Optimized UD filtering algorithm for floating-point hardware execution

Rodrigo González
GridTICs and Lab. de Computación Reconfigurable
Universidad Tecnológica Nacional, FRM
Mendoza, Argentina
Email: rodralez@frm.utn.edu.ar

Gustavo Sutter
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Madrid, España
Email: gustavo.sutter@uam.es

Héctor Daniel Patiño
Instituto de Automática
Universidad Nacional de San Juan
San Juan, Argentina
Email: dpatino@inaut.unsj.edu.ar

***Abstract*—The Kalman filter is an effective tool for fusing signals from multiple sources. The UD filtering is a well-known, numerically-stable formulation of the Kalman filter, owing to G.J. Bierman and C. Thornton. The most popular version of this filter is oriented to be executed in a traditional, sequential microprocessor. In this paper a new algorithm for the UD filtering is presented, specially designed for execution in hardware. It is based upon operations involving matrices and vectors, which is a more suitable approach for hardware optimization. To the best of the authors' knowledge, this is the first reported work about a fully UD filtering implementation in hardware with floating-point arithmetics. Since no previous works were found, the sequential UD filtering is synthesized as a benchmark. When compared with this sequential version, the UD filtering for hardware provides a speed-up of ~10x and presents a performance-vs.-area improvement by ~2x.**

***Keywords*—UD filtering, Kalman, Bierman, Thornton, hardware, FPGA, floating-point**

I. Introduction

The Kalman filter (KF) is an algorithm for linear systems that operates recursively on noisy input and output data to produce a statistically optimal estimate of the system states. The extended Kalman filter (EKF) is a nonlinear extension of the KF, which linearizes about an estimate of the current mean and covariance. The EKF uses the same basic equations of the KF. Integrated navigation systems commonly use an EKF to fuse information coming from inertial and aiding sensors to get estimates with few errors when compared with using these sensors in a separately way.

Since the KF must be executed in a finite-precision computer for real-world applications, serious numerical problems may arise [1, Ch. 6]. Several methods prevent the KF from divergence and numerical instability. One of the most convenient methods is the UD filtering [2] [3] because it combines the sequential KF technique and the UD factorization. A sequential KF avoids the matrix inversion of error covariance matrix P, which is a computing expensive operation and may produce severe round-off errors when P is ill-conditioned [4, Ch. 6.1]. In addition, the UD factorization improves the KF numerical precision without increasing the computational cost when compared with other square root filters [4, Ch. 6.4]. The main idea behind this technique is to represent P as $P = UDU^T$, with U unit upper triangular matrix and D diagonal matrix. This is achieved applying the square-root free Cholesky decomposition. Then, U and D are updated and propagated, not P. This method ensures that P will be a symmetric positive-define matrix over the entire KF operation, which is a key condition for KF numerical stability.

The UD filtering, as other alternative KF formulations, is known for being a computationally intensive algorithm. The computational requirements of the UD filtering are proportional to the size of the system state vector $\delta\hat{\boldsymbol{x}}$, which determines the dimensions of U and D. By example, the EKF of an integrated navigation system should have a minimum of 21 states:

$$\delta\hat{\boldsymbol{x}} = \left[\delta\hat{\boldsymbol{e}}^T,\, \delta\hat{\boldsymbol{v}}^{nT},\, \delta\hat{\boldsymbol{p}}^{nT},\, \hat{\boldsymbol{b}}_g^T,\, \delta\hat{\boldsymbol{b}}_g^T,\, \hat{\boldsymbol{b}}_f^T,\, \delta\hat{\boldsymbol{b}}_f^T\right]^T ,$$

