Implementation of (Normalised) RLS Lattice on Virtex

Felix Albu[1], Jiri Kadlec[2], Chris Softley[3], Rudolf Matousek[2], Antonin Hermanek[2]
Nick Coleman[3], Anthony Fagan[1]

[1]University College Dublin, Ireland
felix@ee.ucd.ie
[2]UTIA Prague, Czech Republic
Kadlec@utia.cas.cz
[3]University of Newcastle upon Tyne, UK
C.I.Softley@ncl.ac.uk

Abstract. We present an implementation of a complete RLS Lattice and Normalised RLS Lattice cores for Virtex. The cores accept 24-bit fixed point inputs and produce 24-bit fixed point prediction error. Internally, the computations are based on 32bit logarithmic arithmetic. On Virtex XCV2000E-6, it takes 22% and 27% of slices respectively and performs at 45 MHz. The cores outperform (4-5 times) the standard DSP solution based on 32 bit floating point TMS320C3x/4x 50MHz processors.

Introduction

The lattice algorithms solve the least-squares problem in a recursive form. They require less arithmetic operations than RLS (order N) [2]. They offer a number of advantages over conventional LMS transversal algorithms such us faster rate of convergence, modular structure and insensitivity to variations in the eigenvalue spread of the input correlation matrix. Another feature of the lattice-based algorithms is their good performance when implemented in finite-precision arithmetic [2]. However, the high computational load of division or square-root operations is one of the reasons why these algorithms are usually not used in real-time applications. They need floating point-like precision, and this has been a severe restriction for FPGA use. FPGA offers a viable alternative to programmable DSP processors or ASIC for some applications (see for example [7,8]). As an alternative to floating-point, the logarithmic number system offers the potential to perform real multiplication, division and square-root at fixed-point speed and, in the case of multiply and divide, with no rounding error at all. These advantages are, however, offset by the problem of performing logarithmic addition and subtraction. Hitherto this has been slower or less accurate than floating-point, or has required very cumbersome hardware. Following the discovery of new arithmetic techniques at Newcastle, however, it is possible to perform logarithmic addition and subtraction with speed and accuracy equivalent to that of floating-point [1,3]. The patented solution developed by the HSLA project team under Dr. Coleman, yields a drastic reduction in the size of the look-up tables required compared to those needed for conventional linear interpolation of both functions. This is achieved by the parallel evaluation of a linear approximant and an error correction term. Furthermore, it has been shown that a modified form of LNS operation is possible, which delivers considerably better precision in applications involving underflow. This “Extended Precision LNS” is described in [4]. Coleman's approach leads to a suitable solution for the FPGA implementation. It avoids the need for a barrel shifter, implementation of which is area-costly and ineffective in an FPGA. The LNS ALU provides one of the first hardware solutions to this problem. For a description of the ALU see [5]. We present results of the implementation based on this 32-bit logarithmic ALU designed in Handel C for the Celoxica DK1 toolset [6]. The core takes just 8% of the XILINX Virtex XCV2000E-6 device. It operates at 53MHz and implements all the basic operations of logarithmic arithmetic (ADD, SUB, MUL, DIV and SQRT), with precision equal to or better than:

– the standard IEEE 32-bit floating point used in new DSPs
– the TI 32-bit floating point standard used in the TMS320C30/C40 devices.

The block diagram for our implementation is presented in Fig.1. In the next two sections we present the Lattice RLS algorithm based on *A Posteriori* errors and the Normalised *A Posteriori* Error Lattice RLS algorithm.

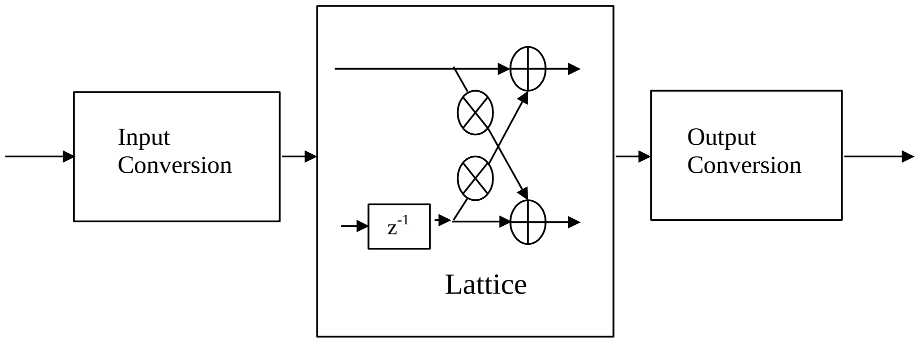

Fig. 1. Block diagram of LNS Lattice implementation

Lattice RLS Algorithm

Table 1 presents the Lattice RLS (LRLS) Algorithm based on *A Posteriori* errors [2]:

