Reconfigurable Quaternion LMS

Alin Tisan, *Member, IEEE,* and Clive Cheong Took, *Senior Member, IEEE*

***Abstract*—The quaternion based least mean square (QLMS) algorithm has found more and more applications since its design in 2009. However, its deployment on edge devices still does not meet the real-time constraints nor the complex design methodology requirements (e.g. parallel computations). We address these issues by proposing a reconfigurable hardware architecture that can be rapid FPGA implemented. For rigour, general formulae to determine the required number of hardware resources prior to implementation are derived. Analyses of the factors affecting the performance of QLMS are provided. Simulations and FPGA implementation reports of the proposed QLMS algorithm applied to real-world quaternion based 3-D orientation data confirm the superiority of our approach over the original QLMS.**

***Index Terms*—Quaternion, Least Mean Square algorithm, FPGA, 3-D orientation.**

I. Introduction

FIELD-PROGRAMMABLE gate arrays (FPGAs) offer us, perhaps, the most straightforward means to translate from an engineering innovation to an industrial implementation. This is due to the advantages brought by its reconfigurability, massive parallel data computation, or real-time processing suitability. As such, these advantages have been leveraged in industrial informatics areas such as machine learning [1], control engineering [2], or signal processing [3].

The least mean square (LMS) algorithm has been widely exploited in these informatics areas. The LMS algorithm, however, cannot cope with data that are multi-dimensional. To this end, the Quaternion LMS (QLMS) algorithm was proposed to cater for 3D and 4D processes [4]. The simplicity of QLMS (as shown in Algorithm 1) means it has been used in various applications such as in audio [5], gait analysis [6], renewable energy [7], and biomedicine [8], [9], to name a few.

Despite its successful adoption by the research community, the QLMS algorithm has one major weakness: its computational cost - which is a major problem to its wider adoption. To this end, there have been attempts to reduce its computational cost by more than 50% [10], [11]. Yet, these computationally cheaper versions of QLMS are not suitable for deployment on an industrial scale due to the following factors: 1) their computational costs are still not comparable to that of four single channel LMS algorithms; 2) they have been designed to operate on central processing units (CPUs) rather than on specialised hardware such as FPGAs [12]; and 3) they cannot offer massively parallel data computation - which is a trend in industrial informatics.

For the deployment of QLMS algorithm on industrial informatics applications, we therefore propose the reconfigurable FPGA based QLMS (fQLMS) that takes into account the constraints and advantages of FPGAs into its design. As such, this work investigates the trade offs between accuracy, data parallellisation, and parameters of the QLMS algorithm (e.g. filter length of the adaptive filter). More specifically, the rest of the paper describes the quaternions and the QLMS algorithms in Section II, the proposed overall fQLMS architecture and its block constituents in Section III, the results of the hardware implementation in Section IV, and the main conclusions.

Both authors are with the Department of Electronic Engineering, Royal Holloway University of London, Egham, TW20 0EX, UK, E-mails: {alin.tisan, clive.cheongtook}@rhul.ac.uk.

II. Quaternions and QLMS

Quaternions are extensions of complex numbers to four dimensions, and a quaternion variable can be described as (1)

$$q = \Re[q] + \Im[q] = q_1 + q_2\mathbf{i} + q_3\mathbf{j} + q_4\mathbf{k} \tag{1}$$

The scalar (real) part is denoted by $q_1 = \Re[q]$, and the vector (imaginary) part (also known as pure quaternion) $\Im[q] = q_2\mathbf{i} + q_3\mathbf{j} + q_4\mathbf{k}$ comprises of three imaginary components. Thus, the quaternion conjugate is given by (2)

$$q^* = \Re[q] - \Im[q] = q_1 - q_2\mathbf{i} - q_3\mathbf{j} - q_4\mathbf{k} \tag{2}$$

The quaternion product between two variables x and y is noncommutative (i.e. $xy \neq yx$) and is given by (3)

$$\begin{aligned} xy = &(x_1y_1 - x_2y_2 - x_3y_3 - x_4y_4) \\ &+ (x_1y_2 + x_2y_1 + x_3y_4 - x_4y_3)\mathbf{i} \\ &+ (x_1y_3 - x_2y_4 + x_3y_1 + x_4y_2)\mathbf{j} \\ &+ (x_1y_4 + x_2y_3 - x_3y_2 + x_4y_1)\mathbf{k} \end{aligned} \tag{3}$$

The quaternion product in (3) incurs a high computational cost, i.e. 16 multiplications and 12 additions. Thus, its computation represents the main stumbling block to the use of quaternions in industrial applications. To alleviate or circumvent the high computational cost of quaternion product, there have been various propositions to address the computation of quaternions [13]–[15]. To this end, we exploit the quaternion product algorithm proposed in [13] as well as take advantage of hardware architecture of FPGA to reduce the computational cost of the QLMS algorithm. QLMS listed in Algorithm 1, shown in (4 - 6), maps the quaternion input $\mathbf{x}(n)$ onto the estimated quaternion output $\mathbf{y}(n)$ via the quaternion weights $\mathbf{w}(n)$. The update of these weights (or filter taps) are governed by the stepsize μ and steered by the quaternion error $e(n)$, which measures the difference between the desired quaternion output $d(n)$ and the estimated quaternion output $y(n)$. To illustrate the computational advantage of our proposed method, Table I compares all variants of QLMS algorithm. Note that the QLMS algorithm proposed in this paper incurs only a third of the computational costs of the original QLMS [4] (when comparing the costlier multiplication operation), and is thus suitable for industrial applications.