3 for attitude estimation ($\delta\hat{\boldsymbol{e}}$), 3 for velocity estimation ($\delta\hat{\boldsymbol{v}}^n$), 3 for position estimation ($\delta\hat{\boldsymbol{p}}^n$), 6 for static and dynamic gyroscopes biases estimation ($\hat{\boldsymbol{b}}_g$ and $\delta\hat{\boldsymbol{b}}_g$), and 6 for static and dynamic accelerometers biases estimation ($\hat{\boldsymbol{b}}_f$ and $\delta\hat{\boldsymbol{b}}_f$). Then, for this particular case matrices U and D are of order 21. According to [1, Table 6.15, p. 270 and Table 6.16, p. 274], the conventional UD filtering will require 27,758 flops to be executed on a DSP microproccesor. It is worth mentioning that this is a very conservative estimate that does not take into account another fundamental operations that must be executed, as memory data transfers and iteration counters updates.

In addition, since inertial sensors typically deliver new measurements at 100 Hz, the UD filtering must be completely updated at least every 10 ms. In a multitasking scheme, as a guidance, navigation, and control system (also known as autopilot), the execution of the UD filtering is one of several functions that the microprocessor has to process, e.g., a real-time operating system. Thus, the UD filtering processing may be overwhelming for an embedded microprocessor usually running at 100~500 MHz. An FPGA (Field Programmable Gate Arrays) is an advantageous electronic platform for solving this problem. It allows to execute computing-demanding algorithms in customized reconfigurable logic, causing the freeing of a soft-core or hard-core microprocessor for managing other less-demanding tasks.

In the revised literature, few papers tackle a fully KF hardware implementation with floating-point precision. Most works are focused on fixe-point arithmetics. Some relevant papers on this area are [5], [6], and [7], among others. After an exhaustive research in the previous literature, only three works

are found that fully implement a KF in hardware, particularly in FPGA, with single-precision floating-point format. In [8] an EKF based on traditional KF equations for localization and mapping of autonomous mobile robots is presented. Paper from [9] shows a sequential EKF for robot localization in Cartesian coordinates. Lastly, [10] exhibits an FPGA implementation of a regular KF for detecting discontinuities in TIG welding processes. None of these works takes into account some numerically-stable version of the KF. It is the authors' opinion that this paper is the first reported work about a fully UD filtering implementation in hardware with single-precision floating-point arithmetics.

In this work an UD filtering algorithm optimized for execution in hardware is presented. The conventional UD filtering is optimized for software execution. It operates recursively on matrices U and D in an element-by-element fashion. This approach is not convenient for hardware realization since it has difficulties to exploit parallelism and pipelining, two key advantages of execution in hardware. In contrast, presented new UD filtering algorithm operates on arrays, which is a more convenient way for hardware design.

The rest of this paper is organized as follows. Section II shows the UD filtering algorithm targeted towards hardware synthesis. Section III provides some aspects of the development of the UD filtering co-processor. Section IV compares our design with the synthesis of the sequential UD filtering. Finally, in section V concluding remarks and future work are commented.

II. UD FILTERING ALGORITHM FOR HARDWARE

In this section, UD filtering algorithm for hardware is provided. As with the KF, the UD filtering is divided in two parts: the UD measurement update [2] and the UD time update [3]. A concise description of the theory behind the UD filtering algorithm is given in [4, Sec. 6.4].

The sequential version of the UD filtering can be found in [11, Appendices V.A and VI.A], in FORTRAN language. This version is targeted for execution on a microprocessor. It operates consecutively only on nonzero entries of matrices U and D, one by one. This strategy saves computation time and accelerates the program. On the other hand, this sequential version also presents loops that do some intermediate calculations with strong dependencies among them, again, one value at a time. From the point of view of hardware designing, this algorithm will lead to loops that contain heterogeneous operations with inefficient pipelines. A better approach for hardware realization is to try to obtain the intermediate values at once, saving them on arrays, and then using them to calculate U and D. This methodology will lead to loops with homogeneous parallel structures and efficient pipelines.

Algorithms that implement UD filtering for hardware are presented in MATLAB language, for a straight comparison with the sequential version of the UD filtering exhibited in [1, Table 6.15, p. 270 and Table 6.16, p. 274].

