Optimized Parallel Implementation of Extended Kalman Filter Using FPGA

Article *in* Journal of Circuits, Systems and Computers · June 2017

DOI: 10.1142/S0218126618500093

3 authors, including:

Amin Jarrah
University of Toledo
30 PUBLICATIONS **47** CITATIONS

SEE PROFILE

Abdel-Karim Al-Tamimi
Yarmouk University
49 PUBLICATIONS **934** CITATIONS

SEE PROFILE

Journal of Circuits, Systems, and Computers
Vol. 27, No. 1 (2018) 1850009 (22 pages)

DOI: 10.1142/S0218126618500093

Optimized Parallel Implementation of Extended Kalman Filter Using FPGA*

Amin Jarrah†, Abdel-Karim Al-Tamimi‡ and Tala Albashir

Department of Computer Engineering,
Yarmouk University,
Irbid 21163, Jordan
†*amin.jarrah@yu.edu.jo*
‡*altamimi@yu.edu.jo*

Received 7 March 2016
Accepted 24 April 2017
Published 1 June 2017

There are enormous numbers of applications that require the use of tracking algorithms to predict the future states of a system according to its previous accumulated states. Thus, many efficient techniques are widely adopted to estimate the future states of a system at every point in time to get the desired performance levels. Kalman filter is a popular and an efficient method for online estimations for linear measurements. Extended Kalman Filter (EKF), on the other hand, is more suited for nonlinear measurements. However, EKF algorithm is well known to be computationally intensive, and may not achieve the strict requirements of real time applications. This issue has motivated researchers to consider the use of parallel processing platforms such as Field Programmable Gate Arrays (FPGAs) and Graphic Processing Units (GPUs) to meet the real time requirements. This paper provides an optimized parallel architecture for EKF using FPGA. Our approach exploits many optimization and parallel techniques such as pipelining, loop unrolling, dataflow, and inlining; and utilizes the inherently parallel architecture nature of FPGAs to accelerate the estimation process. Our experimental analyses show that our optimized implementation of EKF can achieve better results when compared to other implementations using GPU and multicore platforms. Moreover, higher performance levels can be achieved when operating on larger data sizes. This is due to our proposed optimization techniques that we have applied, and the exploited inherent parallelism among EKF operations.

Keywords: Tracking algorithms; time series analysis; data modeling; data prediction; extended Kalman filter; field-programmable gate array (FPGA); high level synthesis tools (HLS Tools); parallel architecture; optimization techniques.

1. Introduction

Object tracking algorithms are designed to accept real-time input measurements from object detection systems like radar, sonar and video devices, and perform

*This paper was recommended by Regional Editor Tongquan Wei.
†Corresponding author.

necessary operations to locate and track a moving object (or multiple objects) over time.[1–3] It has the ability to filter noisy signals, to generate non-observable states, and to predict future states of the system. Figure 1 shows an example for a tracking algorithm.[4]

Kalman filter[4–6] is used in many tracking objects applications[7–12] such as (missiles, faces, heads, and hands), fitting Bezier patches to noisy point data in geometry, and in time series predictions that are vital for economic data forecasting and Internet traffic.[7,8] Kalman filter is also used in many other applications in computer vision such as depth measurements stabilization, feature tracking, and cluster tracking,[4] and especially in linear systems to produce an optimal estimation. Kalman filter is a recursive estimator since the estimation of the current state depends on the estimation of the previous time steps. Kalman filter[5] can be represented in Eq. (1):

$$x_k = F_k x_{k-1} + B_k u_k + w_k \,, \tag{1}$$

where x_k is the true state at time k, F_k is the system dynamics matrix which is applied on the previous state x_{k-1}, B_k is the control input model, u_k is a known vector (control vector), w_k is the process's noise and it is a multivariate normal distribution function with a zero mean and a covariance Q_k, which we can describe as $w_k \sim \mathcal{N}(0, Q_k)$.[6]

Kalman filter[5] requires linear measurements for updating the predicted estimations based on Eq. (2):

$$z_k = H_k x_k + v_k \,, \tag{2}$$

where z_k is the measurement vector, H_k is the measurement matrix (observations), v_k is the white noise represented by a multivariate normal distribution with a zero mean and a covariance Q_k, which we can describe as $v_k \sim \mathcal{N}(0, R_k)$.[6]

However, Kalman filter is known to be optimized for linear applications, but fails when working with non-linear applications.[6] Therefore, Extended Kalman Filter (EKF)[13] was proposed and adopted to be used in systems that exhibit non-linear

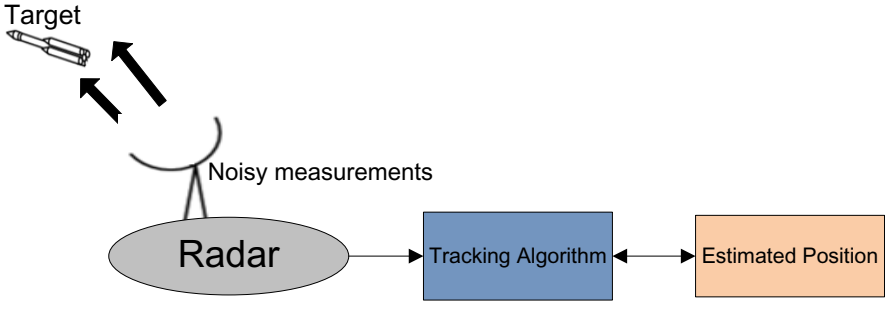


Fig. 1. Position estimation in tracking and navigation applications.

behaviors.[14] EKF model can be represented as in Eq. (3).

$$\begin{aligned} x_k &= f(x_{k-1}, u_{k-1}) \\ z_k &= h(x_k, v_k)\,, \end{aligned} \tag{3}$$

where x_k is the state or estimated value at time step k, and z_k is the measurement or actual values at time step k.

Prediction and estimation stages are the two main stages in Kalman filter that can be represented by the following equations:

Prediction stage:

$$x^{\wedge}_{k/(k-1)} = f(x^{\wedge}_{k-1})\,, \tag{4}$$

$$P_{k/(k-1)} = F_{k-1}P_{k-1}F^T_{k-1} + B_{k-1}QB^T_{k-1}\,, \tag{5}$$

where $P_{k/(k-1)}$ is the predicted covariance estimation.

Estimation stage:

$$x^{\wedge}_k = x^{\wedge}_{k/(k-1)} + K_k(y_k - h(x^{\wedge}_{k/(k-1)})\,, \tag{6}$$

$$k_k = P_{k/(k-1)}H^T_k(H_kP_{k/(k-1)}H^T_k + C_kRC^T_k)^{-1}\,, \tag{7}$$

$$P_k = (1 - k_kH_k)P_{k/(k-1)}\,, \tag{8}$$

where k_k is the near-optimal Kalman gain.

