designs are evaluated in terms of computations per output (number of operations), and performance (number of clock cycles).

4.1. Computations per Output

The number of additions and multiplications are presented in Tbl. 1. For simplicity a B-bit addition is counted as one B-bit MAC, and a division operation is counted as 20 MAC operations. The results from Tbl. 1 show that for filter lengths equal to or larger than the fixed-point bit precision ($L \geq B$), the RDA-FAP approach requires fewer operations. Depending on the application, the savings in number of operations could be significant, even when using 16 or 32 bit precision. For example, adaptive filters used in acoustic echo cancellation typically have filter lengths of $L = 1024$ or longer.

4.2. Performance

The computations per output of RDA-FAP and M-FAP are very similar to each other primarily because of the way the coefficient update is computed. While there are computation gains in using RDA-FAP, when both designs are implemented and analyzed in terms of performance, the gains becomes even greater. Table 1 lists the number of clock cycles needed to compute individual parts of the FAP algorithm. The total number of clock cycles assumes that the filtering operation requires more cycles to complete than the parallel coefficient update. N represents the number of multipliers used for filtering, it does not include the multiplier(s) used for coefficient update. With M-FAP, if p multipliers are available during filtering, then $\mathbf{p}(n)$ and $\bar{\mathbf{r}}^T(n)\underline{\boldsymbol{\eta}}(n-1)$ are each completed in a single cycle. Otherwise $\mathbf{p}(n)$ and $\bar{\mathbf{r}}^T(n)\underline{\boldsymbol{\eta}}(n-1)$ each require $p-1$ cycles. Table 1 also assumes that both RDA-FAP and M-FAP have enough hardware resources to compute $\boldsymbol{\epsilon}(n)$ and $\boldsymbol{\eta}(n)$ in a single cycle. If the amount of time it takes to complete a B-bit multiply-accumulate is assumed to be the same as a B-bit addition, from Tbl. 1, for the case where $L = 1024$, $B = 16$, $N = 32 + 1$, and $p = 4$, the total number of clock cycles per sample for RDA-FAP is $32 + 10 + 3 = 45$, and for M-FAP the total number is $64 + 5 + 3 = 72$. Figure 2(c) shows the effect of B, N, and L on the total number of clock cycles.

Table 2. RDA-FAP Synthesis Results

Filter Length	128	256	512
Number of Slices	9063	16698	32439
Equivalent Gate Count	181433	331864	636783

4.3. FPGA Synthesis

The RDA-FAP design was modeled using VHDL and synthesized onto a Xilinx Virtex-4 LX100 FPGA. This FPGA has 96 XtremeDSP slices which can be used to implement high speed multipliers. Different filter lengths were synthesized, ranging from 128 to 512 taps, with 16-bit precision for the inputs and outputs. The synthesis results are shown in Tbl. 2.

5. CONCLUSION

A high-throughput, low-latency, hardware-efficient architecture for the fast affine projection algorithm has been presented. The proposed RDA-FAP design uses reusable distributed arithmetic in conjunction with algorithm optimizations to significantly reduce the number of clock cycles required per adaptive cycle. For the case where $L = 1024$, $N = 16$, and $B = 16$, one FAP adaptive cycle would require 135 cycles, with RDA-FAP that number has been reduced to 45, a factor of 3 in savings.

6. REFERENCES

[1] K. Ozeki and T. Umeda, "An adaptive filtering algorithm using an orthogonal projection to an affine subspace and its properties," in *Electron. Comm. Japan*, 1984, vol. 67-A, pp. 19–27.

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

[3] S.C. Douglas, "Efficient approximate implementations of the fast affine projection algorithm using orthogonal transforms," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP-96*, 7–10 May 1996, vol. 3, pp. 1656–1659.

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

[5] F. Albu, J. Kadlec, N. Coleman, and A. Fagan, "The gauss-seidel fast affine projection algorithm," in *Proc. IEEE Workshop on Signal Processing Systems (SIPS '02)*, 16–18 Oct. 2002, pp. 109–114.

[6] S. Oh, D. Linebarger, B. Priest, and B. Raghothaman, "A fast affine projection algorithm for an acoustic echo canceller using a fixed-point DSP processor," in *Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing ICASSP-97*, 21–24 April 1997, vol. 5, pp. 4121–4124.

[7] Haw-Jing Lo, Heejong Yoo, and David V. Anderson, "A reusable distributed arithmetic architecture for FIR filtering," in *Proc. 51st Midwest Symposium on Circuits and Systems MWSCAS 2008*, 2008, pp. 233–236.