A. UD measurement update

Fig. 1 shows the function `bierman_hw`, responsible for executing the UD measurement update targeted for hardware. `bierman_hw` receives as arguments `r`, the *i*th diagonal entry of R (diagonal, measurement covariance matrix); `h`, the *i*th row of H (measurement matrix); `z`, the *i*th element of the measurement vector, y, and the *a priori* state vector `xpr`. It returns *a posteriori* state vector `xpo`.

The *a priori* arrays, `Upr` and `dpr`, and the *a posteriori* arrays, `Upo` and `dpo`, are considered as internal states of the algorithm, without visibility outside the UD filtering.

```
1  function [xpo] = bierman_hw(z, r, h, xpr)
2  %   IN:
3  %       z, scalar       xpr, vector nx1
4  %       r, scalar       Upr, matrix nxn (global)
5  %       h, vector 1xn   dpr, vector nx1 (global)
6  %
7  %   OUT:
8  %       xpo, vector nx1
9  %       Upo, matrix nxn (global)
10 %       dpo, vector nx1 (global)
11
12 global Upr dpr Upo dpo
13
14 Ut  = single(zeros(n));
15 P   = single(zeros(n));
16 dpo = single(zeros(n,1));
17 U   = single(eye(n));
18 ut  = single(zeros(1,n));
19
20 for i = 1:n
21     Ut(i,:) =  Upr(:,i) * dpr(i);
22 end
23
24 uh = Ut * h';
25 ph = Upr * uh;
26 alpha = h * ph + r;
27 k = ph ./ alpha;
28
29 xpo = xpr + (k * z);
30
31 for j = n:-1:1
32     uha = -(uh(j) / alpha);
33     p = uha .* uh;
34     p(j) =  dpr(j) + p(j);
35
36     for k = 1:n
37         ut(k) = dpo(k) * U(j,k);
38     end
39
40     for i = j:-1:1
41         sum = p(i);
42
43         for k = 1:n
44             sum = sum - U(i,k) * ut(k);
45         end
46
47         if (i == j)
48             dpo(j) = sum;
49         else
50             U(i,j) = sum / dpo(j);
51         end
52     end
53 end
54
55 Upo = Upr * U;
56
57 end
```

Fig. 1. Hardware-oriented UD measurement update.

First, `bierman_hw` carries out the sequential KF and

updates the *a posteriori* state estimate `xpo` (lines 20 to 29). Second, it updates one column from the *a posteriori* error covariance matrix (lines 32 to 34). Finally, it applies UD factorization to get the *a posteriori* `Upo` and `dpo` (lines 36 to 55).

For loops from lines 36 and 43 could iterate from `k=(j+1)` instead of `k=1`. Although this change will lead to loops with fewer iterations, a loop with variable repetitions will produce extra hardware. Then, the loops are set to a fixed number of iterations.

The sequential KF is more numerically-stable than the classical KF equations, but has a drawback. The sequential KF is an iterative algorithm that manipulates elements from R and H by parts [4, Sec. 6.1]. As a result, `bierman_hw` must be executed l times recursively, where l is the number of measurements, in order to output correct return values (Fig. 3).

B. UD time update

Fig. 2 exposes the function `thornton_hw`, on charge of executing the UD time update. `thornton_hw` receives as arguments the discrete-time state transition matrix `A` and the discrete-time process noise covariance matrix `Qd`.

First, `thornton_hw` defines the matrices `W` and `D` (line 12 to 17). Second, it applies Gram-Schmidt orthogonalization procedure to get the *a priori* `Upr` and `V` (line 28 to 41). Lastly, the *a priori* `dpr` is calculated (line 43).

One optimization is carried out to decrease the computational load. Matrix `Qd` is not calculated at every iteration, as the sequential UD time update does. It must be provided to `thornton_hw` already updated.