Algorithm 1 : Quaternion LMS for 3D and 4D Processes [4]

Initialisation
$\mathbf{w}(0) = \mathbf{0}$
For all $n \geq 1$

$$\text{Output:} \quad y(n) = \mathbf{w}^{\mathrm{H}}(n)\mathbf{x}(n) = \sum_{\ell=1}^{L} w_{\ell}^{*}(n)x(n-\ell+1) \tag{4}$$

$$\text{Error:} \quad e(n) = d(n) - y(n) \tag{5}$$

$$\text{Update weights:} \quad \mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathbf{x}(n)e^{*}(n) \tag{6}$$

where L, n and ℓ, $(\cdot)^{\mathrm{H}}$ denote respectively the number of weight coefficients (or filter taps), the time index, the ℓth filter tap, and the conjugate transpose operator.

TABLE I: Computational complexities of Quaternion LMS algorithms. Additions and subtractions are considered the same.

Algorithms	Multiplications	Additions
Original QLMS [4]	$48L$	$56L$
QLMS by Neto and Nascimento [10]	$32L+4$	$32L$
QLMS by Xiang et al. [11]	$20L+4$	$64L$
Proposed fQLMS	$16L$+4	$84L+8$

The computational cost of our proposed fQLMS is twice the cost of four single-channel LMS, but half the cost of the multichannel LMS [10]. In contrast to other methods [4], [10], [11], fQLMS does not use the Matlab quaternion toolbox [12] and its computational cost can be further reduced through the exploitation of the hardware implementation, as described in the next sections.

III. The Hardware Design of the Quaternion LMS

The nature of the QLMS algorithm [such as Equation (4)] provides a high degree of parallelism which is successfully exploited and deployed when implemented on the FPGA architecture. In this section, the design of an architecture with a quaternion component parallelism level is firstly explained, followed by its implementation results and discussions in Section IV. The proposed architecture was developed in Xilinx's Model Composer Add-on for MATLAB, a tool that enables low level model design within the MathWorks Simulink [16].

Fig. 1 shows the overall architecture[1] of QLMS on our FPGA design. The main computational blocks are described as follows: the DATA IN and DATA OUT blocks are used as hardware interfaces for defining the QLMS input – output ports size and data representation, and the TAPn, ΣTAPn and ERROR blocks implement the equations (4)-(6) described in Section II. The Goto block (white) sends the data calculated in the ERROR block (CONJ_ERROR) to the TAPn blocks.

[1]The blocks in Fig. 1 have the hardware constraints (data representation and synchronization) accurately defined to automatically comply with the fQLMS filter length and the data magnitude.

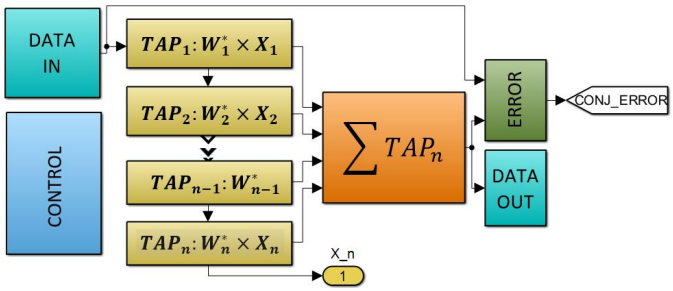

Fig. 1: Overall hardware architecture of the n-TAP fQLMS

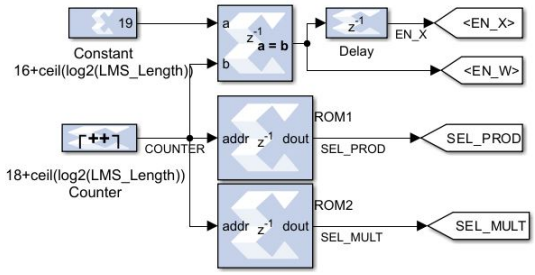

Fig. 2: The Control Block architecture

More details on the main constituent blocks are provided in the following subsections.

A. The Control Block

The CONTROL BLOCK shown in Fig. 2 calculates for a given QLMS length (L): (i) the T_W time (10) to update the filter with the newly calculated weights (block W_TAP in Fig.3) triggered by EN_W, (ii) the T_X time (9) to load into the filter new input samples (block X_TAP in Fig.3) triggered by EN_X, and (iii) pipelines the quaternion multiplication by selecting the quaternion product inputs via SEL_PROD (Fig.3), and the multiplication inputs via SEL_MULT (Fig.5).

The Counter's current value is compared with Delay_L (7) to generate the EN_X and EN_W signals (shown in Fig.2 and Fig.3). It also provides the address for the pre-loaded values of the signals SEL_PROD and SEL_MULT stored in the memory ROM1 (11) and ROM2 (12).

$$Delay\_L = FixedDelay + ceil(log_2(L)) \tag{7}$$

$$COUNT = Delay\_L + 2 \tag{8}$$

$$T\_X = Delay\_L + 2 \tag{9}$$

$$T\_W = Delay\_L + 1 \tag{10}$$

$$ROM1 = [zeros(1, T\_X), ones(1, T\_X)] \tag{11}$$

$$ROM2 = [0, 1, 1, zeros(1, T\_W), 0, 1, 1, zeros(1, T\_W)] \tag{12}$$

where FixedDelay = 16 (value determined according to the data path and QLMS block delays), COUNT represents the upper counting limit value stored in the COUNTER, and *ceil* rounds up to the nearest integer.