Initialisation
Do for $i = 0,1,\ldots,N$

$$\delta(-1,i) = \delta_D(-1,i) = 0, \xi^d_{b_{\min}}(-1,i) = \xi^d_{f_{\min}}(-1,i) = \varepsilon$$

$$\gamma(-1,i) = 1, e_b(-1,i) = 0$$

Do for $k \geq 0$

$\gamma(k,0) = 1, e_b(k,0) = e_f(k,0) = x(k), e(k,0) = d(k)$

$\xi^d_{b_{\min}}(k,0) = \xi^d_{f_{\min}}(k,0) = x^2(k) + \lambda \xi^d_{f_{\min}}(k-1,0)$

For each $k \geq 0$, do for $i = 0,1,\ldots, N$

$$\delta(k,i) = \lambda \delta(k-1,i) + \frac{e_b(k-1,i) e_f(k,i)}{\gamma(k-1,i)}$$

$$\gamma(k,i+1) = \gamma(k,i) - \frac{e_b^2(k,i)}{\xi^d_{b_{\min}}(k,i)}$$

$$k_b(k,i) = \frac{\delta(k,i)}{\xi^d_{f_{\min}}(k,i)}$$

$$k_f(k,i) = \frac{\delta(k,i)}{\xi^d_{b_{\min}}(k-1,i)}$$

$$e_b(k,i+1) = e_b(k-1,i) - k_b(k,i) e_f(k,i)$$

$$e_f(k,i+1) = e_f(k,i) - k_f(k,i) e_b(k-1,i)$$

$$\xi^d_{b_{\min}}(k,i+1) = \xi^d_{b_{\min}}(k-1,i) - \frac{\delta^2(k,i)}{\xi^d_{f_{\min}}(k,i)}$$

$$\xi^d_{f_{\min}}(k,i+1) = \xi^d_{f_{\min}}(k,i) - \frac{\delta^2(k,i)}{\xi^d_{b_{\min}}(k-1,i)}$$

Feed-forward Filtering

$$\delta_D(k,i) = \lambda \delta_D(k-1,i) + \frac{e(k,i) e_b(k,i)}{\gamma(k,i)}$$

$$v_i(k) = \frac{\delta_D(k,i)}{\xi^d_{b_{\min}}(k,i)}$$

$$e(k,i+1) = e(k,i) - v_i(k) e_b(k,i)$$

Table 1. The Lattice RLS Algorithm based on *A Posteriori* Errors.

$e_f(k,i)$ represents the instantaneous *a posteriori* forward prediction error, $e_b(k,i)$ represent the instantaneous *a posteriori* backward prediction error, $\xi_{f_{\min}}(k,i)$ and $\xi_{b_{\min}}(k,i)$ are the minimum in least-squares sense of the forward and backward prediction errors respectively. The coefficients $k_f(k,i)$ and $k_b(k,i)$ are called the forward and backward reflection coefficients. $\gamma(k,i)$ is a conversion factor between *a priori* and *a posteriori* errors and $v_i(k)$ are the feedforward multiplier coefficients.

The Normalised LRLS Algorithm

Table 2 presents the Normalised *A Posteriori* Error LRLS Algorithm [2] :