As can be seen from Fig. 1 and 2, intermediate calculations are saved on arrays (e.g., `ut`, `p`, `vDv`, and `V`), whose rows or columns are obtained at one iteration. As said, this approach will produce regular hardware structures with optimal pipelines.

It is worth mentioning that `bierman_hw` and `thornton_hw` are tested in the context of a simulator for integrated navigation systems (INS) developed in MATLAB, with single-precision floating-point arithmetics. A real-world data set provided by [12] is used to stimulate the simulator. It is composed by the following sensors: Crossbow IMU400CD, Novatel GPS, navigation-grade Honeywell IMU H764G-1 and a differential GPS. Data from the last two sensors are fused to obtain a reference data set. The trajectory takes 9 minutes and contains high and low dynamics. The availability of GPS signal is 100% throughout the entire trajectory.

Table I shows RMS errors for sequential (UD-SEQ) and hardware (UD-HW) UD algorithms. As seen from the third column of Table I, differences between the two UD filtering RMS errors are negligible. Therefore, we validate the UD filtering algorithm for hardware.

Finally, function `UD_filtering`, shown in Fig. 3, implements the UD filtering. It gives the full picture of how functions `bierman_hw` and `thornton_hw` interact.

```
1  function thornton_hw(A, Qd)
2  %   IN:
3  %       Qd, matrix nxn  Upo, matrix nxn (global)
4  %       A,  matrix nxn  dpo, vector nx1 (global)
5  %
6  %   OUT:
7  %       Upr, matrix nxn (global)
8  %       dpr, vector nx1 (global)
9
10 global Upr dpr Upo dpo
11
12 [n,¬]= size(A);
13
14 Dpo = diag(dpo);
15 W  = [A*Upo eye(n)];          % W,  matrix nxm
16 Dt = [zeros(n) Qd];          % Dt, matrix nxm
17 D  = [Dpo zeros(n); Dt];     % D,  matrix mxm
18
19 [n,m]= size(W);              % m = 2*n
20
21 V   = single(zeros(n,m));    % V,   matrix nxm
22 Upr = single(eye(n));        % Upr, matrix nxn
23 Dv  = single(zeros(m,n));    % Dv,  matrix mxn
24 vDv = single(zeros(n,1));    % vDv, vector nx1
25
26 V(n,:) = W(n,:);
27
28 for i = n-1:-1:1
29     Dv(1:n,i+1)      = dpo .* V(i+1,1:n)';
30     Dv(n+1:end,i+1) = Qd   * V(i+1,n+1:end)';
31     vDv(i+1) = 1 / ( V(i+1,:) * Dv(:,i+1) );
32
33     Usum = zeros(1,m);
34
35     for j = i+1:n
36         Upr(i,j) = (W(i,:) * Dv(:,j)) * vDv(j);
37         Usum = Usum + Upr(i,j) * V(j,:);
38     end
39
40     V(i,:) = W(i,:) - Usum;
41 end
42
43 dpr =  diag(V * D * V');
44
45 end
```

Fig. 2. Hardware-oriented UD time update.

TABLE I. RMS ERRORS FOR SEQUENTIAL AND HARDWARE UD FILTERINGS.

		UD-SEQ	UD-HW	Diff.
Roll angle	deg	0.198	0.174	0.024
Pitch angle	deg	0.233	0.204	0.029
Yaw angle	deg	3.601	3.439	0.162
Velocity north	m/s	0.139	0.120	0.019
Velocity east	m/s	0.190	0.165	0.025
Velocity down	m/s	0.198	0.172	0.026
Latitude	m	0.336	0.347	-0.011
Longitude	m	0.409	0.403	0.006
Altitude	m	0.214	0.213	0.001