In EKF, first-order partial derivations for the square matrix (Jacobian matrix) should be determined in order to derive the values of the prediction and estimation stages variables[6] as shown in Eqs. (9)–(12).

$$F_{k-1} = \frac{\partial f(x_{k-1})}{\partial x_{k-1}}\bigg|x_{k-1} = x^{\wedge}_{k-1}\,, \tag{9}$$

$$H_k = \frac{\partial h(x)}{\partial x}\bigg|x = x^{\wedge}_{k/(k-1)}\,, \tag{10}$$

$$B_{k-1} = \frac{\partial f(x^{\wedge}_{k-1})}{\partial x_{k-1}}\bigg|u_{k-1} = 0\,, \tag{11}$$

$$C_k = \frac{\partial h(x^{\wedge}_{k/(k-1)})}{\partial v_k}\bigg|v_k = 0\,. \tag{12}$$

However, EKF[13,14] is a complex algorithm, and may not meet the strict requirements of real time applications. Therefore, researchers have been working towards adopting several parallelism techniques to implement it by exploiting all the available optimization and parallelization techniques that can reflect positively on the performance of the algorithm. Therefore, in order to overcome the complexity issues normally associated with EKF, EKF needs to be approached in a manner to facilitate exploiting it using different parallel processing platforms to achieve a higher

performance, such as Field Programmable Gate Arrays (FPGAs).[15,16] Different optimization and parallelization techniques are supported on FPGA platforms. Parallelization techniques help in dividing large problems into smaller tasks that can be solved in parallel instead of executing only one small task at a time (i.e., serial processing).[7]

FPGA is one of the most efficient parallel platforms that can be used for parallelizing complex algorithms.[15,16] FPGA was adopted in this work since it provides numerous hardware resources that can be re-configured based on the required computational process, provides the ability to test the design without a manufacturing process, is relatively a low cost solution, and provides a high level of flexibility by providing adjustable hardware implementations of software codes.[15,16] FPGA is a semiconductor device that contains a matrix of reconfigurable logical blocks that can be connected together by programmable interconnections. The Hard Description Language (HDL), such as VHDL and Verilog languages, can be used to configure the logical blocks to meet the tasks of a system's custom specifications.[18]

However, HDL languages are widely considered as complex, and require hardware designers to understand the underlying architecture to reach the desired optimization levels, especially when the required algorithm is complex.[19,20] Therefore, Xilinx provides a design suite called Vivado High-Level Synthesis Tools (HLST)[21] that helps designers achieve high performance levels and meet real time constraints without the need to program their design in HDL languages. HLST can be used to synthesize and analyze HDL designs as shown in Fig. 2. It allows designers to obtain

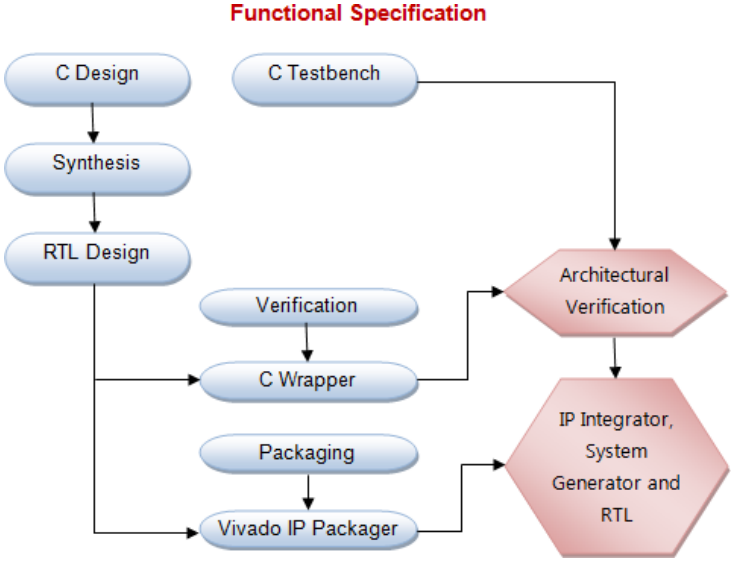


Fig. 2. Xilinx Vivado High Level Synthesis Tools (HLST) design flow.

the optimized bit-stream of the algorithm by applying different optimization techniques that are suitable for design requirements and for obtaining the best performance levels without having a previous knowledge in HDL. It supports different optimization techniques such as loop pipelining, loop unrolling, dataflow, and inlining to improve the latency and the throughput of the system.[21]

The remainder of this paper is organized as follows: in Sec. 2, the analysis and optimization techniques for EKF implementation are presented. Section 3 provides EKF algorithm simulations and synthesis results. Finally, Sec. 4 concludes the paper with our findings.

2. Analysis and Optimization Techniques

In order to exploit all EKF optimization possibilities, it must be analyzed for any small and inherent parallelism possibility in its operations. The EKF can be divided into several steps as shown in Fig. 3, which shows the relation between its operations. The pseudocode of the EKF algorithm is also presented in Fig. 4, which represents all the required mathematical operations that are needed to be optimized to achieve the desired goal.

Due to the limited size of this paper, we urge our readers to refer to each algorithm's corresponding reference for more information about its details and characteristics. Different parallelization and optimization techniques can be adopted and

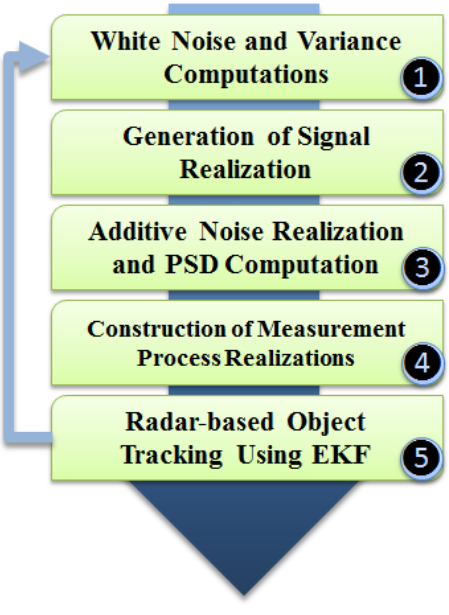


Fig. 3. Main EKF steps.

Algorithm 1 Extended Kalman Filter

Input: $s_{t-1}, P_{t-1}, u_t, Z_{t-1}$
Output: $\hat{s_t}$, $\hat{P_t}$

1: $s_t \leftarrow f(s_{t-1}, u_t)$
2: $P_t \leftarrow F_t P_{t-1} F_t^T + Q$
3: $K \leftarrow P_t H^T (H P_t H^T + C)^{-1}$
4: $\hat{Z} \leftarrow (z_t - h(s_t))$
5: $\hat{s_t} \leftarrow s_t + K\hat{z}$
6: $\hat{P_t} \leftarrow (1 - k_t H_t) P_t$
7: **return** $\hat{s_t}$, $\hat{P_t}$

Fig. 4. EKF algorithm pseudocode.

applied among EKF internal operations, where each step of EKF can be optimized efficiently to improve both the throughput and the latency as shown in the following sections.