Initialisation

Do for $i = 0,1,\ldots,N$

$$\bar{\delta}(-1,i)=0, \bar{\delta}_D(-1,i)=0, \bar{e}_b(-1,i)=0$$

$$\sigma_x^2(-1)=\lambda\sigma_d^2(-1)=\varepsilon$$

Do for $k \geq 0$

$$\sigma_x^2(k)=\lambda\sigma_x^2(k-1)+x^2(k)(\text{Inputsignal energy})$$

$$\sigma_d^2(k)=\lambda\sigma_d^2(k-1)+d^2(k)(\text{Reference signal energy})$$

$$\bar{e}_b(k,0)=\bar{e}_f(k,0)=x(k)/\sigma_x(k)$$

$$\bar{e}(k,0)=d(k)/\sigma_d(k)$$

For each $k \geq 0$ do for $i = 0,1,\ldots,N$

$$\bar{\delta}(k,i)=\delta(k-1,i)\sqrt{\left(1-\bar{e}_b^2(k-1,i)\right)\left(1-\bar{e}_f^2(k,i)\right)}+\bar{e}_b(k-1,i)\bar{e}_f(k,i)$$

$$\bar{e}_b(k,i+1)=\frac{\bar{e}_b(k-1,i)-\bar{\delta}(k,i)\bar{e}_f(k,i)}{\sqrt{\left(1-\bar{\delta}^2(k,i)\right)\left(1-\bar{e}_f^2(k,i)\right)}}$$

$$\bar{e}_f(k,i+1)=\frac{\bar{e}_f(k,i)-\bar{\delta}(k,i)\bar{e}_b(k-1,i)}{\sqrt{\left(1-\bar{\delta}^2(k,i)\right)\left(1-\bar{e}_b^2(k-1,i)\right)}}$$

Feedforward filter

$$\bar{\delta}_D(k,i)=\bar{\delta}_D(k-1,i)\sqrt{\left(1-\bar{e}_b^2(k,i)\right)\left(1-\bar{e}^2(k,i)\right)}+\bar{e}(k,i)\bar{e}_b(k,i)$$

$$\bar{e}(k,i+1)=\frac{1}{\sqrt{\left(1-\bar{e}_b^2(k,i)\right)\left(1-\bar{\delta}_D^2(k,i)\right)}}\left[\bar{e}(k,i)-\bar{\delta}_D(k,i)\bar{e}_b(k,i)\right]$$

Table 2. The Normalised *A Posteriori* Error LRLS Algorithm

The reconstructed $e(k,N+1)$ is the standard prediction error and it must be up to the numerical rounding identical to the prediction error produced by the normal RLS lattice. Only this reconstructed prediction error can be reasonably compared with the prediction error produced by other RLS algorithms.

$$e(k, N+1) = \left(\bar{e}(k, N+1) \prod_{j=0}^{N} \sqrt{\left(1 - \bar{e}_b^2(k, j)\right)} \prod_{j=0}^{N} \sqrt{\left(1 - \bar{\delta}_D^2(k, j)\right)} \right) \sigma_d(k) \quad (1)$$

The "chain" of 2 x N supplementary multiplications can be performed by Extended-LNS. Only the final single re-scaling needs the standard LNS multiplication and returns us to the "real" domain.

Results

The Lattice RLS Algorithm based on *A Posteriori* Errors and the Normalised *A Posteriori* Error LRLS Algorithm were used to identify a system with impulse response **h**=[0.1 0.3 0.0 −0.2 −0.4 −0.7 −0.4 −0.2]. The input signal was generated as a first-order AR process with the eigenvalue spread of the correlation matrix of 20 [2]. The forgetting factor was $\lambda = 0.96$ and the parameter $\varepsilon = 0.01$. The standard deviation of the input was 1 and the standard deviation of measurement noise was 0.01. Results for the double implementations of LRLS and NLRLS are identical. However, the finite implementations have slightly different performances (Fig.2).

In order to compare the numerical properties of the different implementations we used a procedure similar to that described in [4]. An input noise signal was generated. Starting from time 500 the noise signal is changed to a sine, thereby creating non-persistent excitation and hence poorly conditioned operation. An accurate standard for comparison of the outputs was obtained by presenting this input data to the IEEE double precision floating-point versions of each filter.

Figs. 3-4 present the absolute sum of errors for the output results of the identification of a 30th order FIR regression model by this filter, with exp. weighting factor 0.9. The two algorithms have different numerical properties and each has errors in a different range [2]. Some time after the start of non-persistent excitation at time 500, however, each implementation could become unstable, with the accumulation of successively larger errors. We notice that FLOAT-LRLS and the LNS-LRLS errors start to grow very much after 100 samples of the non-persistent excitation. In the same time ELNS-NLRLS and LNS-NLRLS start to drift at about 200 samples of the non-persistent excitation. These results are comparable to the RMGS and NL RMGS RLS reported in [4]. Therefore LNS enables at no substantial cost to go to the normalised version of the algorithm and gain the robustness in the comparison with the un-normalised version. The normalised lattice is particularly suitable for extended LNS implementation because of its normalised internal variables.