For completeness, Fig. 4 shows how matrices `A` and `Qd` are formed, where `F` is the continuous-state transition matrix, `Q` is the continuous process noise covariance matrix, `G` is coupling matrix between random process noise and the states, and `dt` is the time step.

```
function [xpr] = UD_filtering(xpr, y, A, H, ...
    Qd, R)
%   IN:
%       xpr, vector nx1   H,  matrix lxn
%       y,   vector lx1   Qd, matrix nxn
%       A,   matrix nxn   R,  matrix lxl
%
%   OUT:
%       xpr, vector nx1

global Upr dpr Upo dpo

[l,¬] = size(R);

for i = 1:l

    h = H(i,:);
    r  = R(i,i);
    z  = y(i);

    xpo = bierman_hw(z, r, h, xpr);
    xpr=xpo; Upr=Upo; dpr=dpo;
end

thornton_hw(A, Qd);

xpr = A * xpo;

end
```

Fig. 3. UD filtering.

```
% F, matrix nxn   G,  matrix nxp
% Q, matrix pxp   dt, scalar
[n,¬] = size(F);
I  = eye(n);
A  = I + (F * dt);
Qd = (G * Q * G') * dt;
```

Fig. 4. Update of matrices A and Qd.

III. UD FILTERING HARDWARE DESIGN

The primary idea behind the development of the UD filtering for hardware is to produce a co-processor in charge of executing the UD filtering algorithm. This co-processor (from here named as UD-HW co-processor) is part of a system-on-chip (SoC) system, whose main processing unit is a microprocessor. Microprocessor and co-processor are interfaced through a common data bus. Thus, the microprocessor will off-load the UD filtering update to the UD-HW co-processor. Such an architecture provides an efficient hardware/software exploration.

The UD-HW co-processor is developed by using Vivado High-Level Synthesis (HLS) [13]. This is an IDE (integrated development environment) from FPGA vendor Xilinx. It offers the capability of designing a digital system using C/C++ languages. The IDE also provides tools to create a software test bench in order to prove the functional correctness of an algorithm targeted towards FPGA execution.

The advancement of using C/C++ languages, which abstract the details of the computing platform, is that it allows the designer to test different approaches to implement a particular algorithm. If it is detected that a certain C/C++ algorithm, initially designed to run on a microprocessor, has a high computational complexity, the designer can quickly move this algorithm to hardware. Thus, the microprocessor is released to execute other assignments. Moreover, it is avoided to recode the algorithm in a register-transfer level (RTL) description language, e.g., VHDL or Verilog, often not a simple task. However, sometimes hardware synthesis is not straightforward. The designer must structure the algorithms in a way that leads to efficient parallelization and pipelining.

Vivado HLS extracts the best possible circuit-level implementation of the software application applying three techniques: 1) scheduling, which is the process of identifying the data and control dependencies among different operations to determine when each one will execute; 2) pipelining, which divides a particular circuit into a chain of independent stages; and 3) dataflow, which extracts program's level of parallelism. Vivado HLS has limitation respect to the C/C++ functions that can synthesize. By example, dynamic memory allocation is not supported because it requires that used memory to be completely known at compile time.

Functions from Fig. 1 and 2 are recoded in C. To validate these programs, a test bench program is developed. The test is carried out using testing vectors from the mentioned INS simulator for one system iteration. Once debugged, each program is compiled using MATLAB built-in tool MEX. MEX compiles C/C++ code into a dynamically linked MATLAB executable. Then, C programs are tested under the INS simulator for the entire operation of the system. It is found that RMS errors are close to those from Table I. As a result, C version of UD filtering algorithm for hardware is validated.

Vivado HLS provides ad-hoc directives or pragmas to optimize the design from the source code. Thus, the designer can adjust the digital system and choose the solution that best meets the application restrictions on area and latency. Typically, optimization involves a trade-off between area (hardware resources) and performance. Pragmas are also used to bind arrays and arithmetic operators to particular hardware elements to balance the FPGA area usage.