2.1. *Initialization and white noise generation stage*

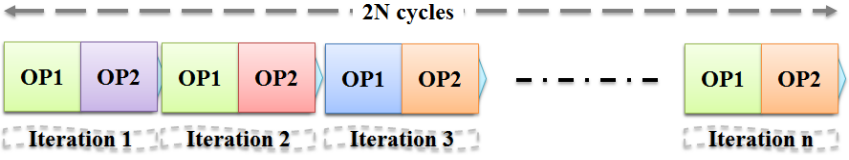
The pseudocode of the initialization and the white noise generation steps are shown in Fig. 5.[6] As we can note from Fig. 5, there is no dependency between the operations and thus we can apply different parallelization techniques such as loop pipelining and unrolling techniques to reduce the latency as shown in Figs. 6 and 7. Figure 6 shows how loop pipelining mechanism helps reduce the number of the execution cycles, which will positively reflect on the performance level.[21] Figure 7 shows the loop unrolling mechanism that will be applied to additionally enhance the performance of EKF.

Algorithm 2 White Noise and Variance Computations

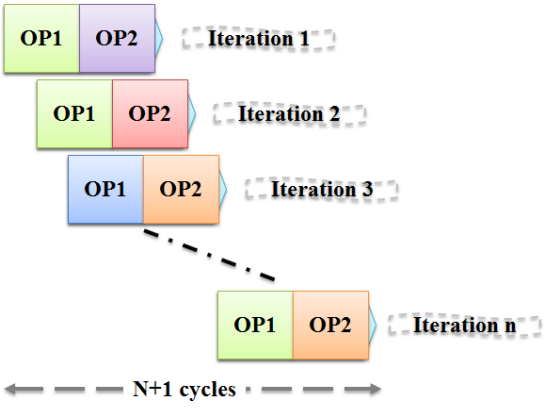
Input: sea,sea2,sew,sew2
Output: ead,edw,sa2,sw2

1: $eda \leftarrow sea \times Random(1, length\ of\ realization + 500)$
2: $edw \leftarrow sew \times Random(1, length\ of\ realization + 500)$
3: $sa2 \leftarrow sea2/(1 - 0.9^2)$
4: $sw2 \leftarrow sew2/(1 - 0.9^2)$
5: **return** $sa2, sw2, eda, edw$

Fig. 5. White noise and variance computations pseudocode.

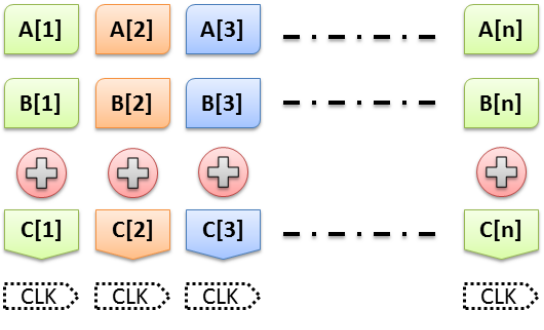


(a) Without loop pipelining

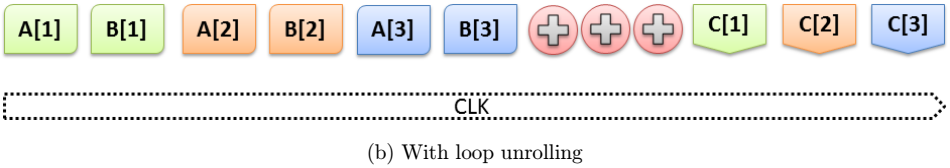


(b) With loop pipelining

Fig. 6. Loop pipelining effect on the number of required clock cycles.



(a) Without loop unrolling



(b) With loop unrolling

Fig. 7. Loop unrolling technique effect on the number of required clock cycles.

Algorithm 3 Generation of Signal Realization

1: **for** t=2, *length of realization* + 500 **do**
2: $da(t) \leftarrow 0.9 \times da(t-1) + eda(t)$
3: $dw(t) \leftarrow 0.95 \times dw(t-1) + edw(t)$
4: $w(t) \leftarrow w0 + dw(t)$
5: $theta(t) \leftarrow theta(t-1) + w(t)$
6: **end for**
7: $s \leftarrow a \times sin(theta)$
8: $s \leftarrow s(501 : length\ of\ realization + 500)$

Fig. 8. Generation of signal realizations pseudocode.

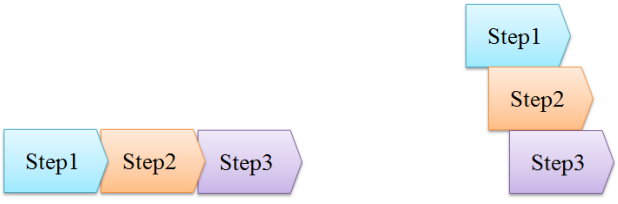
2.2. *Generation of signal realization stage*

Stage two requires the generation of signal realizations, which is presented in the pseudocode shown in Fig. 8.

As can be noted, the pseudocode statements shown in Fig. 8 have high inter-dependencies, where each statement depends on the previous statement(s). For example, the value of $a(t)$ between lines 1 and 2 depends on the computation of the $da(t)$ value. If we examine all algorithm lines, we can find dependencies between the operations in all its iterations. This prevents the operations in this step to be executed in parallel and limiting any possible performance improvements. To overcome this constraint, data flow technique can be applied between the algorithm's operations to improve the performance as shown in Fig. 9. This will help in forwarding the output data of each step to the next one without the need to wait for the previous operations to be performed completely.

2.3. *Additive noise realization and PSD compuation stage*

Stage three requires adding both noise realization and Power Spectral Density (PSD) computations. This step is divided into two parts, which are shown in Figs. 10



(a) Without Applying Dataflow Method (b) With Applying Dataflow Method

Fig. 9. Overview of dataflow method main advantage.

Algorithm 4 Additive Noise Realization

Initialize sn=0.5

1: $n \leftarrow sn \times Random(1, length\ of\ realization + 500)$
2: $sn2 \leftarrow sn^2$
3: $nar(1) \leftarrow$n(1)
4: $nar(2) \leftarrow$n(2)
5: $a1 \leftarrow -1.6 * cos(\frac{Pi}{5})$
6: $a2 \leftarrow 0.64$

Fig. 10. Additive noise realization pseudocode.

and 11. Figure 10 depicts the process of generating the additive noise, while Fig. 11 shows the process of computing PSD.