B. The TAPn processing unit

The architecture of the TAPn processing unit implements in hardware the main QLMS calculation tasks. Its main blocks,

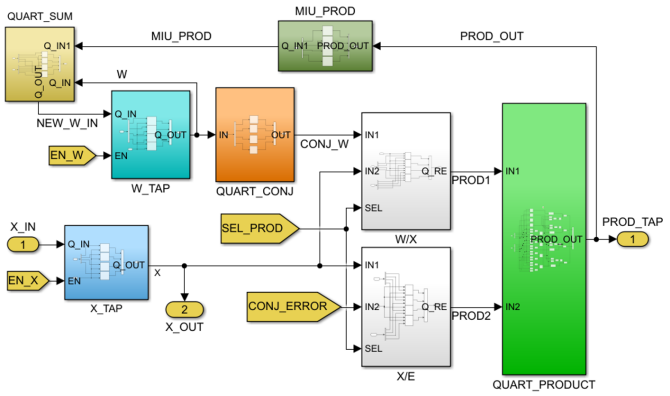

Fig. 3: The block architecture of the TAPn processing unit

shown in Fig. 3, calculates the $w_\ell^*(n-1)x(n-\ell+1)$ terms in (4) and the updated quaternion weights $w_(n)$ in (6).

The input signal X_IN and the QLMS weights NEW_W_IN are stored respectively in the X_TAP block (blue) and W_TAP block (cyan), and then updated when triggered by the EN_X and EN_W. Next, the weights are conjugated in the QUART_CONJ block and multiplied with the input signal in the quaternion multiplier QUART_PRODUCT block (described in Section III-C). The W_X and X_E blocks act as multiplexers controlled by SEL_PROD, selecting the signals to be multiplied by the QUART_PRODUCT block: CONJ_W×X or X×CONJ_ERROR as required in w^*x (4) and xe^* (6).

The result is then passed to the ΣTAPn block (described in Section III-D) to calculate the predicted output value. After the conjugated error is calculated in the ERROR block (described in Section III-E), its quaternion product with the input signal is multiplied with the learning rate in the MIU_PROD block. The result is summed with the current weight in QUART_SUM block to obtain the updated weight NEW_W_IN (6).

C. The quaternion multiplier QUART_PRODUCT block

The algorithm to compute the quaternion product adopted in this paper follows Equations (13)-(14) proposed in [13]. The advantage lies in the lower number of multiplications required than the default quaternion product definition given in (3). However, the reduction (8 from 16) comes at the expense of increasing the additions from 12 to 40. The implications are discussed in Section IV.

$$\begin{aligned} xy = &(+2T_1 - (T_5+T_6+T_7+T_8)/4) \\ &+ (-2T_2 + (T_5+T_6-T_7-T_8)/4)\mathbf{i} \\ &+ (-2T_3 + (T_5-T_6+T_7-T_8)/4)\mathbf{j} \\ &+ (-2T_4 + (T_5-T_6-T_7+T_8)/4)\mathbf{k} \end{aligned} \tag{13}$$

where

$$\begin{aligned} &T_1 = x_1y_1;\ T_2 = x_4y_3;\ T_3 = x_2y_4;\ T_4 = x_3y_2; \\ &T_5 = (x_1+x_2+x_3+x_4)(y_1+y_2+y_3+y_4) \\ &T_6 = (x_1+x_2-x_3-x_4)(y_1+y_2-y_3-y_4) \\ &T_7 = (x_1-x_2+x_3-x_4)(y_1-y_2+y_3-y_4) \\ &T_8 = (x_1-x_2-x_3+x_4)(y_1-y_2-y_3+y_4) \end{aligned} \tag{14}$$

Equations (13) and (14) can be rearranged to : (i) improve the data accuracy, by dividing the values before multiplication (the division by 4 is moved from (13) to (16) such that both factors are divided by 2 before the multiplication) and (ii) form recursive addition/subtraction patterns, by rearranging the terms in (14). The result of these algorithmic considerations are described in (16).

$$\begin{aligned} xy_{\text{proposed}} = &(+2T_1 - ((T_5+T_6)+(T_7+T_8))) \\ &+ (-2T_2 + ((T_5+T_6)-(T_7+T_8)))\mathbf{i} \\ &+ (-2T_3 + ((T_5-T_6)+(T_7-T_8)))\mathbf{j} \\ &+ (-2T_4 + ((T_5-T_6)-(T_7-T_8)))\mathbf{k} \end{aligned} \tag{15}$$

where

$$\begin{aligned} &T_1 = x_1y_1;\ T_2 = x_4y_3;\ T_3 = x_2y_4;\ T_4 = x_3y_2; \\ &T_5 = \frac{(x_1+x_2)+(x_3+x_4)}{2}\frac{(y_1+y_2)+(y_3+y_4)}{2} \\ &T_6 = \frac{(x_1+x_2)-(x_3+x_4)}{2}\frac{(y_1+y_2)-(y_3+y_4)}{2} \\ &T_7 = \frac{(x_1-x_2)+(x_3-x_4)}{2}\frac{(y_1-y_2)+(y_3-y_4)}{2} \\ &T_8 = \frac{(x_1-x_2)-(x_3-x_4)}{2}\frac{(y_1-y_2)-(y_3-y_4)}{2} \end{aligned} \tag{16}$$