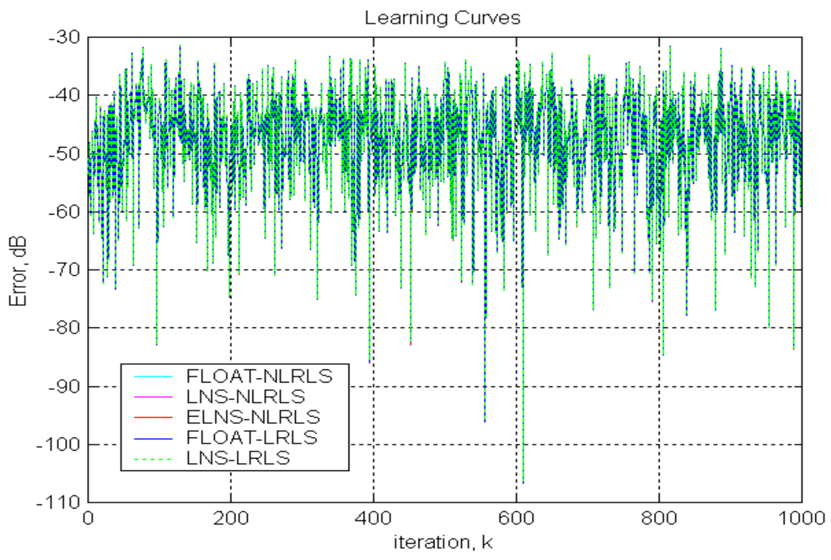

Fig. 2. The learning curves for FLOAT 32bit and LNS implementations of Lattice RLS and Normalised Lattice RLS Algorithms

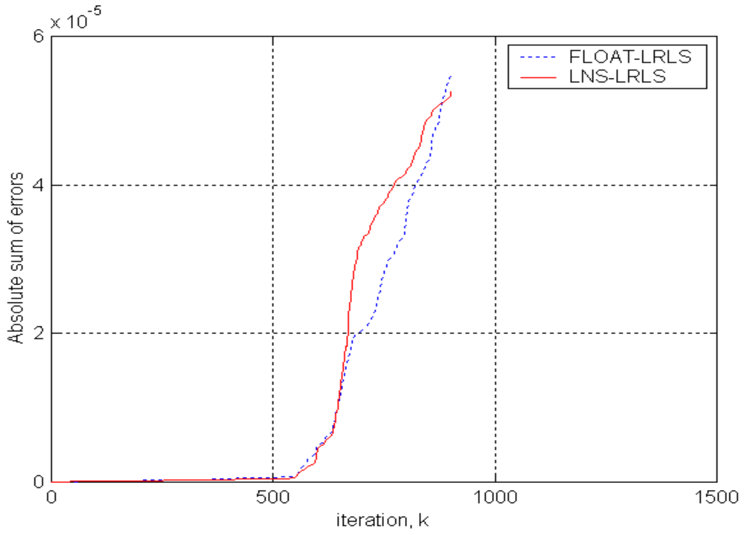

Fig. 3. The absolute sum of errors for FLOAT and LNS implementation of the Lattice RLS Algorithm.