Figure 10 does not show any dependency between the process operations. This helps us in providing the ability of asserting the desirable values in parallel. This part can be fully parallelized using both pipelining and unrolling optimization techniques to decrease the latency to the lowest limit. However, there is a clear dependency between the operations of PSD computations as shown in Fig. 11. The first two lines require the evaluation of both ***fvec*** and the first 1024 elements from the ***ftt*** function, which can be performed concurrently. However, ***ARspec*** value as shown in line 4 depends on the ***den*** value from line 3, and ***ARspecdB*** value in line 5 depends on ***ARspec*** value from line 4. Moreover, the *for* loop in lines 6 and 7 also depends on the previous values of $\boldsymbol{nar(t-1)}$ and $\boldsymbol{nar(t-2)}$. All these dependencies can be alleviated by applying the data flow technique. HLST design suite supports the implementation of the dataflow technique by automatically inserting data channels between the operations. This helps in reducing the overall system's

Algorithm 5 PSD Computation

Initialize delf=1/2048

1: vector $fvec = 0 : delf : 0.5 - delf$
2: calculate fft with operants $([1\ a1\ a2], 2048)$
3: $den \leftarrow$ the result of fft from first 1024 elements
4: $ARspec \leftarrow sn2 \times (abs(den)^{-2})$
5: $ARspecdB \leftarrow 10 \times log10(ARspec)$
6: **for** t=2, *length of realization*+500 **do**
7: $\quad nar(t) \leftarrow -a1 \times nar(t-1) - a2 \times nar(t-2) + n(t)$
8: **end for**
9: $n \leftarrow nart$ [501 to $length\ of\ relization + 500$]

Fig. 11. PSD computation pseudocode.

computation time, since it will forward the required results to the current operation as soon as it is available without waiting until all other computation processes are performed.

2.4. *Construction of measurement process relaization*

Stage four requires adding two matrices to get the z matrix shown in Fig. 4, where $(Z = S + n)$. This can be parallelized easily by applying different optimization techniques such as loop unrolling and loop pipelining.

2.5. *Radar-based object tracking using EKF stage*

Stage five is an essential part of our work in order to calculate EKF equations. Figure 12 shows the pseudocode of the EKF equations for the radar-based object tracking application.

The first two lines are for the initializations of the vectors *Hs* and *H* that do not require intensive computations. However, line 3 requires three computational processes that are matrix multiplication, matrix transpose, and matrix inversion, which are explained in detail in the following subsections. These three processes could reduce the performance of our system, and therefore efficient optimization techniques must be adopted and applied to reduce their computation times.

Algorithm 6 Radar-Based Object Tracking Algorithm Using EKF

1: **for** t=1, *length of realization*+500 **do**
2: $vector(Hs) \leftarrow [0 \;\; xm(3) \times cos(xm(2)) \;\; sin(xm(2))]$
3: $H \leftarrow [Hs, Hn]$ *where* $Hn = [1\ 0]$
4: $k \leftarrow pm \times H' \times (H \times Pm \times H' + R)^{-1}$
5: $Zm \leftarrow xm(3) \times sin(xm(2))$
6: $xhatk \leftarrow xm + k \times (z(k) - zm)$
7: $x \leftarrow [x, xhatk]$
8: $P \leftarrow (I - k \times H) \times pm$
9: $xm \leftarrow phi \times xhatk$
10: $Pm \leftarrow Phi \times P \times Phi' + Q$
11: **end for**
12: $x \leftarrow x(:, 2 : length\ of\ realization + 501)$
13: $fhat \leftarrow$ first row of $matrix\ x/(2 \times pi)$
14: $ahat \leftarrow thirdrowofmatrix\ x$
15: $thetahat \leftarrow secondrowofmatrix\ x$
16: $Shat \leftarrow ahat \times sin(thetahat)$ ▷ to reconstruct signal estimate

Fig. 12. EKF pseudocode for the radar based object-tracking algorithm.

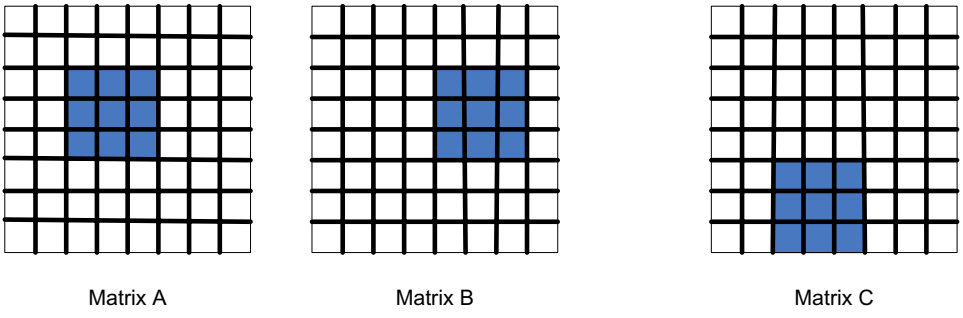


Fig. 13. Block multiplication for the matrix.

2.5.1. *Matrix multiplication process*

Matrix multiplication process's optimization is a necessary step for improving system's performance, since it is used in most of the algorithm steps. Therefore, we investigated implementing and parallelizing it using three optimization techniques: blocking technique,[22] Strassen algorithm,[23] and Cannon algorithm[24] in order to choose the best technique and show which method should be considered for our optimized FPGA design.

Blocking technique[22] is adopted and applied to boost the performance of matrix multiplication process where the matrix is divided first into many smaller blocks that can be processed in parallel. Figure 13 shows how the blocking technique can be used to perform the matrix multiplication process.

Strassen algorithm[23] is one of the most popular methods for optimizing matrix multiplications, which is based on linear algebra. Eight multiplications are required to calculate the result matrix C based on Eq. (13). However, Strassen algorithm uses only 7 multipliers instead of 8 to perform the multiplication operations as shown in Table 1.

Table 1. M terms used in Strassen's algorithm.

M terms	Description
$M1 : (A11 + A22)(B11 + B22)$	Addition result diagonal in A matrix with diagonal in B matrix.
$M2 : (A21 + A22)(B11)$	Second column in A matrix with first element in B matrix
$M3 : (A11)(B12 + B22)$	Addition of elements of Second column in B matrix with first element in A matrix.
$M4 : (A22)(B21 - B11)$	Addition of elements of Second column in B matrix with first element in A matrix.
$M5 : (A11 + A12)(B22)$	Addition of elements first row in A matrix with last element in B matrix.
$M6 : (A21 - A11)(B11 + B12)$	Subtraction result each element in first column in A matrix with summation of first row in B matrix
$M7 : (A12 + A22)(B21 + B22)$	Summation of second row in A matrix with summation second row in B matrix.

The result matrix, i.e., matrix C, is calculated using the following relations based on Table 1.

$$\begin{aligned} C11 &= M1 + M4 - M5 + M7 \\ C12 &= M3 + M5 \\ C21 &= M2 + M4 \\ C22 &= M1 - M2 - M3 + M6\,. \end{aligned} \tag{13}$$

After reserving the required memory spaces for A and B matrices, and asserting their values, we found that we have the ability to obtain the seven values of M in one clock cycle, and thus decreasing the latency of the required multiplication operations.