The computational model of (15)-(16) is shown in Fig. 4. It can be observed that certain addition/subtraction and multipliers blocks are grouped under the AddSub System (coloured in purple) and Multiplier System (light green). Due to the sequential processing through the computational blocks shown Fig. 4, the number of multipliers can be further decreased from 8 to 4 by reusing the hardware resources grouped in the *Multiplier System* without additional clock cycles. The *Multiplier System* inputs, are selected using 2 multiplexers (light blue), Fig. 5. Here, SEL_MULT (white) generates the multiplier inputs selection signal defined in the *CONTROL BLOCK* (as shown in Section III-A), X0.5 implements the 1 bit left shift operation (yellow) and X2 implements a 2 bit right shift operation. Moreover, a further addition/subtraction blocks reduction, from 28 to 12, can be achieved through an additional reuse of the AddSub block (purple block, containing 8 addition/subtraction blocks). This incurs an increase of the number of clock cycles by 8, due to the 4 extra clock cycles delay[2] for each of the two 4-element multiplications done in a quaternion product: one multiplication to calculate the T_1 to T_4 terms and second multiplication for calculating the T_5 to T_8 terms, as shown in (16). Considering the number of addition/subtraction and multiplication blocks that are required to implement the quaternion product, the increased availability of the addition/subtraction blocks over the multiplication blocks in the FPGA circuit used, and the addition of zero extra clock cycles when pipelining the multiplication, in the rest of the paper only the *Multiplier System* block will be reused. The hardware implication is discussed in Section IV.

D. The ΣTAP$_n$ block

The ΣTAP$_n$ block calculates the weighted sum of the signal sampled across the entire filter length (L) and provides the

[2]Two for AddSub and two for MUX blocks

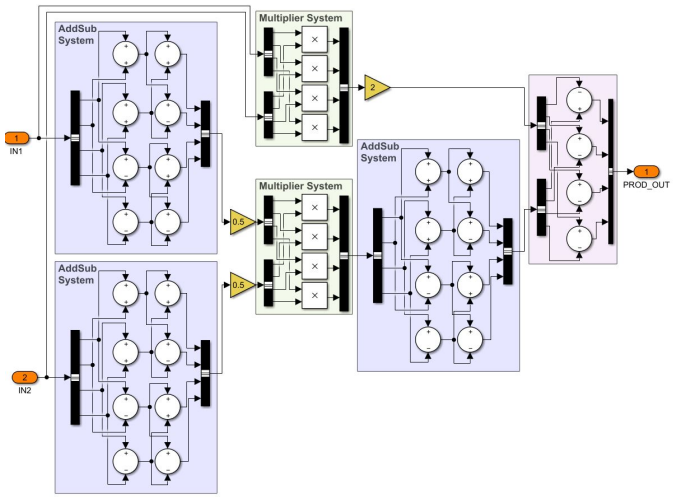

Fig. 4: The quaternion multiplication algorithm shown as recursive patterns grouped under the AddSub System (purple) and Multiplier System (light green) blocks

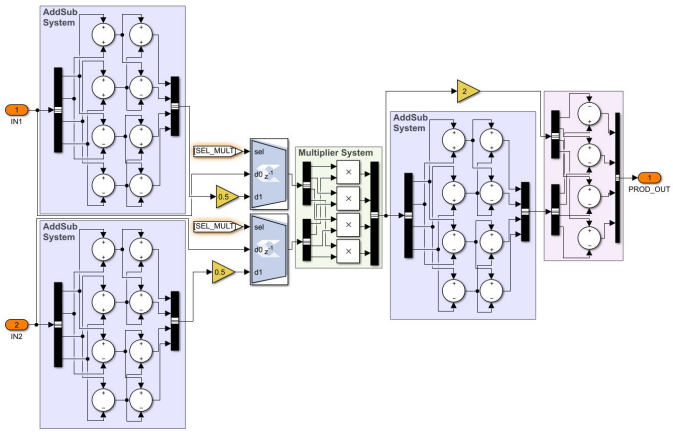

Fig. 5: The block diagram of the quaternion multiplication algorithm

predicted output value. It consists of Xilinx ADD blocks for each quaternion component (the QUART_SUM blocks) which support only two inputs at the time. Therefor the sum of all TAPn outputs is done in a number of sequential stages equal to *ceil(log$_2$(L))* as shown in Fig. 6. For $L \neq 2^n$, the missing abs(L - 2^n) addition terms will be each replaced with an 1 clock cycle delay block. The predicted output value, *SUM_OUT*, is made available for external interfacing through the *DATA_OUT* block shown in magenta in Fig.1.

E. The ERROR function block

The ERROR function block calculates target - estimated difference (5), and its conjugate value (CONJ_ERROR) is further re-used as the CONJ_ERROR input to the multiplexer X/E (as shown in Fig. 3).

IV. The FPGA implementation of the QLMS. Results & Discussions

The FPGA implementation of the QLMS requires a number of hardware resources mainly determined by the length of