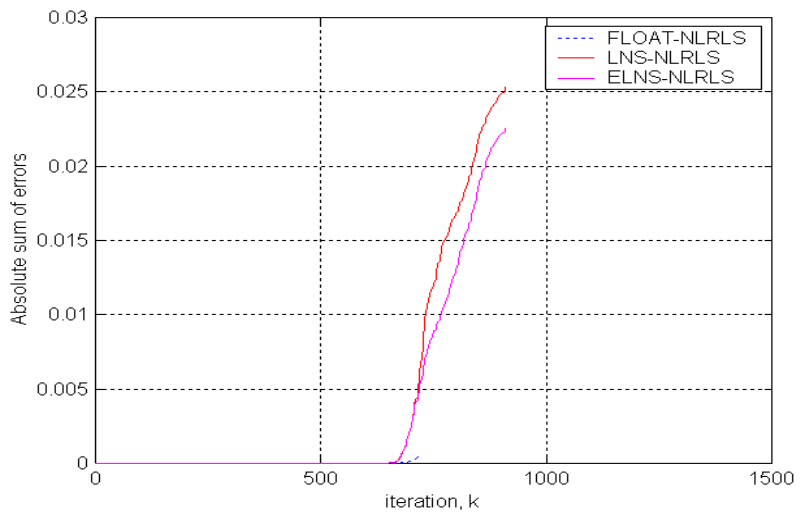

Fig. 4. The absolute sum of errors for ELNS, FLOAT and LNS implementation of the Normalised Lattice RLS Algorithm

Comparison of LNS FPGA implementation with TMS320C30/C40

The instruction counts for 8-th order filters are presented in Tables 4-6. In counting FLP operations, additions and subtractions were recorded from LRLS or NLRLS algorithm equations. The number of multiplications and divisions is about twice the number of additions and subtraction operations. Therefore these algorithms are suitable for LNS implementation. Clock cycles for each type of operation are presented in Table 3. int2* and *2int indicate the functions for conversion to/from int domain to/from log or floating point domain. The conversion to/from log domain is based on evaluation of the rational polynomial approximation of log and antilog in the range (-1,1) conversion. The log ALU is used for the conversion. For details see [5].

	add	sub	mul	div	Sqrt	Int2*	*2int
C3X/4X	4	4	4	62	92	2	2
Log	10	10	1	1	1	100	60

Table 3. Comparison of LNS and FLOAT execution time (clock cycles)

The LNS multiplication, division and square-root operations are implemented by fixed-point addition, subtraction, right shift and are extremely efficient. LNS addition and subtraction, implemented as described in [1], requires a number of table-lookups. In the current arrangement these tables are located in four external banks of SRAM, and hence require several cycles to access [5]. Future FPGA implementations will use the on-chip RAM, which will yield a substantial decrease in the latency of these operations.

	add	sub	mul	div	int2*	*2int	Cycles	Speedup
C3X/4X	17	48	74	64	2	1	**4,530**	**1**
Log	21	44	74	64	2	1	**1,048**	**3.9**
Log/par	21	44	24	16	2	1	**950**	**4.3**

Table 4. Clock cycle counts (LRLS algorithm) TMS320C30/40 in comparison with FPGA.

	add	sub	mul	Div	sqrt	int2*	*2int	Cycles	Speedup
C3X/4X	18	72	167	26	50	2	1	**7,246**	**1**
Log	24	66	167	26	50	2	1	**1,403**	**4.6**
Log/par	24	66	48	0	16	2	1	**1,224**	**5.3**

Table 5. Clock cycle counts (NLRLS algorithm) TMS320C30/40 in comparison with FPGA.

Tables 4-5 compare the clock cycles for processing of one input/output sample for the 8-th order filter.

The last line of the tables, denoted Log/par, indicates additional savings and speedup gained by parallel execution of mul, div and sqrt operations in the FPGA. This is illustrated in the following example.

```
par

{ lsub(lxifmin[lj], ltemp2, lxifmin[lj+1],zsl);

   { ltemp5= lm(lkappab[lj], lef);

     ltemp6= lm(lkappaf[lj], loldeb[lj]);

     ltemp1=ld(leb[lj], lxibmin[lj]);

   }

}
```

This small section of code is taken from the Handel C implementation. It outlines the style of parallel programming of the LNS ALU. The subtract operation `lsub()` executes in parallel with 2 instances of the HW macros for logarithmic multiplication `lm()` and one for division `ld()`. It is possible, because the log. mult is just 32bit integer add and divide is 32bit integer subtract. Macro `lsub()` is in fact an interface to a parallel HW module. The HW receives operands through a set of channels. After 9 to 12 clock cycles, the 32bit LNS results and status are returned by a second set of channels. We "programme" the Lattice algorithm as a HW module communicating with the large single LNS ADD/SUB module. All other operations are created in parallel, distributed logic. Our Virtex XCV2000E-6 implementations of LRLS and NLRLS algorithms work at 45MHz clocks and takes 22% and 27% of slices respectively on this device.