Cannon's algorithm has been also adopted and parallelized in our work, since it could help us achieve low latency computations.[24] In Cannon's algorithm, rows and columns of the input matrices are circulated continuously in a wrap-around fashion, such that the multiplication in a given place is done independently of other places. Hence, the algorithm is considered to be memory-efficient since each involved processing unit needs only to hold one sub-block of the input matrices. In detail, repeatedly, the elements of the first input matrix are shifted one place left and the elements of the second input matrix are shifted one place up. Accordingly, each processing unit multiplies the adjacent elements of both input matrices and accumulates the result in the same position in the output matrix. Thus, the final resultant matrix is obtained after shifting each row of the first input matrix by the number of its columns, and each column of the second input matrix by the number of its rows. To speed up the calculations of matrix multiplication process, we distribute the task of multiplying different blocks and accumulating their local summation values to multiple independent processing units. However, the circular shift operations must be accomplished sequentially after each parallel calculation is performed. Figure 14 shows the calculation steps of a block of the result matrix that are performed by the same processing unit.

The pseudocode of Cannon's algorithm is shown in Fig. 15. We initially compute $A(i, j)$ and $B(i, j)$ values. To perform this operation, elements of A and B must be aligned in such a way that the process should initially have a pair that consists of $A(i, *)$ and $B(*, j)$ whose product contributes to the final value of $C(i, j)$. Therefore, the ith row is shifted i-places to the left and the jth column is shifted j-places up. After the initial alignment, if we shift A's to the left, and B's upwards in a circular fashion, the process will always get a unique pair $A(i, k)$ and $B(k, j)$, where its product contributes to the $C(i, j)$ value. After n circular shifts, the process computes the final $C(i, j)$ value. As a result of this operation parallelism, each parallel process multiplies its two sub-matrices and accumulates the results in its product sub-matrix. These parallel processes continue until all sub-matrices have reached their starting locations at the end of the multiplication process.

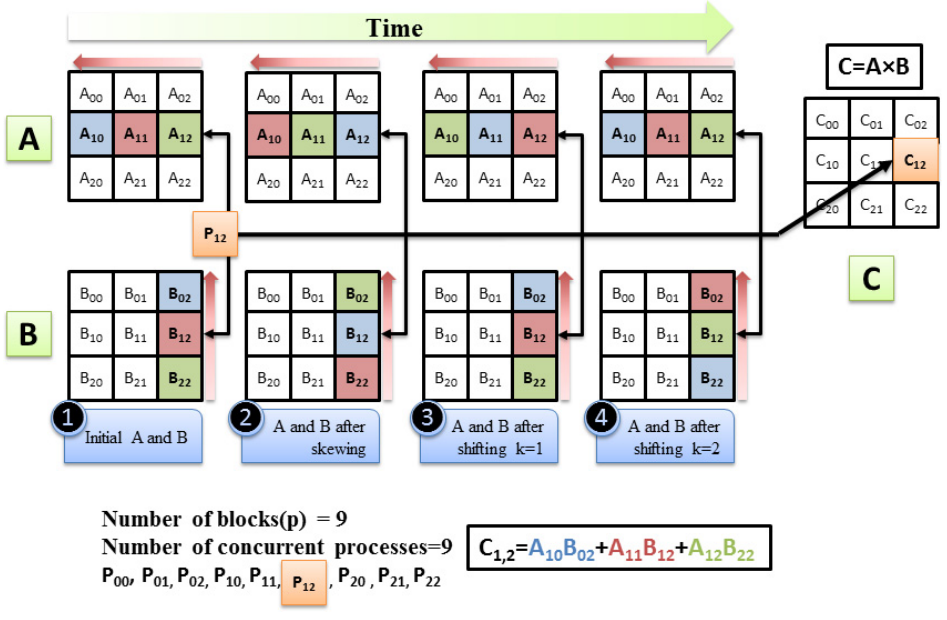


Fig. 14. Calculating one block of result matrix using Cannon's algorithm.

Table 2 shows how we managed to group and reduce the required nine steps of Cannon's algorithm to only five steps to ease the parallelization scheme. Steps 1 and 2 have no dependencies between them because the processes operate on different blocks of A and B matrices. As a result, we can apply both pipelining and loop

Algorithm 7 Cannon's Algorithm

```
1:  procedure
2:      for all i=0 to s-1 do
3:          left-circular-shift row i of A by i
4:      end for
5:      for all i=0 to s-1 do
6:          up-circular-shift B column i of B by i
7:      end for
8:      for k=0 to s-1 do
9:          for all i=0 to s-a and j=0 to s-1 do
10:             C(i,j)← C(i,j) + A(i,j)×B(i,j)
11:             left-circular-shift each row of A by 1
12:             up-circular-shift each row of B by 1
13:         end for
14:     end for
15: end procedure
```

Fig. 15. Pseudocode of matrix multiplication using Cannon's algorithm.

Table 2. Cannon's algorithm applied parallelization techniques.

	Step	Parallelization Technique	Before optimization	After optimization
1. 2.	**Loop1**:for all i=0 to s-1 left-circular-shift row i of A by i **Task1**	Loop pipelining + dataflow + loop unrolling	**O(s^2)**	**O(s)**
3. 4.	for all i=0 to s-1 up-circular-shift B column i of B by i **Task2**	Loop pipelining + dataflow + loop unrolling	**O(s^2)**	**O(s)**
5. 6. 7.	**Loop2:**for k=0 to s-1 for all i=0 to s-1 and j=0 to s-1 C(i,j) = C(i,j) + A(i,j)*B(i,j) **Task3**	Loop unrolling +Loop pipelining +data flow	**O(s^3)**	**O(s^2)**
8.	left-circular-shift each row of A by 1 **Task4**	Loop pipelining	**O(s)**	**O(1)**
9.	up-circular-shift each row of B by 1 **Task5**	Loop pipelining	**O(s)**	**O(1)**

unrolling techniques between them. However, Step 3 cannot be fully parallelized since it depends on the results of both Steps 1 and 2 because it performs its computations after Steps 1 and 2 perform their required shifts. Steps 4 and 5 also cannot perform the required shift operations until the first iteration of the *for* loop is performed. Dataflow technique can be applied here as shown in Table 2 between Step 3 and both Steps 4 and 5. Lastly, Steps 4 and 5 do not depend on each other's outcomes, hence, we can parallelize them using pipelining and loop unrolling techniques. To analytically verify our optimization performance, we use Big O notation to estimate the time complexity of each step of the algorithm before and after the applied optimization techniques as shown in Table 2.

We performed a performance comparison among all available optimization techniques for matrices multiplication in order to choose the best one in terms of

Table 3. Performance comparison of different matrix multiplication optimization techniques.

Technique	Matrix dimension	Time (μs)
Conventional multiplication	256	870
Block multiplication method	256	412
Strassen algorithm	256	271
Cannon's algorithm	256	195

computation time. As shown in Table 3, we chose and applied Cannon's algorithm in our work since it outperformed the other matrices multiplication techniques.