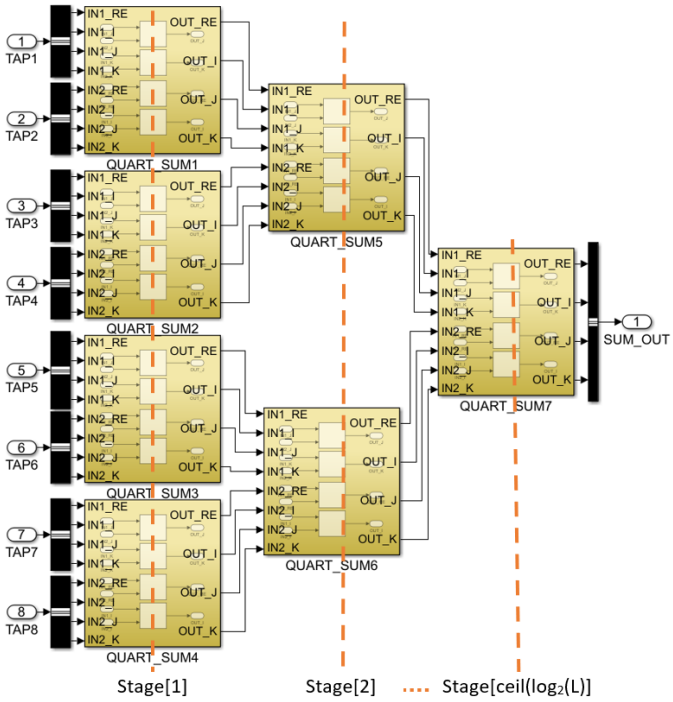

Fig. 6: The block diagram of ΣTAPn module showing the addition of $L = 8$ TAPn elements done in ceil(log$_2$(8)) = 3 stages

the QLMS and the bit resolution of the data representation (the quantization error), both dictated by the application. In this work, the QLMS algorithm implemented in FPGA will be tested against two applications: the most well-known application of quaternions i.e. 3D orientations in which the algorithm is used to predict 3D positions and as Lowpass Filter Orientation for data denoising.

A. 3D position prediction

In the first application considered, i.e. the 3D orientation application, the tracking of 3D position was simulated to demonstrate the usefulness of reconfigurable QLMS (fQLMS) in a real-world setting. Indeed, the new 3D position $p^\dagger(n)$ can be obtained from an old 3D position $p(n)$, following a 3D rotation by $q(n)$ as:

$$p^\dagger(n) = q(n)p(n)q^*(n) \tag{17}$$

For rigour, fQLMS was used to track the new 3D position $p^\dagger(n+10)$ in a 10 step-ahead prediction based on its input which was comprised of the current and previous 3D rotations $\big[q(n), \ldots, q(n-L+1)\big]$. To assess the robustness of our fQLMS, white Gaussian noise was added to its input at a signal-to-noise ratio of 20 dB.

The simulation results of the fQLMS predicting the $p^\dagger(n+10)$ position (blue) are compared with the actual 3D position (red) in Fig. 7; these are shown on the unit sphere on the right and on the three different axes on the left. The bottom left plot of Fig. 7 shows the convergence of our proposed method, where $e(n)$ denotes the difference between the 10 steps

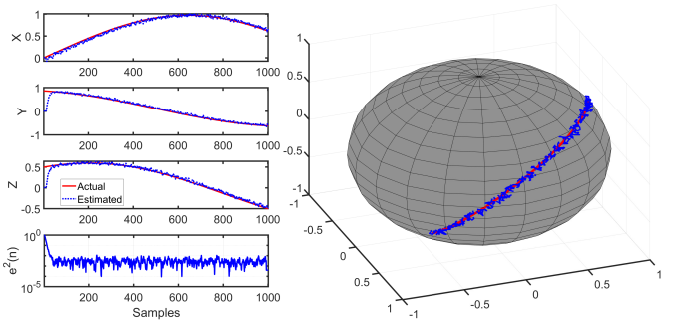

Fig. 7: 3D rotation simulation results

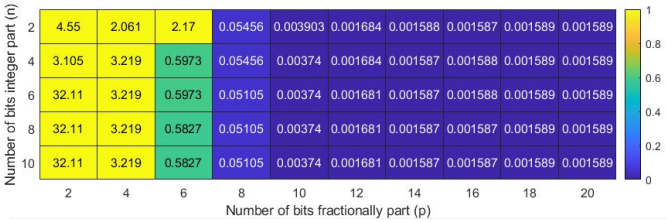

Fig. 8: The relative RMSE (rRMSE) value of the proposed $L = 8$ fQLMS output function of n and p

ahead desired quaternion position and the predicted quaternion position calculated from the full quaternion rotation.

The success of a hardware implementation depends on the availability of the resources required on a given FPGA. As their number vary with the QLMS characteristics, an in-depth analysis of hardware resources needed vs QLMS features is given in the following subsections.

1) Hardware resources vs QLMS data type: To highlight the influence of the data type on the performance of the QLMS, the relative *RMSE* (*rRMSE*) across all 4 quaternion components (18) was calculated as the root mean square error (*RMSE*) normalised by the root mean square (*RMS*) value of the quaternion signal. The *rRMSE* values are shown in Fig. 8 as a function of the number of bits allocated for data implementation *sn.p*, with *s* representing the sign, *n* the number of bits representing the integer part, and *p* the number bits allocated for the fractionally part of the number.

$$rRMSE = \frac{RMSE}{RMS} = \sqrt{\frac{\sum_{n=1}^{N}\left(y_n^{\text{double}} - y_n^{\text{sn.p}}\right)^2}{\sum_{n=1}^{N}\left(y_n^{\text{double}}\right)^2}} \tag{18}$$