Algorithms were coded in Handel-C 2.1 and Celoxica DK1. The reported performance has been achieved by this path:

1. Celoxica DK1 (using the Handel C2.1 compatible code) with export to VHDL.
2. Synplify 5.3 from Synplicity to create EDIF.
3. XILINX Alliance 3.3i tools to place and route from the EDIF netlist for the FPGAs.
4. The Virtex XCV2000E-6 on the RC1000 board [6] was used for the implementation.
5. MSVC code was used for interfacing of RC1000 board to Matlab. See [5] for details.

Conclusions

The LNS implementation of the LRLS algorithms in an FPGA offers better speed than C30/C40 DSP floating-point and provides a low-cost, efficient solution for different system-on-chip applications. The resulting RLS Lattice cores operate with 24-bit precision fixed point input/output signals. Therefore, the internal conversion to the log domain and the internal LNS operations can be hidden from the user. Our Virtex XCV2000E-6 implementation works with 45MHz clocks and if compared with 50MHz TI C30/C40 DSP, it provides significant speedup without any loss of precision.

The analyzed 8-th order Normalised RLS Lattice filter works on this FPGA device at 36.7 kHz, while 50MHz C30/C40 allows just 6.9 kHz. This gives the sustained performance 12 Mflops for Virtex (input domain conversion is counted just one operation) and 2.3 Mflops for C30. Both RLS lattice algorithms have efficient LNS implementation because of numerous divisions or square-roots operations. The un-normalised LRLS algorithm is less complex than the normalized one. However, the sampling rate is just about 20% faster (47kHz for the same FPGA). We have demonstrated that the Normalised RLS Lattice has superior robustness to the non-persistent excitation. These algorithms could be used in applications like echo cancellation, noise reduction, channel equalization. Our future work will be focused in implementing these algorithms using multiple pipelined logarithmic ALUs.

Acknowledgment

The authors wish to thank the referees for their helpful suggestions. This work has been performed under the EU ESPRIT 33544 HSLA Long-term research project (http://napier.ncl.ac.uk/HSLA), coordinated by the University of Newcastle, UK.

References

[1] J.N. Coleman, E.I.Chester, 'A 32-bit Logarithmic Arithmetic Unit and Its Performance Compared to Floating-Point', *14th Symposium on Computer Arithmetic',* Adelaide, April 1999

[2] Paulo S.R. Diniz, Algorithms and Practical Implementation, Kluwer Academic Publishers, 1997

[3] J.N.Coleman, E.Chester, C.Softley and J.Kadlec "Arithmetic on the European Logarithmic Microprocessor", IEEE Trans. Comput. Special Edition on Computer Arithmetic, July 2000. Vol. 49, No. 7, p702-715.

[4] Coleman J. N., Kadlec J.: Extended Precision Logarithmic Arithmetics. In Proceedings of the 34-th IEEE Asilomar Conference on Signals, Systems and Computers, Monterey USA. November 2000.

[5] J. Kadlec, A. Hermanek, Ch.Softley, R. Matousek, M. Licko "32-bit Logarithmic ALU for Handel C 2.1 and Celoxica DK1 (53 MHz for XCV2000E-6 based RC1000 board)"
Results will be presented at Celoxica user conference (In Stratford, UK, 2-3. April, 2001. Download from: http://www.celoxica.com/programs/university/academic_papers.htm

[6] RC1000-PP Hardware Reference Manual, Celoxica, United Kingdom
http://www.celoxica.com/products/boards/DATRHD001.2.pdf

[7] R.L. Walke, R.W.M.Smith, G. Lightbody, "20 GFLOPS QR processor on a Xilinx Virtex-E FPGA", SPIE, San Diego, 2000, U.S.A

[8] R.L. Walke, J. Dudley, "An FPGA based digital radar receiver for Soft Radar", 34th Asilomar Conference on Signals, Systems, and Computers, Monterey, 2000, California, U.S.A

The codes for the proposed algorithms can be obtained from http://falbu.50webs.com/List_of_publications_lns.htm

The reference for the paper is:

F. Albu, J. Kadlec, C. Softley, R. Matousek,, A. Hermanek, A. Fagan, N. Coleman, "Implementation of (Normalized) RLS Lattice on VIRTEX", Field Programmable Logic and Applications, Gordon Brebner and Roger Woods Editors, pp. 91-100, FPL 2001, Belfast, Northern Ireland, UK