2.5.2. *Matrix inversion process*

Gaussian elimination is one of the most popular methods for solving linear equations.[25] It can be used to calculate the determinant and the inverse of a matrix. The decomposing of the identity I matrix is performed as $(I|A) \rightarrow (I|A^{-1})$ as shown in Eq. (14).

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} : \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{14}$$

Matrix inversion is performed on the square matrix $A = [a_{ij}]$, with n-dimension size, so the inverse calculation is repeated n-times. We note P as the number of iterations and a'_{ij} as the new element in the inverse matrix. This algorithm is based on selecting a pivot diagonally starting from element $a_{1,1}$ to element $a_{n,n}$, then it calculates the determinant of the matrix (denoted by d) through successive multiplications of the selected pivot. However, if the pivot value is zero, then the inverse can't be calculated, otherwise d has values of the determinant of matrix A. The pseudocode shown in Fig. 16 illustrates the algorithm steps.

Algorithm 8 Gaussian Elimination Algorithm for Matrix Inversion

Initialize $P = 0, d = 1$

1: $p \leftarrow P + 1$
2: **if** $a_{p,p} = 0$ **then** ▷ You can't calculate the matrix inverse
3: **go to** Line 14
4: **end if**
5: $d' \leftarrow d \times a_{p,p}$
6: Next, calculate the new value of the pivot row:
7: $a'_{p,j} \leftarrow (\frac{a_{p,j}}{a_{p,p}}), where\ j = 1 \ldots n\ and\ j \neq P$
8: $a'_{i,p} \leftarrow (\frac{a_{i,p}}{a_{p,p}}), where\ i = 1 \ldots n\ and\ i \neq P$
9: $a'_{i,j} \leftarrow a_{i,j} + a_{p,j}a'_{i,p}, where\ i = 1 \ldots n,\ and\ j = 1 \ldots n,\ and\ i\&j \neq P$
10: Next, calculate new element for present pivot : $a'_{p,p} = \frac{1}{a_{p,p}}$
11: **if** $P < 2$ **then**
12: **go to** Line 2
13: **end if**
14: **END**, *When A has inverse values and d has a determinate*

Fig. 16. Gaussian elimination algorithm pseudocode.[4]

Steps 1, 2, and 4 are the initialization steps that can be fully parallelized in one-clock cycle. Step 3 needs to calculate the determinant, and the value of $a_{p,p}$ is changed depending on the number of required iterations. Loop unrolling approach can be applied here to find this value. Steps 5, 6 and 8 are iterative and independent steps where the pipelining approach can be applied. Unfortunately, there is a dependency between steps 7 and 6, and between steps 9 and 10.

After examining lines 3 and 4 carefully in Fig. 12 for EKF equations (Sec. 2.5, Stage 5), there is no dependency between the k matrix and Zm value. Therefore, we can compute these lines concurrently by using the pipelining technique. Line 5 depends on lines 3 and 4 in each iteration (for Stage 5). Therefore, data flow technique can be applied here. Data flow technique improves the throughput and the latency performance by passing the desired values through data flow channel.[21] Similarly, data flow technique is also applied between line 4 and line 5 to pass the value of Zm to $xhatk$ in each iteration.

Lines 7, 8, 9, and 16 require matrices multiplications. Hence, the optimized Cannon's algorithm is applied here to optimize these operations. Lines 11 to 15 are fully parallelized by asserting the values of x, $fhat$, $ahat$ and $thetahat$ in one clock

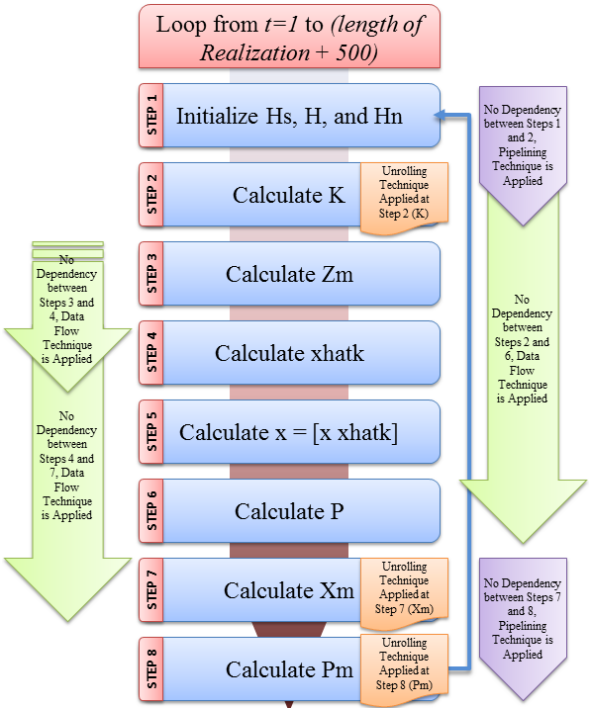


Fig. 17. EKF computational steps and its optimization techniques.

cycle. Figure 17 shows the entire computational steps for implementing the EKF algorithm for a radar-based tracking application. It also shows the data dependency between various operations and their computational sequence, where the proper optimization technique for each step is applied.

3. Simulations and Results

The optimized VHDL components of EKF were synthesized on Xilinx ISE[16] on XC7A100T CSG324 -3FPGA device. Table 4 lists the overall performance results in terms of area, power consumption, and maximum frequency for both high and medium precision profiles (the precision identifies the measurement precision, since we used fixed point implementation; high precision profile identifiers (Integer bits=10, fraction bits=20) and medium precision profile identifiers (Integer bits=8, fraction bits=12). The performance is measured with respect to many evaluation metrics; the throughput is given in terms of the frequency, and hardware utilization is given in terms number of slices, Flip Flop (FF), Lookup table (LUT), BRAM_18K, DSP48E, and the power dissipation values. It is noted that the performance for both directions is the same as in towards the radar and far away from the radar because it can be executed in parallel since there is no dependency between them. It is noted that the hardware resources and computation time are higher for high precision profile than for medium precision profile which is expected.

It is important to analyze and measure the time complexity of the EKF algorithm. This will help in obtaining the bounds and relations for complexity. Big-O notation[26] is applied here to obtain the upper bound on the running time as shown in Table 5. Table 5 summarizes the major steps involved in EKF and shows the time complexity before and after the optimization for each step.

Table 4. Resources utilization and overall implementation performance on Artix7 (XC7A100T CSG324 -3).

	Medium precision profile			High precision profile		
Parameters	Toward radar	Far from radar	Both	Toward radar	Far from radar	Both
Maximum frequency (MHz)	12.89	12.89	12.89	7.78	7.78	7.78
Occupied Slices	3157	2789	3458	5789	5478	5987
Slice LUTs	8200	7907	8654	17898	17256	18978
Slices of FF	2145	2017	2458	2789	2789	2987
Number LUT FF Pairs	8798	8125	8987	17987	17458	19100
DSP48E1s	12	12	15	21	21	29
Number of IOBs	117	117	121	133	133	139
Power Consumption (mW)	2601	2654	2789	3658	3587	4100

Table 5. Time complexity for the major steps in EKF before and after optimization.