Ad-hoc pragmas provided by the IDE are intensively used in the design to decrease the UD-HW co-processor latency. As example, Fig. 5 shows the function for matrix-vector multiplication, which is extensively used in `bierman_hw` (Fig. 1). It exposes how `PIPELINE`, `UNROLL` and `ARRAY_PARTITION` pragmas are used.

Analyzing the function from Fig. 5, one can notice that is not the trivial matrix-vector multiplication. Matrix `U_i` is accessed by columns, not rows, and partial MAC (multiply-and-accumulate) results are saved in vector `v_o`. This way, elements from vector `v_i` are read only once, which reduces the memory access and produce a faster pipeline. This is a simple example that exposes some considerations must be taken into account when designing hardware using C language.

The `PIPELINE` pragma instructs the compiler to synthesize the inner for loop splitted into a chain of independent stages, instead of a standard combinational circuit. Thus, the loop can accept a new input every 1 clock cycle, which improves the circuit latency. Usually, FPGA have hard-coded MAC circuits with intermediate registers in the data path

```
1  #define  N  21
2
3  void matrix_x_vector(float U_i[N][N], float ...
       v_i[N], float v_o[N])
4  {
5      int i, j;
6      float prod, vt;
7
8  #pragma HLS ARRAY_PARTITION variable=U_i
9          cyclic factor=7 dim=1
10 #pragma HLS ARRAY_PARTITION variable=v_i
11         cyclic factor=7 dim=1
12 #pragma HLS ARRAY_PARTITION variable=v_o
13         cyclic factor=7 dim=1
14
15     for (i=0; i<N; i++)
16         {
17         for (j=0; j<N; j++)
18             {
19     #pragma HLS PIPELINE
20     #pragma HLS UNROLL factor=7
21
22                 if (j == 0) vt = v_i[i];
23                 prod = vt * U_i[j][i];
24
25                 if (i == 0) v_o[j] = prod;
26                 else v_o[j] += prod;
27         }
28     }
29 }
```

Fig. 5. C-code function to multiply a matrix by a vector.

that allow to pipeline arithmetic operators to get a better throughput.

Additionally, the UNROLL pragma lets divide a for loop in several parallel circuits, and thus several operations will be executed at the same time using less clock cycles, at a cost of a larger area. A loop can be partially or completely unroll, according to area restrictions. Besides, it is convenient that partial unrolling to be multiple of the amount of iterations, in favor of getting more homogeneous hardware structures. Thus, since the UD-HW co-processor is addressed to process matrices of order 21, loops from bierman_hw iterates $N = 21$ times and are partially unrolled by a factor of 3 or 7. In thornton_hw, some loops iterates $M = 42$ times and can be partially unrolled by a factor of 2, 3, 6 or 7. Often, a complete unroll for matrices of these orders can be impractical. In fact, complete unrolls for all UD-HW co-processor's loops will lead to hardware starvation for a particular FPGA.

Due to unroll brings an area increase, computing-demanding parts of the algorithm should be exclusively unrolled by higher factors. The inner loop from thornton_hw (Fig. 2, line 35) has a strong influence on latency reduction. This loop contains intensive calculations and is the most executed, exactly $\sum i_{i=1}^{N-1}$ times. Then, too much attention is put on reducing the execution latency of this loop. Therefore, it is partially unrolled by the maximum factor, 7, and vectors that hold intermediate values are mapped in distributed RAM, which is accessed faster than block RAM.

FPGA memories only have a limited amount of read and write ports that may limits the throughput. This bottleneck can be avoided by splitting up the original memory into multiple smaller memories, effectively increasing the number of ports. The ARRAY_PARTITION pragma sets how many blocks an array is divided into. The dim option specifies which array dimension is partitioned. As example, matrix U_i (Fig. 5) is splitted into 7 arrays on the first dimension, accordingly, each one will have an order of 3x21. The cyclic option sets that elements from the original array are stored into partitioned arrays in an interleaved fashion. Since U_i entries are distributed in a cyclic way, it is possible to read two adjoining entries at the same time.