where y_n^{double} denotes the Matlab quaternion toolbox double data type [12] and $y_n^{\text{sn.p}}$ represents the proposed sn.p fixed point data type implementation. As expected, the worst *rRMSE* value corresponds to the lowest bit resolution $sn.p = s2.2$, as shown in Fig. 8. On the other hand, *rRMSE* decreases exponentially as n and p increases, reaching a plateau of 0.1% dissimilarity for $n \geq 2$ and $p \geq 12$. Ideally, n and p should be chosen so that $rRMSE = 0$. Yet, this requires higher bit resolutions, which in turn demands more hardware resources. Therefore, a trade-off between the *rRMSE* value and the hardware resources required must first be found in terms of n and p values. It can be noticed from Fig.8 that the minimum *sn.p* resolution that ensure the lowest *rRMSE* value of 0.1% is *s.2.12*. The hardware resources used (especially the dedicated DSP48 multipliers) strongly depended on the data representation adopted. However, due to the multiplier's 18 x 25-bit input architecture [17], the QLMS implementation requires a constant number of 32 DSP48 blocks for $L = 8$ QLMS, for any data representation under 18-bit, as shown in Table II for *s2.8* and *s2.12*. Nevertheless, the number of DSPs will increase proportionally with the quotient of *(data representation)/18*, as bit resolution is increased: e.g: for an $L = 8$ QLMS with 19 to 36-bit resolution, there are required $(quote(36/18) * 32 = 64)$ DSP48 blocks. Consequently, in the rest of the paper, the implementations will be made in the *s2.12* resolution, to ensure the highest accuracy/hardware resources ratio.

TABLE II: Hardware resources used vs. data representation for $L = 8$ QLMS implementation

Data representation	Hardware resources				
	LUT	FFs	BRAMs	LUTRAM	DSP
s2.8	4514	4991	1	160	32
s2.12	6850	7087	1	256	32

TABLE III: Hardware resources vs. QLMS length (L) for s2.12 data representation in Artix 7 (XC7A100T)

L	Hardware resources				
	LUT	FFs	BRAMs	LUTRAM	DSP
8	6850	7087	1	256	32
16	14009	13999	1	512	64
32	27497	27823	1	1024	128
48	41943	41647	1	1536	192
56	49047	48559	1	1792	224
64	*55271*	*55471*	*1*	*2048*	*256*
available	63400	126800	135	19000	240

2) Hardware resources vs QLMS length L: Besides the bit resolution, the hardware resources required for implementing the QLMS model are strongly determined by the QLMS length and the targeted FPGA architecture: i.e. an L-QLMS requires L-TAPn units (as detailed in Section III-B), and each of them uses 4 DSP48 blocks, where each DSP48 block has 25 × 18 bit two's-complement multiplication, 48-bit accumulators and 24-bit add/subtract capabilities. For a complete picture, the numbers of RAM blocks, LUTs, flip-flops (FF) and DSP48 blocks function of L are shown Table III. Based on Table III, it can be inferred formulae to approximate the number of used hardware blocks function of the QLMS length, (19)-(22).

$$\text{LUTs estimated} = 850 \times L \tag{19}$$

$$\text{FFs estimated} = 875 \times L \tag{20}$$

$$\text{LUTRAMs estimated} = 32 \times L \tag{21}$$

$$\text{DSPs estimated} = 4 \times L \tag{22}$$

These formulae give the advantage of predicting the maximum QLMS length that fits an FPGA, prior to its hardware implementation. Consequently, it can be shown that the maximum QLMS lengths that can be implemented in an Artix7-XC7A100T is 60. Note that the resources used for implementing the L64-QLMS (Table III) were obtained for

TABLE IV: Ideal μ vs. QLMS length (L) and minimum data resolution (sn.p) for a relative error $< 0.05\%$

L	8	32	64	128	200	256	515	1024
$\mu \times 10^{-2}$	33	26	15	9	5	3	1	1
min sn.p	1.10	1.10	1.10	1.10	1.10	1.10	1.10	1.10

TABLE V: Maximum data path delay vs. QLMS length (L)

L	8	16	32	48	56	64
Data path delay (ns)	9.57	10.08	10.05	10.07	11.02	14.7

a different FPGA than XC7A100T (hence the text in italic). Here, it was used Artix-7 XC7A200T with 740 DSP blocks.

3) Hardware resources vs QLMS learning factor: Another possible limiting factor in deploying large QLMS filters is the learning factor μ implementation, which, in order to keep the filter stable, takes values that decrees with the QLMS length. The appropriate data resolution for μ implementation for the 3D rotation application is shown in Table IV. For this particular application, although μ varies, the required data representation remains constant to 11 bits and therefore we were able to exploit (19)-(22) to calculate the necessary resources for the QLMS implementation.

4) Data Throughput: The data throughput of the QLMS depends on the system clock frequency and the QLMS length. The minimum time period to calculate the QLMS output is determined by the COUNT value, which is a function of $\log_2(L)$ (8). Due to the concurrent nature of the implementation, the QLMS length has a little influence on the maximum data path delay, but it greatly determines the throughput (due to the sequential part of the QLMS algorithm) and the hardware resources used to accommodate the concurrency, TableV.

The implementation timing report, shown in Table VI, gives for $L = 8$ a minimum clock period of $9.57ns$, which ensures a data throughput of $(19 + log_2(8)) \times 10ns = 220ns$ or 4.7M samples per second. Fig. 9 shows the benchmark processing time for the QLMS calculated using the Matlab's quaternion toolbox [12] (on the green axis) and the cost effective QLMS algorithm reported in [11] (on the red axis), where the proposed fQLMS implemented in FPGA is shown on the blue axis. The significantly lower running time of the FPGA implementation (in order of hundreds of nanoseconds) over the 4 cores at 3.60 GHZ CPU implementation (in order of seconds) confirms the computational advantage of our proposed method.