Number of lines in Fig. 12	Before optimization	After optimization
3	$2n^{2.376} + 2n^2$	$n^{2.376}/m^* + 1$
4	n^2	1
5	$n^2 + n$	$n + 1$
6	N	1
7	$m \times n$	$1 \times m$
8	$n^2 + n$	$n^2/m + 1$
9	$2n^{2.376} + 2n^2$	$n^{2.376} + 1$

Note: where m is number of blocks and n is the size of the matrix.

To complete the performance evaluation circle and for comparison purposes, EKF was coded in C for serial computations. The C program has been executed on a conventional PC powered by a 2.6 GHz i7-3720QM CPU with memory RAM 8.0 GB (DDR3). The results of the execution times for the i7 processor, GPU, and FPGA implementations are summarized in Table 6. The results show that the FPGA implementations outperform other alternative implementations. The superior performance of the FPGA-based implementations is attributed to the highly parallelized and pipelined architecture inherently present in FPGA.

The results in Table 6 also show the impact of changing the number of targets on the performance. It shows that all implementations achieve, in general, higher speed up with increasing the number of targets due to the exploitation of the GPU and FPGA resources and the available parallelism nature present in the EKF operations. For example, while increasing the number of targets from 50 to 100 targets using FPGA with high precision profile, the speed up from CPU implementation has increased from 34 times to 73 times. GPUs[27] differ from FPGAs in that they can only use the available soft cores on their platforms to execute the assigned tasks, while FPGAs can use all available low-logic components such as multipliers and adders to perform the assigned tasks providing more flexibility to the system design to match

Table 6. Execution times of the different implementations of the object-based radar tracking application using EKF on different platforms.

	Density (Number of targets)		
Implementation	50	100	200
i7-3720QM CPU (ms)	50.45	210.4	390.78
GPU (ms)	2.34 Speed up: 21 times	3.47 Speed up: 60 times	4.29 Speed up: 91 times
Medium precision/FPGA (ms)	1.145 Speed up: 44 times	1.89 Speed up: 111 times	2.69 Speed up: 144 times
High precision/FPGA (ms)	1.45 Speed up: 34 times	2.87 Speed up: 73 times	3.14 Speed up: 124 times

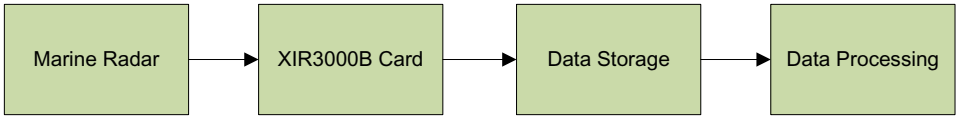


Fig. 18. Experimental setup of the entire system.

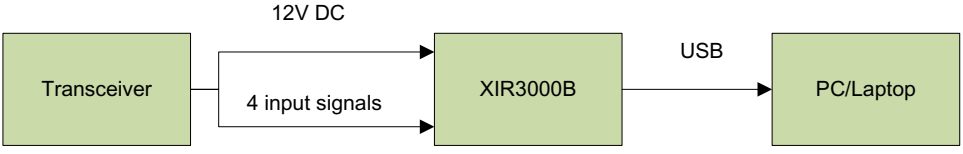


Fig. 19. XIR3000B inputs and output connections.

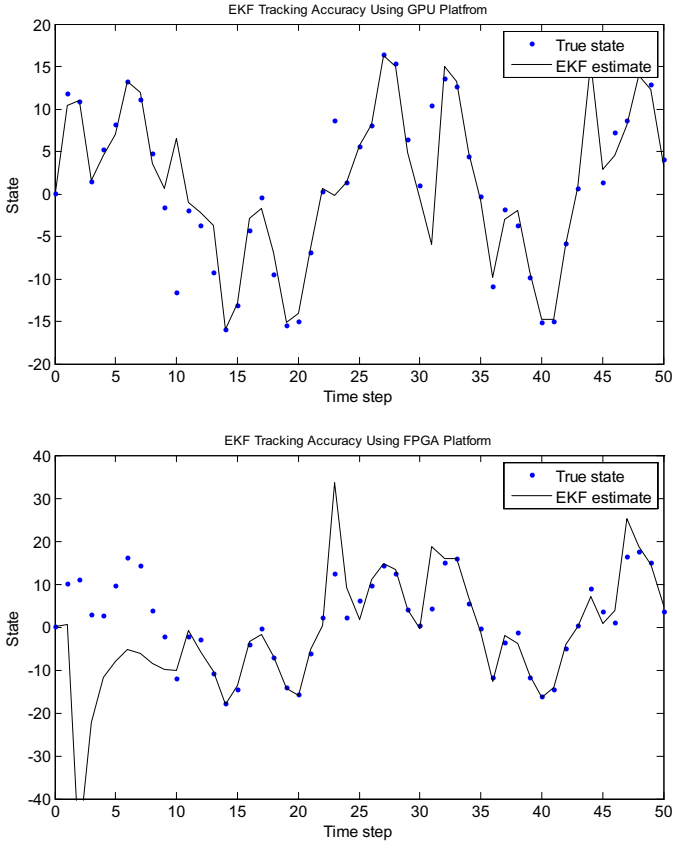


Fig. 20. Tracking accuracy between the true state and the estimated state for both FPGA and GPU.

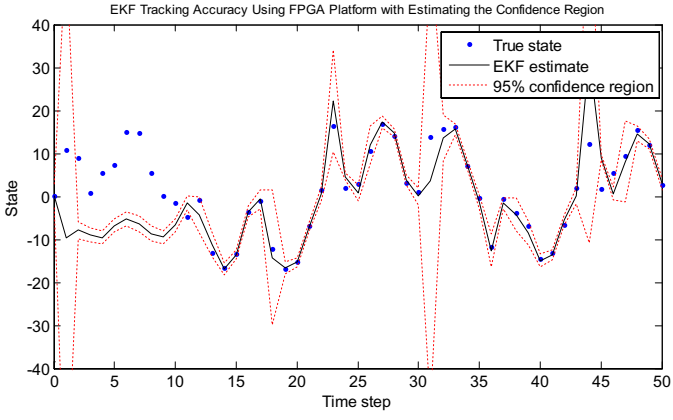


Fig. 21. EKF tracking accuracy using FPGA platform with estimating the confidence region.

system requirements. GPUs are also plagued by the fact that all input data need to be transferred from the CPU to the GPU platform in order to be processed, which results in high latency increase. Therefore, this high level of customizability of FPGA-based platforms is the main reason why FPGA-based platforms normally yield better performance results than their GPU-based counterparts.