Tipically, UNROLL and ARRAY_PARTITION pragmas are used together to achieve the required performance for a particular loop.

IV. Results analysis

In this section, synthesis results from both UD filterings, sequential and hardware versions, are exposed.

Since no previous works in the literature were found against which to compare the UD-HW co-processor with, the sequential UD filtering is also implemented as a hardware co-processor, as a proposed benchmark. It is recoded in C from MATLAB [1, Table 6.15, p. 270 and Table 6.16, p. 274]. C functions for sequential UD filtering are validated applying the same method used for C programs for hardware implementation (Sec. II). Obtained RMS errors are similar to those of Table I. Consequently, C version of sequential UD filtering is validated.

No pragmas are used for optimization. The filter is synthesized "as is". Nevertheless, the resulting execution profile may run faster in hardware than on a microprocessor. This is due to data are fetched from memory in a concurrent way when no data contention occurs. Thus, some data are read from and/or written to memory in parallel, which accelerates the sequential UD filtering execution.

Both co-processors are targeted towards execution on a Xilinx Zynq Z-7020 [14] at 100 MHz. This programmable mid-range SoC (system-on-chip) provides in a single chip an ARM dual-core Cortex-A9 microprocessor and FPGA recourses, both connected through an AMBA bus. It combines the software programmability of a microprocessor with the hardware reconfigurability of an FPGA.

Table II shows the synthesis results for both filters, sequential (UD-SEQ) and hardware-oriented (UD-HW, Fig. 3) versions, for a 21-states UD filtering. Third column shows the hardware utilization relationship between the two co-processors. All floating-point cores are synthesized with IEEE-754 standard compliance. Typical hardware resources for FPGA are exposed. DSP slice is a hard-coded MAC circuit. BRAM is a hard-coded RAM memory block. Finally, FF stands for flip-flops and LUT for look-up tables.

The UD-HW co-processor provides an improvement by 10.64x in performance, which is a consequence of the strong reduction on the maximum latency. As expected, this achievement is at the cost of more area usage. As shown in the third column from Table II, when compared with the UD-SEQ, the UD-HW filter's hardware utilization is not excesive, except for BRAM and FF. This high numbers of BRAM is caused by memory partitions, which are essential to achieve the

accomplished performance. However, on average, the UD-HW co-processor only consumes ~5 times more hardware resources than the UD-SEQ co-processor and gets a ~10x performance boost. Hence, the UD-HW co-processor has a performance-vs.-area improvement by ~2x, when compared with the UD-SEQ co-processor.

TABLE II. SYNTHESIS RESULTS FOR SEQUENTIAL AND HARDWARE-ORIENTED UD FILTERINGS.

	UD-SEQ	UD-HW	HW utilization relationship
Maximum latency	686,738	64,543	–
Performance	1x	**10.64x**	–
DSP slices	40	116	2.90
BRAM	15	84	5.60
FF	5,060	25,060	4.95
LUT	7,996	31,037	3.88
		Average	**5.31**

Table III shows the hardware resources utilization for the targeted FPGA. It can be seen that the UD-HW consumes almost half of DSP slices and LUT. Nevertheless, enough area is left to synthetize another co-processor, similar to the UD-HW. This moderate use of resources states that the UD-HW co-processor is suitable for synthesis on mid-range FPGA.

TABLE III. ZYNQ Z-7020 TOTAL RESOURCES.

	Z-7020 total resources	UD-HW utilization
DSP slices	220	53%
BRAM	280	30%
FF	106,400	24%
LUT	53,200	58%

V. CONCLUSIONS AND FUTURE WORK

In the present work, a new algorithm for UD filtering that exploits hardware parallelism and pipelining is presented. It is the authors' believe that this is the first report about a hardware realization of the UD filtering. The algorithm is shown as MATLAB source code, and is validated employing a test bench and a simulator for integrated navigation systems.