B. Filtering of noisy 3D rotations

The second application addressed the denoising of the 3D rotation measurements, i.e. $q(n)$ in (17), whereas the aim of the first application was to predict the 3D coordinates in $p^{\dagger}(n)$ in (17). As such, Fig. 11 shows the 3 rotations (in degrees) about the three axes $Z - Y - X$ for the second application, whereas the left plots of Fig. 7 shows the 3D positions in the sphere for the first application. In the second application, the adaptive filter takes in $x(n)$ as the noisy 3D rotation measurement inputs, i.e.

$$x(n) = q(n) + \eta(n) \tag{23}$$

TABLE VI: Maximum data path delay vs. data representation for $L = 8$ QLMS implementation

sn.p	s2.8	s2.10	s2.12	s2.14
Data path delay (ns)	9.57	9.63	9.67	9.82

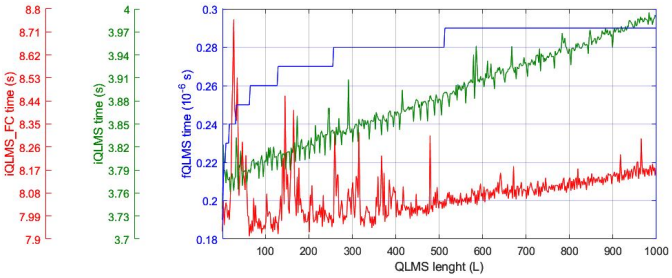

Fig. 9: Processing time for iQLMS (green) and iQLMS_FC (red) [11] vs proposed fQLMS (blue) against QLMS length for a system clock of 10ns.

The noise $\eta(n)$ was added to 3D rotation $q(n)$ at a signal-to-noise ratio of 10 dB. As such, the aim is to denoise $x(n)$ so that its output $y(n)$ is as close as possible to $q(n)$. Fig. 11 shows the noisy 3D rotation measurements in brown, and its filtered counterpart in blue (which was denoised by our method).

The hardware resources constraint and accuracy: Although the main topology of the fQLMS remained the same, the purpose of the second simulation is to demonstrate another application and the method's adaptability to another set of data. To ensure the same level of accuracy as the one considered in Section IV-A, the hardware implementation of the QLMS algorithm in the FPGA required a specific data representation. For this, as in the case of the 3D position prediction, there were a series of FPGA implementations to calculate the *rRMSE* between the signal filtered using the QLMS implemented in Matlab Quaternion Toolbox and the QLMS implemented in FPGA, for a range of number of bits used to represent the integer part (*n*) and the fractionally part (*p*) of the value. After analysing the *rRMSE* results presented in Fig. 10, it can be concluded that for an *rRMSE* $< 0.1\%$, a minimum of 16-bit representation (*n = 4* and *p = 12*) was required. This had an impact on the hardware resource used for FPGA implementation, presented in Fig. VII. Comparing them with the implementation results shown in Table II and the equations (19)-(22) it is noteworthy to see that the resources used for the *s4.10* implementation were similar to the ones reported

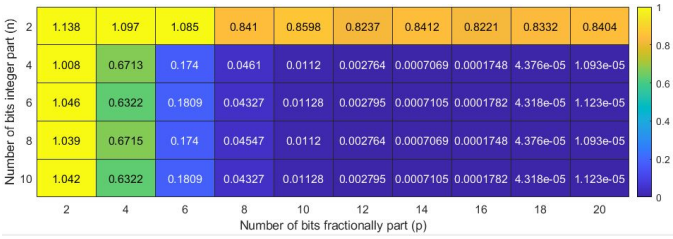

Fig. 10: The relative RMSE (rRMSE) value of the $L = 8$ fQLMS for data denoising vs data representation (*n* and *p*)

TABLE VII: Hardware resources used vs. data representation for $L = 8$ QLMS implementation

Data representation	Hardware resources LUT	FFs	BRAMs	LUTRAM	DSP
s4.10	7025	7151	1	256	32
s4.12	7801	8199	1	256	32

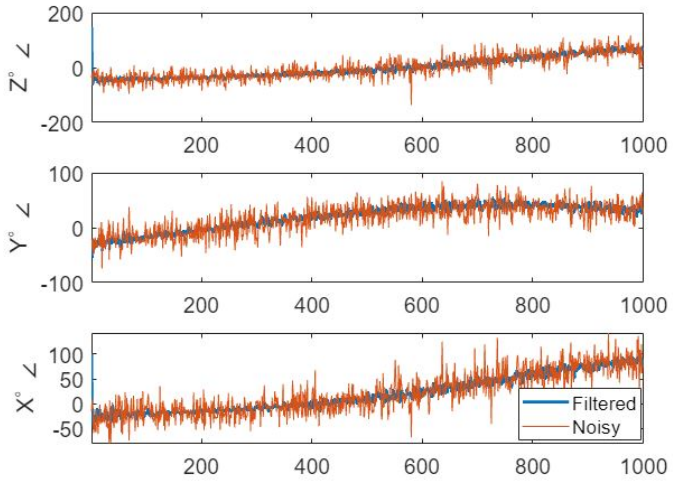

Fig. 11: The filtering results of the $L = 8$ fQLMS applied to rotational quaternion data

for s2.12 implementation, as the total number of bits used in data representation remained the same, i.e. 14. (Note: the slight differences come from the use of a different learning stepsize μ values, here it was 0.1). As expected, the number of used DSP blocks remained the same, i.e. 32, for *s4.10* and *s4.12*, due to the block's 25 × 18 bit two's-complement multiplication capability. The filtering results of applying the $L = 8$ fQLMS to noisy quaternion rotational data is shown in Fig. 11. They demonstrate the efficiency of the hardware fQLMS applied to noisy rotational data and the capability of accommodating variable algorithm parameters (such as *sn.p* data representation, or the μ values) and the inherent hardware implementation constraints.