To test the performance and the accuracy of EKF on both FPGA and GPU implementations efficiently, a real input sample data (50000 data points) was obtained from the Marine radar project, Communications, Control and Signal Processing Lab, University of Toledo, Ohio, US.[28] Marine radar has been used for bird observation and quantification of their activities since 2012. X-band marine radars with high resolutions are used for bird detection. The radar data are collected using the XIR3000B digitizing card from Russell Technologies. The collected data are then processed and parallelized using FPGA-based and GPU-based platforms for target detection and tracking. The experimental setup of the entire system is shown in Fig. 18. The input ports of the digitizing card are connected to the transceiver, and the output port is connected to the laptop or PC via a USB connection as shown in Fig. 19.

The noisy input real data from Marine radar were applied and processed using EKF on both FPGA-based and GPU-based platforms to examine and verify its accuracy and show the performance of the two platforms. Figure 20 shows the tracking accuracy between the true state, i.e., the ground truth, and the estimated state for both FPGA-based and GPU-based platforms. GPU architecture is a slightly more accurate than FPGA architecture, since FPGA uses fixed-point implementations. Figure 21 shows the EKF estimation performance compared to the true state values with 95% confidence interval/region, which validates the accuracy performance of the optimized EKF operations.

4. Conclusions

In this paper, an efficient implementation is proposed for Extended Kalman Filter by exploring and adopting several optimization techniques and opportunities for parallel processing. Optimization techniques such as loop pipelining, loop unrolling, and dataflow are applied by exploiting the parallelism and the pipelining opportunities between its operations. Moreover, different algorithms are used and optimized in different computational processes in the EKF algorithm, such as Gaussian elimination for matrix inversion and Cannon's algorithm for matrix multiplication. The optimized implementation of the object-based radar tracking application with the proposed techniques is implemented and parallelized on different platforms for comparison purposes such as FPGA, GPU, and single core architectures. Our experimental results show that our optimized implementation of the EKF application can achieve better performance than implementations on other platforms (i.e., GPU and multicore machines), where GPU implementation achieved 91 speed up, FPGA (with medium precision) 144 speed up, and FPGA (with high precision) 124 speed up in performance when compared to multicore implementation (2.6 GHz i7-3720QM CPU). Moreover, higher performance can be achieved with larger data sizes. This is due to the optimization techniques that we have adopted and applied, and the exploited inherent parallelism in EKF operations.

Acknowledgments

It was supported by the Deanship of Scientific Research and Graduate Studies at Yarmouk University, under Grant Number 10/2016.

References

1. J. Gunnarsson, L. Svensson, L. Danielsson and F. Bengtsson, Tracking vehicles using radar detections, *Proc. IEEE Intelligent Vehicles Symposium* (Istanbul, June 2007).
2. D. E. Clark and J. Bell, Bayesian multiple target tracking in forward scan sonar images using the PHD filter, *IEE Radar, Sonar and Navigation*, **152**, 2005, pp. 327–334.
3. D. E. Clark, J. Bell, Y. de S-Pern and Y. Petillot, PHD filter multi-target tracking in 3D sonar, *IEEE Oceans Europe Conf.* (Brest, June 2005). Vol. 1, 20–23 June 2005, pp. 265–270.
4. Matlab support documentation: Online version available at: https://www.mathworks.com/help/dsp/examples/estimating-position-of-an-aircraft-using-kalman-filter.html.
5. Kalman Filter Applications, Subject MI63: Kalman Filter Tank Filling.
6. P. Zarchan and H. Musoff (2015). Fundamentals of Kalman Filtering - A Practical Approach (4th Edition) - Progress in Astronautics and Aeronautics, Volume 246. American Institute of Aeronautics and Astronautics/Aerospace Press. Online version available at: http://app.knovel.com/hotlink/toc/id:kpFKFAPAE2/fundamentals- kalman-filtering/fundamentals-kalman-filtering.

7. A. Al-Tamimi, C. So-In and R. Jain, Modeling and resource allocation for mobile video over WiMAX broadband wireless networks, *IEEE J. Sel. Areas Commun.* **28** (2010) 354–365.
8. A. Al-Tamimi, R. Jain and C. So-In, Modeling and generation of AVC and SVC-TS mobile video traces for broadband access networks, *Proc. 1st Annual ACM SIGMM Conf. Multimedia Systems*, 22 Feb 2010, pp. 89–98.
9. G. Welch and G. Bishop, An introduction to the Kalman Filter, SIGGRAPH, Course 8 (2001).
10. T. Lacey, C. U. Tutorial: The kalman filter. 2013. url: http://mpdc.mae.cornell.edu/Courses/UQ/kf1.pdf.
11. G. Welch and G. Bishop, An introduction to the Kalman filter, TR 95-041 (2006), http://www.cs.unc.edu/welch/media/pdf/Kalman intro.pdf.
12. D. Simon, Kalman filtering, Embedded systems Programming (2007), jettronics.de/trac/projects/export/169/.../Kalman-dan-simon.pdf.
13. S. J. Julier and J. K. Uhlmann, "New extension of the Kalman filter to nonlinear systems." *AeroSense'97*. International Society for Optics and Photonics, 1997.
14. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/WELCH/Kalman.2.html.
15. Virtex-6 FPGA Memory Interface Solutions User Guide (2011), http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf.
16. Xilinx Corporartion (2002), www.xilinx.com.
17. B. Rathod, R. Khadse and M. F. Bagwan, Serial computing vs. parallel computing: A comparative study using MATLAB, *IJCSMC* **3** (2014) 815–820.
18. http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm.
19. C. Hymans, Checking safety properties of behavioral VHDL descriptions by abstract interpretation, *Int. Static Analysis Symp.* (Springer, 2002), pp. 444–460.
20. C. Hymans, Design and implementation of an abstract interpreter for VHDL, *CHARME* (L'Aquila, Italy, 2003), pp. 263–269.
21. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_4/ug902-vivado-high-level-synthesis.pdf.
22. D. A. Bini, Fast matrix multiplication, *Handbook of Linear Algebra*, L. Hogben (ed.), (Chapman & Hall/CRC press, Boca Raton, 2007).
23. Q. Luo and J. Drake, A scalable parallel strassen's matrix multiplication algorithm for distributed memory computers, *Proc. ACM symp. Appl. Comp.*, 1995, pp. 221–226.
24. L. E. Cannon, A cellular computer to implement the Kalman filter algorithm, Ph.D. thesis, Montana State University (1969).
25. D. Parkinson, A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers, *Parallel Comput.* **1** (1984) 65–73.
26. T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 2nd Edn. (MIT Press and McGraw-Hill, 2001). ISBN 0-262-03293-7, pp. 41–50.
27. Nvidia Corporation Geforce GTX 260 http://www.nvidia.com/object/product_geforce_gtx_260_us.html.
28. Communications, Control and Signal Processing, from: http://www.eng.utoledo.edu/eecs/research /groups/com_and_sig_proc.html.