The UD-HW co-processor is designed using Vivado HLS, an IDE that synthesizes digital systems written in C/C++. The co-processor is optimized through directives that the IDE provides.

Since no previous works are found in the literature that report a UD filtering implementation for hardware, the sequential UD filtering is implemented in hardware for benchmark purpose. It is observed that the UD-HW co-processor provides a speed-up of 10.64x and a performance-vs.-area improvement by ~2x. Moreover, it has a relative low area utilization for a 21-states UD filtering, which makes it suitable for synthesis targeted towards mid-range FPGA.

As future work, we find some interesting research topics that are worth investigating further. The UD-HW co-processor will be implemented in a Xilinx Zynq Z-7020. On the other hand, the sequential UD filtering algorithm will be compiled to run on Zynq's ARM microprocessor. Several aspects from both UD filtering solutions are relevant to be investigated, as the trade-off among area, energy consumption and time execution.

ACKNOWLEDGMENT

The authors would like to thank to Dr. Charles K. Toth (from Ohio State University), Dr. Allison Kealy, and M.Sc. Azmir H. Rabiain (both from The University of Melbourne) for generously sharing IMU and GPS data sets, and in particular, for Rabiain's unselfish help.

REFERENCES

[1] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice Using MATLAB*, 3rd ed. John Wiley & Sons, Inc., 2008.

[2] G. J. Bierman, "Measurement updating using the U-D factorization," *Automatica*, vol. 12, no. 4, pp. 375–382, 1976.

[3] C. L. Thornton and G. J. Bierman, "Gram-Schmidt algorithms for covariance propagation," *International Journal of Control*, vol. 25, no. 2, pp. 243–260, 1977.

[4] D. Simon, *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley & Sons, Inc., 2006.

[5] C. Lee and Z. Salcic, "High-performance FPGA-based implementation of Kalman filter," *Microprocessors and Microsystems*, vol. 21, no. 4, pp. 257–265, 1997.

[6] Y. Liu, C. Bouganis, and P. Y. K. Cheung, "Efficient mapping of a Kalman filter into an FPGA using Taylor expansion," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 345–350.

[7] L. Idkhajine, E. Monmasson, and A. Maalouf, "FPGA-based sensorless controller for synchronous machine using an extended Kalman filter," in *Power Electronics and Applications, 2009. EPE '09. 13th European Conference on*, Sept 2009, pp. 1–10.

[8] V. Bonato, E. Marques, and G. Constantinides, "A floating-point extended Kalman filter implementation for autonomous mobile robots," in *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, Aug 2007, pp. 576–579.

[9] S. Cruz, D. Muñoz, M. Conde, C. Llanos, and G. Borges, "FPGA implementation of a sequential extended Kalman filter algorithm applied to mobile robotics localization problem," in *Circuits and Systems (LASCAS), 2013 IEEE Fourth Latin American Symposium on*, Feb 2013, pp. 1–4.

[10] R. Hurtado, S. C. A. Alfaro, and C. Llanos, "A methodology for "on-line" monitoring system in a welding process using FPGAs," in *Industrial Technology (ICIT), 2010 IEEE International Conference on*, March 2010, pp. 162–167.

[11] G. J. Bierman, *Factorization Methods for Discrete Sequential Estimation*, R. Bellman, Ed. Academic Press, 1977.

[12] C. Toth, D. Brzezinska, N. Politi, and A. Kealy, "Reference data set for performance evaluation of MEMS-based integrated navigation solutions," in *FIG Working Week 2011*, Marrakech, Morocco, May 2011.

[13] *Introduction to FPGA Design with Vivado High-Level Synthesis, UG998 v1.0*, Xilinx, Inc., July 2 2013.

[14] Xilinx, Inc., "Zynq-7000 all programmable SoC," June 2014. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/