V. Conclusion

We have proposed a hardware architecture for the QLMS algorithm that can leverage the parallel processing of FPGAs. The model fQLMS is adaptable and easily reconfigurable to accommodate hardware implementation constraints (hardware resources and data representation) for variable QLMS parameters (QLMS length and accuracy). fQLMS exhibited a massive 10^6 fold decrease of the processing time over QLMS. No fully reconfigurable FPGA-based hardware with prior implementation FPGA resources estimators have yet been reported. As such, the model is intended as a tool for FPGA designers to rapid prototype real-time applications. For rigour, various analyses such as the output sampling time (8) and the hardware resources used (19)-(22) have been derived to facilitate the pre-implementation design of fQLMS for other industrial applications.

References

[1] A. Tisan and J. Chin, "An enduser platform for FPGA-based design and rapid prototyping of feedForward artificial neural networks with on-chip back propagation learning," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 3, pp. 1124–1133, 2016.

[2] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in industrial control applications," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 2, pp. 224 – 243, 2011.

[3] R. Morales-Caporal, E. B. Huerta, C. H. Flores, M. A. A. López, and M. Pacas, "Transducerless acquisition of the rotor position for predictive torque controlled PM synchronous machines based on a DSP-FPGA digital system," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 2, pp. 799–807, 2013.

[4] C. Cheong Took and D. P. Mandic, "The Quaternion LMS Algorithm for Adaptive Filtering of Hypercomplex Real World Processes," *IEEE Transactions on Signal Processing*, vol. 57, pp. 1316 – 1327, 2009.

[5] J. Cao, J. Liu, J. Wang, and X. Lai, "Acoustic vector sensor: reviews and future perspectives," *IET Signal Processing*, vol. 11, pp. 1–9, 2017.

[6] A. Bravi and A. M. Sabatini, "A multidimensional approach to postural sway modeling," in *2010 IEEE International Workshop on Medical Measurements and Applications*, pp. 121–124, 2010.

[7] C. C. Took, G. Strbac, K. Aihara, and D. Mandic, "Quaternion-valued short-term joint forecasting of three-dimensional wind and atmospheric parameters," *Renewable Energy*, vol. 36, no. 6, pp. 1754–1760, 2011.

[8] G. Cosma et al., "A survey on computational intelligence approaches for predictive modeling in prostate cancer," *Expert Systems with Applications*, vol. 70, pp. 1–19, 2017.

[9] K. Adhikari, S. Tatinati, W. T. Ang, K. C. Veluvolu, and K. Nazarpour, "A quaternion weighted Fourier linear combiner for modeling physiological tremor," *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 11, pp. 2336–2346, 2016.

[10] F. G. A. Neto and V. H. Nascimento, "Low-complexity quaternion adaptive filters," *arXiv:1410.2854*, Oct. 2014.

[11] M. Xiang, C. Took, and D. P. Mandic, "Cost-effective quaternion minimum mean square error estimation: From widely linear to four-channel processing," *Signal Processing*, vol. 136, pp. 81 – 91, Jul. 2017.

[12] S. Sangwine and N. L. Bihan, "Quaternion and octonion toolbox for matlab." http://qtfm.sourceforge.net/, 2015. Accessed: 2021-06-16.

[13] T. Howell and J. Lafon, "The complexity of the quaternion product," *Technical Report Cornell University (TR 75-245)*, pp. 1–13, 1975.

[14] A. Cariow and G. Cariowa, "An algorithm for dividing quaternions," *arXiv:2009.00425*, Aug. 2020.

[15] J. L. Contreras-Hernandez, D. L. Almanza-Ojeda, S. Ledesma-Orozco, A. Garcia-Perez, R. J. Romero-Troncoso, and M. A. Ibarra-Manzano, "Quaternion signal analysis algorithm for induction motor fault detection," *IEEE Transactions on Industrial Electronics*, vol. 66, no. 11, pp. 8843–8850, 2019.

[16] Xilinx, *Model-Based DSP Design using Add-on for MATLAB and Simulink*, 11 2020. v2020.2 Accessed: 2021-06-16.

[17] Xilinx, *7 Series DSP48E1 Slice User Guide*, 3 2018. v1.10.

Alin Tisan, PhD, MIEEE, CEng, MIET Alin received a BEng (1997) and a MSc (1998) in Physics from Babeş-Bolyai University and a PhD (2008) in neuromorphic systems for olfaction applications from Technical University of Cluj Napoca, RO. He is currently lecturer in the Electronic Engineering Department at Royal Holloway University of London, UK. His research interests include AI on system on chip (neuromorphic software/hardware systems), machine learning for neuronal data, data processing in olfaction, signal processing for gas sensors, IoT with applications in pervasive, unobstructive healthcare and smart homes. He has published over 50 papers and co-chaired several Electronic Systems-on-Chip and Embedded Systems related Technical Tracks and Special Sessions.

Clive Cheong Took, SMIEEE received his PhD degree in signal processing from Cardiff University in 2007. Clive is now a senior lecturer at Royal Holloway, University of London. He was part of the editorial team at IEEE Transactions on Neural Networks and Learning Systems and was awarded the outstanding associate editor in 2019. Clive co-edited special issues on “Deep neural network representation and Generative Adversarial Learning” and on “Deep Representation and Transfer Learning for Smart and Connected Health”. He is currently holds an editorship position at the Elsevier Neural Networks and is an executive committee member of IET Healthcare Network. His research interests lie in neural networks and signal processing.